Hand Pose Estimation with Convolutional Networks using RGB-D Data

A Thesis

submitted to the designated
by the General Assembly of Special Composition
of the Department of Computer Science and Engineering
Examination Committee

by

Evangelos Kazakos

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

WITH SPECIALIZATION
IN TECHNOLOGIES - APPLICATIONS

University of Ioannina
February 2017

Dedication

Dedicated to my family.

Acknowledgements

Firstly, I would like to thank my thesis advisor Prof. Christophoros Nikou, for his invaluable advices during the writing of this report. He supported my ideas and helped me significantly to improve them. Our discussions were really stimulating and he was always available for any questions. Besides being an admirable advisor he is also my mentor and I want here to express my deep respect to him.

Thanks are also due to Prof. Aristidis Likas and Prof. Konstantinos Blekas. Through attending their courses of Data Mining and Machine Learning I gained significant knowledge that guided my scientific way of thinking and aided me in writing this paper.

I wish to express my sincere thanks to Prof. Ioannis A. Kakadiaris for offering me the opportunity to have a summer internship at his group at University of Houston. I worked in a multinational simulating environment with advanced research level. During my summer internship I obtained invaluable experience and knowledge since I worked in a challenging research project, which helped me significantly in the writing of this thesis. I also want to thank him for providing me access to the computing resources of University of Houston, which were essential for the implementation of this work.

I acknowledge the use of Opuntia Cluster and the advanced support from the Center of Advanced Computing and Data Systems at the University of Houston to carry out the research presented here.

At this point, I would like to express my great gratitude and love to my parents Sotiris and Maria, and my sister Katerina for their generous support all these years of my studies. They believe in me and support me, which gives me great strength. Their financial and mental aid were the main components thanks to which I came up to this point of my life and therefore being able to write this thesis and finish my master studies.

Last but not least, I am indebted to my friends for being there in difficult as well as delightful moments of my life. We share great discussions and lots of fun which give meaning in my life and will to move forward.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Evangelos Kazakos, M.Sc. in Computer Science, Department of Computer Science and Engineering, University of Ioannina, Greece, February 2017.
Hand Pose Estimation with Convolutional Networks using RGB-D Data.
Advisor: Christophoros Nikou, Associate Professor.

In this work, we study the problem of 3D articulated hand pose estimation from RGB-D images, which consists of estimating all the kinematic parameters of a hand expressed in joint angles or joint positions. Hand pose estimation is a very challenging problem due to the articulated nature of the human hand, which exhibits self-occlusions and large viewpoint variations. The popularization of RGB-D sensors has motivated the interest of the computer vision community in pose estimation as depth images have significantly improved the performance of the related methods. Moreover, the advance of deep learning has spurred this interest and most recent approaches propose convolutional network based methods. The architecture of a convolutional network, its depth as well as its training play a crucial role in its performance. In the first part of our work, we design and evaluate several different convolutional network architectures. Our experiments show that the depth of the network plays a crucial role in the performance, as our deepest convolutional network outperforms the state-of-the-art.

Most methods use single depth images for 3D hand pose estimation. Depth images are noisy with quantization errors that result in missing parts around the hand boundaries. We conjecture that the combination of RGB images, which provide a more accurate description of the hand surface with color and texture information, with depth images, can further improve the performance of a convolutional network. Based on these observations, in the second part of our work we propose fusion methods of RGB and depth information using convolutional networks. We propose three different approaches, input fusion, score level fusion and double-stream architecture fusion. Input level fusion aggregates RGB-D data and trains a convolutional network with images that contain both RGB and depth channels, while score level fusion trains two different convolutional networks with RGB and depth images respectively and fuses their predictions. Finally, double-stream architecture fusion, is based on training two separate convolutional networks in parallel and at any arbitrary layer of the network fusing their feature maps. We employ fusion functions proposed in state-of-the-art activity recognition methods. The performance of input fusion and score level fusion is limited, as they are applied in a very early and a very late stage of the network respectively. We employed double-stream fusion to mitigate this problem since the fusion takes place inside the network and lets subsequent learning stages to define correspondences between RGB and depth features. Indeed, double-stream fusion outperforms input fusion and score level fusion. Double-stream fusion has comparable performance with the state-of-the-art, nevertheless our deep convolutional network trained only with depth images, outperforms double-stream fusion providing us state-of-the-art performance. From our experiments we conclude that RGB-D fusion does not leverage further useful information towards more accurate 3D hand pose estimation.

Εκτεταμενη Περιληψη

Ευάγγελος Καζάκος, Μ.Δ.Ε. στην Πληροφορική, Τμήμα Μηχανικών Η/Υ και Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Φεβρουάριος 2017.
Εκτίμηση του προσανατολισμού του χεριού με Συνελικτικά Δίκτυα χρησιμοποιώντας RGB-D δεδομένα.
Επιβλέπων: Χριστόφορος Νίκου, Αναπληρωτής Καθηγητής.

Η παρούσα εργασία πραγματεύεται το πρόβλημα της 3Δ εκτίμησης του προσανατολισμού των αρθρώσεων ενός ανθρώπινου χεριού (3D articulated hand pose estimation), από RGB-D εικόνες, διαδικασία η οποία συνίσταται της εκτίμησης όλων των κινηματικών παραμέτρων του χεριού, εκφραζόμενες είτε σε γωνίες που σχηματίζουν οι αρθρώσεις, είτε στις θέσεις των αρθρώσεων στον 3Δ χώρο. Η εκτίμηση του προσανατολισμού του χεριού είναι πρόβλημα με πολλές προκλήσεις εξαιτίας της αρθρωτής δομής του ανθρώπινου χεριού η οποία προκαλεί αποκρύψεις και επικαλύψεις μεταξύ διαφορετικών αρθρώσεων και παρουσιάζει μεγάλη διακύμανση ως προς το σύνολο όλων των δυνατών προσανατολισμών των αρθρώσεων. Η εμπορική εμφάνιση χαμηλού κόστους RGB-D αισθητήρων και η διαθεσιμότητα τους στο ευρύ κοινό, έστρεψε το ενδιαφέρον της κοινότητας της Υπολογιστικής Όρασης στο πρόβλημα της εκτίμησης του προσανατολισμού του χεριού, καθώς οι εικόνες βάθους συνέβαλαν σημαντικά στην βελτίωση της απόδοσης των σχετικών μεθόδων. Επιπλέον, η πρόοδος στο πεδίο της Βαθειάς Μηχανικής Μάθησης (Deep Learning) ώθησε αυτό το ενδιαφέρον και οι πιο πρόσφατες προσεγγίσεις προτείνουν μεθόδους βασιζόμενες σε συνελικτικά δίκτυα (Convolutional Networks). Η αρχιτεκτονική ενός συνελικτικού δικτύου, το βάθος του καθώς και η εκπαίδευση του παίζουν πολύ σημαντικό ρόλο στην δυνατότητα του να παρέχει ακριβείς εκτιμήσεις. Στο πρώτο μέρος αυτής της εργασίας σχεδιάζουμε και αξιολογούμε πειραματικά διαφορετικές αρχιτεκτονικές συνελικτικών δικτύων μεταβάλλοντας το βάθος καθώς και άλλες παραμέτρους των δικτύων. Οι πειραματικές μετρήσεις μας, δείχνουν ότι το βάθος του

δικτύου παίζει καθοριστικό ρόλο στην απόδοση του, όπου το πιο βαθύ συνελικτικό μας δίκτυο σημειώνει καλύτερη επίδοση από την τρέχουσα πρόοδο της τεχνολογίας.

Οι περισσότερες σκαταμέθοδοι χρησιμοποιούν μόνο εικόνες βάθους για την 3Δ εκτίμηση του προσανατολισμού του χεριού. Οι εικόνες βάθους είναι θορυβώδεις και περιέχουν σφάλματα κβαντοποίησης τα οποία οδηγούν σε ασυνέχειες βάθους. Ως αποτέλεσμα, σε κάποια εικονοστοιχεία απουσιάζουν οι τιμές βάθους. Αυτές οι ασυνέχειες συμβαίνουν σε περιοχές γύρω από τα όρια του χεριού και οδηγούν στην απουσία τμημάτων τις εικόνας σε περιοχές γύρω από τα όρια του χεριού. Υποθέτουμε ότι ο συνδυασμός RGB εικόνων, οι οποίες παρέχουν πιο ακριβή περιγραφή της επιφάνειας του χεριού με πληροφορία χρώματος και υφής, με εικόνες βάθους, μπορεί να βελτιώσει περαιτέρω την απόδοση ενός συνελικτικού δικτύου.

Βασιζόμενοι σε αυτές τις παρατηρήσεις, στο δεύτερο κομμάτι της παρούσας εργασίας προτείνουμε μεθόδους συγχώνευσης πληροφορίας RGB και πληροφορίας βάθους με τη χρήση συνελικτικών δικτύων. Προτείνουμε τρεις διαφορετικές προσεγγίσεις, την συγχώνευση των εισόδων, την συγχώνευση των εκτιμήσεων και τη συγχώνευση διπλής αρχιτεκτονικής. Η μέθοδος της συγχώνευσης των εισόδων συσσωματώνει RGB-D εικόνες και εκπαιδεύει ένα συνελικτικό δίκτυο με εικόνες που περιέχουν κανάλια τόσο RGB όσο και βάθους. Η μέθοδος της συγχώνευσης των εκτιμήσεων εκπαιδεύει δύο διαφορετικά νευρωνικά δίκτυα με εικόνες RGB και βάθους αντίστοιχα και συγχωνεύει τις προβλέψεις τους. Τέλος η συγχώνευση διπλής αρχιτεκτονικής βασίζετε στην εκπαίδευση δύο διαφορετικών συνελικτικών δικτύων παράλληλα και σε οποιοδήποτε αυθαίρετο επίπεδο του δικτύου να συγχωνεύει τους χάρτες χαρακτηριστικών τους, με δοθείσες συναρτήσεις συγχώνευσης χαρτών χαρακτηριστικών. Χρησιμοποιούμε συναρτήσεις συγχώνευσης οι οποίες έχουν προταθεί σε μεθόδους αναγνώρισης ανθρώπινης δραστηριότητας οι οποίες είναι τελευταία πρόοδος της τεχνολογίας. Η επίδοση των μεθόδων της συγχώνευσης στο επίπεδο της εισόδου και της συγχώνευσης των εκτιμήσεων είναι περιορισμένη, καθώς η συγχώνευση εφαρμόζεται σε ένα πολύ αρχικό και σε ένα πολύ τελικό επίπεδο του δικτύου αντίστοιχα. Προτείναμε την συγχώνευση διπλής αρχιτεκτονικής ώστε να αντιμετωπίσει αυτό το πρόβλημα καθώς σε αυτήν την περίπτωση η συγχώνευση λαμβάνει μέρος στο εσωτερικό του δικτύου και επιτρέπει στα επακόλουθα στάδια μάθησης, να ορίσουν αντιστοιχίες μεταξύ RGB χαρακτηριστικών και χαρακτηριστικών βάθους. Πράγματι, η συγχώνευση διπλής αρχιτεκτονικής ξεπερνάει σε ακρίβεια τη μέθοδο συγχώνευσης των εισόδων και τη μέθοδο συγχώνευσης των εκτιμήσεων.

Η μέθοδος συγχώνευσης διπλής αρχιτεκτονικής έχει συγκρίσιμες επιδόσεις με την τρέχουσα πρόοδο της τεχνολογίας, παρόλα αυτά το βαθύ συνελικτικό δίκτυο που προτείνουμε το οποίο εκπαιδεύτηκε μόνο με εικόνες βάθους, ξεπερνάει τις επιδόσεις των μεθόδων συγχώνευσης, παρέχοντας μας αποτελέσματα τελευταίας προόδου της τεχνολογίας. Από τα πειράματα μας συμπεραίνουμε ότι η συγχώνευση RGB-D δεδομένων δεν εκμεταλλεύεται επιπλέον χρήσιμη πληροφορία για πιο ακριβής 3Δ εκτίμηση του προσανατολισμού του χεριού.

Chapter 1

Introduction

In this work, we study the problem of 3D articulated hand pose estimation from RGB-D images, which consists of estimating all the kinematic parameters of a hand expressed in joint angles or joint positions. Hand pose estimation has several useful applications, such as in human computer interaction (HCI), augmented reality, gesture recognition, gaming, as well as robots learning by demonstration. It is a very challenging problem, since hand articulation has high degrees of freedom, there are self-occlusions between joints which can make the estimation of the position very difficult and the projected image of a human hand is very small which results in low resolution images. The popularization of RGB-D sensors has motivated the interest of the computer vision community in pose estimation as depth images have significantly improved the performance of the related methods. Depth images are very good features for 3D pose estimation since they provide 3D information which is directly correlated with the estimation of 3D joint positions.

The release of large annotated datasets of depth images with human bodies, made discriminative, classification based methods arise and give outstanding performance in the problem of human pose estimation. These methods were based on random decision forests (RDF) trained with depth images, where at the first step an RDF classifies pixels in body parts and subsequently a further step estimates the position of the body joints in each part. These methods were also applied in hand pose estimation with less success. The reasons are that the human hand has higher degrees of freedom, it exhibits much wider pose variation, viewpoint change and self-occlusions which can significantly influence the performance of the classifier, and hence the subsequent estimation of the position of the joints. Regression based methods were applied with greater success than classification based methods for hand pose estimation. Regression based methods do not rely on a subsequent estimation step, they learn a direct mapping from a depth image to a 3D hand pose; hence, they can mitigate the problem of self-occlusion and recover the hand pose from occluded joints since they can learn meaningful mappings from occluded joints to the ground truth pose. Random regression forests and several proposed variants have shown state-of-the-art performance in the problem of hand pose estimation with depth images.

The advance of deep learning has influenced significantly computer vision were convolutional networks gave impressive results in several tasks such as image recognition and detection. As a result, most recent hand pose estimation methods turned their attention towards convolutional networks, where they outperform previous regression forest based methods. In this work, we study the problem of 3D hand pose estimation with convolutional networks using RGB-D data. In the first part of the work, we make a thorough analysis of basic concepts related with feedforward neural networks and convolutional networks. We analyze their main properties and we discuss several widely used practices. Since training neural networks can be a very difficult task we pay special attention to methodologies used for training neural networks. Subsequently, we introduce the problem of hand pose estimation and we discuss several difficulties of the problem that make it quite challenging. We present, thoroughly previous related work in hand pose estimation and we analyze more extensively the methods based on convolutional networks.

In the second part of our work, we introduce our main methodology for 3D hand pose estimation with convolutional networks. The depth of a convolutional network as well as its general structure, such as the size of the convolutional kernels and their number, are of great importance concerning its performance. To this end, in the first part of our proposed methodology, we design and evaluate several different convolutional network architectures by alternating the depth of the network as well as other parameters that change the structure of the network. Our experimental results show that the depth of a network plays a crucial role in its performance as our deepest convolutional network outperforms the state-of-the-art. This convolutional network is our proposed architecture that we use in the rest of our experiments. Apart from the depth we find that the amount of pooling layers is of great importance and an excessive number can lead to information loss and downgraded performance. Moreover, we found that dropout serves as a very good regularizer for regression convolutional networks, which prevents very effectively our deep architecture from overfitting with very small inclusion probabilities. Nevertheless, we observed that dropout has a very strong regularization effect on regression convolutional networks and very small dropout probabilities should be used, otherwise the training of the network can very easily face an underfitting scenario. Finally, we found that several hyperparameters related to the network training and regularization, such as the learning rate and dropout probabilities respectively, are too sensitive and slightly different values can easily lead to underfitting or overfitting. To this end, we performed hyperparameter optimization with cross-validation in order to find good settings for some of the network hyperparameters.

Most methods use single depth images for 3D hand pose estimation. Depth images are noisy with quantization errors that result in missing parts around the hand boundaries. We conjecture that the combination of RGB and depth images can improve the performance of convolutional networks, since the benefits of one domain are complementary with the drawbacks of the other. RGB images provide an accurate description of objects with color and texture information, yet they do not contain 3D information. Depth images are noisy with an imprecise description of the objects, but with 3D information that proved very useful in hand pose estimation. Based on these observations, in the second part of our methodology we propose fusion methods of RGB and depth information using convolutional networks. We propose three different approaches: input fusion, score level fusion and double-stream architecture fusion. Input level fusion aggregates RGB-D data and trains a convolutional network with images that contain both RGB and depth channels, while score level fusion trains two different convolutional networks with RGB and depth images respectively and fuses their predictions. Finally, double-stream fusion architecture, is based on training two separate convolutional networks in parallel whose feature maps are fused at an intermediate layer of the network using a fusion function. We employ fusion functions proposed in state-of-the-art activity recognition methods.

The performance of input fusion and score level fusion is limited, as they are applied in a very early and a very late stage of the network respectively. We believe that double-stream fusion can mitigate this problem since the fusion takes place inside the network and lets subsequent learning stages to define correspondences between

RGB and depth features. Our experiments confirm that double-stream architecture fusion outperforms both input fusion and score level fusion. Double-stream architecture fusion performs quite comparably with the state-of-the-art, but still our proposed convolutional network which is trained only with depth images outperforms double-stream architecture fusion and provide us state-of-the-art performance. From our experiments we conclude that fusion of RGB and depth information do not leverage further useful information towards more accurate 3D hand pose estimation. Nevertheless, our knowledge of the problem is limited since we did not perform experiments for fusing double-stream architectures in multiple different layers of the network and we did not consider every possible fusion function. More investigation towards that direction may reveal better correspondences between RGB and depth features and provide improved performance comparing to training a network only with depth images.

Chapter 2

Feedforward neural networks and deep learning

2.1 Introduction

Deep learning is a machine learning field that researches the construction of gradient-based learning models for supervised and unsupervised learning. Here, we investigate supervised learning where given a dataset $X = \{\boldsymbol{x}^{(i)}, y^{(i)}\}_{i=1}^{N}$, where N is the number of samples, $\boldsymbol{x}^{(i)}$ is the input vector and $y^{(i)}$ is the target ($y \in \mathbb{R}$ for regression, $y \in \mathbb{Z}$ for classification) for the i-th example, the goal is to approximate a function f^* where $y = f^*(\boldsymbol{x})$. *Deep feedforward networks*, also often called *feedforward neural networks* or *multilayer perceptrons* (MLPs), are the quientessential deep learning models.

A feedforward network defines a mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ such that $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$, where $J(\boldsymbol{\theta})$ is a cost function over the dataset. The name deep learning arises from the fact that unlikely classical neural networks, deep networks are composed of several hidden layers, thus they are **deeper**, leading to the approximation of more complex functions that can solve more complex tasks. The idea behind this is that we allow computers to understand the world in terms of a hierarchy of concepts, with each concept defined in terms of its relation to simpler concepts. In terms of neural networks, we learn hierarchical feature representations, through each layer by composing more complex features from simpler ones. This has the big advantage that we don't have to manually obtain traditional hand-crafted features for a task but the model learn these features from our data. This framework is called *representation learning* or *feature learning*.

Providing sufficient large models, deep networks can solve very complex problems, as they can approximate more complex functions. Nevertheless, very complex models are prone to overfitting. Thus, very good regularization methods are needed in order deep models to perform well. As we know large datasets can reduce the overfitting of a machine learning model and improve its generalization performance. Recent publications of big annotated datasets gave the opportunity to deep learning to arise and give outstanding performance in several tasks such as speech recognition and image recognition and detection. However, even with the use of sufficiently big amount of data, overfitting remains a major problem in deep models. Thus, there is extensive research on regularization methods in deep learning literature. Another major problem in training deep neural networks is that optimization is harder, which can lead to underfitting, thus better optimization methods are needed in order to obtain deep models with good performance.

In this chapter, we will analyze the basic principles of training feedforward neural networks. Furthermore, we will describe some techiniques that are developed in deep learning research in order to tackle the aforementioned problems. We will not go thoroughly through many deep learning methodologies, such as unsupervised pre-training, restricted Boltzman machines, autoencoders etc, but we will limit our discussion to methods that are involved in training convolutional networks which is the basic model we are studying in this work.

2.2 Definition of a feedforward neural network

A feedforward neural network is a network composed of computational units which are called *neurons*. Neurons are organized in layers, where there are no connections between neurons of the same layer and full connections between neurons of succesive layers (each neuron in a layer is connected with all the neurons in the previous layer). They are called feedforward neural networks, because all the computations follow the same direction, from the input to the output and there are no feedback connections of the output of a neuron to other neurons; hence they can be interpreted by a directed acyclic graph . They are composed from the input layer, one or more hidden layers and the ouput layer.

Given an input vector $\boldsymbol{x} \in \mathbb{R}^d$, the input layer takes this vector and passes it to the next layer of the network, where the hidden layers perform computations and finally the output layer computes an output $\boldsymbol{f}(\boldsymbol{x})$ based on $\boldsymbol{x}$. Assume that we have L hidden layers. At a given layer $k > 0$, each hidden unit first computes its *preactivation* (or input activation), that is:

$$a^{(k)}(\boldsymbol{x})_i = b_i^{(k)} + \sum_j W_{(i,j)}^{(k)} h^{(k-1)}(\boldsymbol{x})_j, \tag{2.1}$$

where $a^{(k)}(\boldsymbol{x})_i$ is the preactivation of the i-th neuron in layer k and $h^{(k-1)}(\boldsymbol{x})_j$ is the output of the j-th hidden unit in the layer below. $\boldsymbol{W}$ is referred to as the *connection matrix* which contains the weights of the connections between neurons, where $W_{(i,j)}^{(k)}$ is the weight that connects the i-th hidden unit with its j-th input, and $\boldsymbol{b}$ is the *bias* vector where $b_i^{(k)}$ is the bias of the i-th hidden unit in layer k. The computation in equation (2.1) is the inner product of the weights with the input plus the bias term. Alternatively, equation (2.1) can be written in vector form as:

$$\boldsymbol{a}^{(k)}(\boldsymbol{x}) = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}(\boldsymbol{x}), \tag{2.2}$$

where $\boldsymbol{a}^{(k)}(\boldsymbol{x})$ is the vector that contains the preactivations of all neurons in layer k. For the input layer, $\boldsymbol{h}^{(0)}(\boldsymbol{x}) = \boldsymbol{x}$.

Given the preactivations, hidden units compute their activations that in vector form is:

$$\boldsymbol{h}^{(k)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{a}^{(k)}(\boldsymbol{x})), \tag{2.3}$$

where $k = 1, ..., L$, $g(\cdot)$ is a nonlinear function that transforms the preactivation of a hidden unit, known as the *nonlinearity* of the hidden unit. Later in this chapter, we

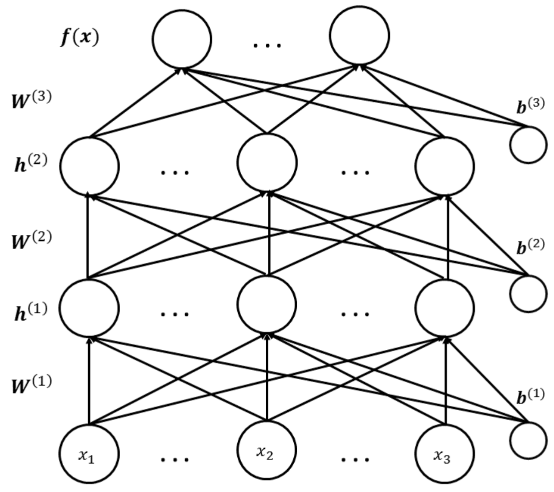


Figure 2.1: A typical feedforward neural network. $\boldsymbol{x}$ is the input vector, $\boldsymbol{h}^{(k)}(\boldsymbol{x})$ are the activations of the k-th layer, $\boldsymbol{W}^{(k)}$ is the weight matrix of the k-th layer, $\boldsymbol{b}^{(k)}$ are the biases of the layer, and $\boldsymbol{f}(\boldsymbol{x})$ is the output of the neural network. In the specific example the network has two hidden layers.

will see some popular choices of activation functions. The output of the network is computed as:

$$\boldsymbol{h}^{(L+1)} = \boldsymbol{o}(\boldsymbol{a}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{f}(\boldsymbol{x}), \tag{2.4}$$

where o is a nonlinear function of the output preactivations, i.e. the activation of the output units, and we say that $\boldsymbol{f}(\boldsymbol{x})$ is the output of the neural network. Additionally $\boldsymbol{o} \in \mathbb{R}^p$, where p is the number of outputs of the network. We use the symbol $\boldsymbol{f}$ for the output of the neural network, as in the general case it can have multiple outputs; hence, $\boldsymbol{f}$ is the vector with the outputs of the neural network. In the next section, we will see some reasonable choices for output activations depending on different kind of problems. For a graphical illustration of a neural network see figure 2.1. The computation of $\boldsymbol{f}(\boldsymbol{x})$ given an input vector $\boldsymbol{x}$ is called forward propagation.

Neural networks were invented based on perceptron [17], hence their alternative name multilayer perceptrons. A perceptron is actually a single neuron which computes a linear combination of an input vector with some weights and then pass it through a nonlinearity to make decisions. The limitation of perceptron is that it can compute only linear decision surfaces. Hence it can solve only linearly separable problems. Neural networks overcome this problem by combining multiple neurons organized in layers in order to obtain more complex functions constructed by simpler ones, i.e. by multiple linear functions followed by nonlinearities. For example, if we have a neural network with three layers including the output layer and each layer can be represented as a function $\boldsymbol{f}^i$ which take as input the previous layer, we result in $\boldsymbol{f}^3(\boldsymbol{f}^2(\boldsymbol{f}^1(\boldsymbol{x})))$. Thus, they create nonlinear decision surfaces which are suitable for solving nonlinearly separable problems. Actually, the hidden layers transform the input space by computing feature representations of the input that attempt to make the problem linearly separable. These feature representations are then fed to the output layer which is essentially a linear classifier which solves the problem using linear decision surfaces in the new space. Therefore, these feature representations make the problem easier so that the linear classifier at the output layer pay less effort to solve the problem. It arises that the hidden layers compute powerful representations with high discrimination level. The activation functions in the hidden units make the neural network able to construct functions with higher degree of nonlinearity; thus, the decision surfaces are even more complex which makes the network able to solve more challenging problems.

Neural networks define a mapping $\boldsymbol{x}^d \to \boldsymbol{f}(\boldsymbol{x})^p$. This mapping is learned from the dataset we provide to the training algorithm. Training is the procedure of learning the parameters of the model which in neural networks are:

$$\boldsymbol{\theta} \equiv \{\boldsymbol{W}^{(1)}, \boldsymbol{b}^{(1)}, \boldsymbol{W}^{(2)}, \boldsymbol{b}^{(2)}, ..., \boldsymbol{W}^{(L+1)}, \boldsymbol{b}^{(L+1)}\}.$$

In the rest of this chapter, we will discuss about several aspects of training neural networks including some modern practices applied on deep learning.

2.3 Activation functions

Here, we will discuss several available options of activation functions for hidden units as well as output units. Output units are selected based on the task we assign to the neural network, which affects the output of the model, e.g. classification requires integer outputs while regression requires continuous outputs. The choice of the activation function for the hidden units on the other hand is not straightforward as we do not know in advance the optimal values for the hidden units. The design of hidden units is an extremely active area of research and does not yet have many definitive guiding theoretical principles. It can be difficult to determine when to use which kind. It is usually impossible to predict in advance which will work best. The design process consists of trial and error, intuiting that a kind of hidden unit may work well, and then training a network with that kind of hidden unit and evaluating its performance on a validation set.

We will first give the definitions of some famous activations functions and then we will describe the cases where each one is suitable.

The first and simplest activation function that can be used is the linear activation function, that is:

$$g(a) = a, \tag{2.5}$$

which takes the input and reproduces it. A graphical illustration of a linear activation function can be seen in figure 2.2a. Another interesting choice of activation function is the sigmoid activation function which is:

$$g(a) = \sigma(a) = \frac{1}{1 + \exp(-a)}. \tag{2.6}$$

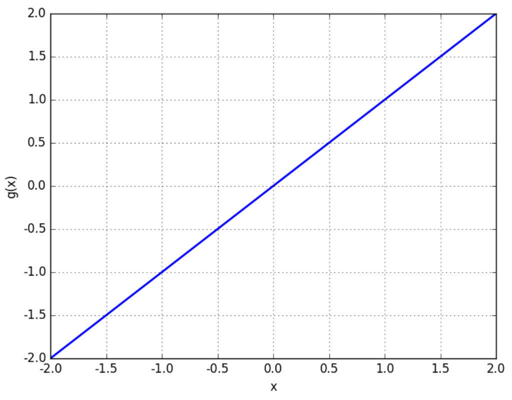
Sigmoids squash the neuron's preactivation in the range $[0, 1]$. Sigmoid's graph is depicted in figure 2.2d. Hyperbolic tangent or tanh activation function is very similar to sigmoid. Their difference is that a tanh activation function squashes a neuron's range in $[-1, 1]$. Its type is given by:

$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}, \tag{2.7}$$

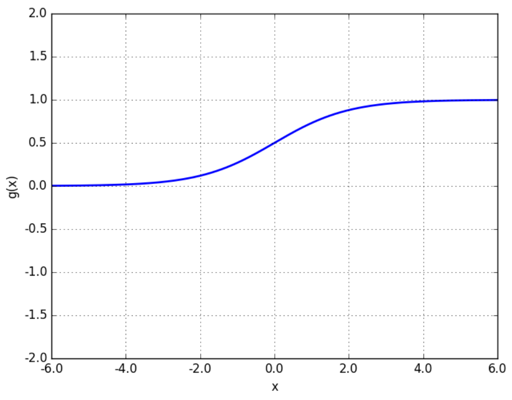
and for its graph you can have a look at figure 2.2c. Finally a very popular choice for activation functions especially in the context of deep learning is the Rectified Linear Unit activation function or ReLU which is given by:

$$g(a) = \text{ReLU}(a) = max(0, a), \tag{2.8}$$

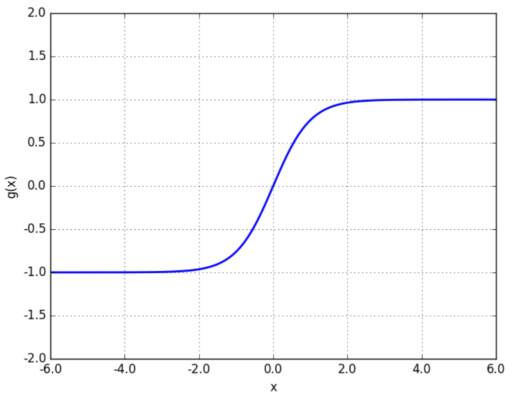
which is linear for $a > 0$ and zero for $a < 0$. Its plot is illustrated in figure 2.2d. This particular choice of activation function behaves differently from the aforementioned activation functions. It tends to give neurons with sparse activities, which means that neurons are often exactly zero and that is because there is a very big area in its domain where the activation is zero. That doesn't happen with the other hidden units, e.g. in sigmoid a neuron is zero when its preactivation is $-\infty$ which practically will not happen and in tanh only when it takes the particular value of zero. We will see that this is an appealing property and ReLU works quite well in practice.



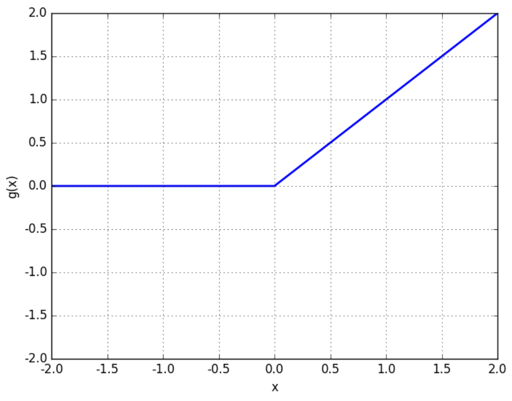
(a) Linear



(b) Sigmoid



(c) Tanh



(d) Rectified Linear Unit (ReLU)

Figure 2.2: Activation functions of neurons.

Most modern neural networks are trained using maximum likelihood. That is, our neural network defines a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ of predicting $\boldsymbol{y}$ given the input $\boldsymbol{x}$ and the parameters of our model $\boldsymbol{\theta}$. The cost function that we are minimizing is the negative log-likelihood of our data over $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$. The maximum likelihood approach is to define the correct distribution based on the type of $\boldsymbol{y}$. Below, we describe the proper selection of the output units based on the selection of the probability distributions under a maximum likelihood framework.

In a regression problem, it is typical to model the conditional distribution with a Gaussian distribution where the mean of the Gaussian is our model's predictions so that:

$$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{f}(\boldsymbol{x}), \boldsymbol{I}), \tag{2.9}$$

where $\boldsymbol{I}$ is a unit covariance matrix. Minimizing the negative log-likelihood is then equivalent to minimizing the mean squared error. A reasonable choice is to use a linear activation function for modeling the mean of the distribution, i.e $\boldsymbol{f}(\boldsymbol{x})$. As we want the model to be able to predict any real value it wouldn't be wise to use, say a sigmoid or a tanh which would suppress the outputs in a specific range. On the other hand, if we know that in our dataset, $\boldsymbol{y}$ takes values between say 0 and 1, a sigmoid would be a convenient choice. Thus, depending on the range of the targets different activation functions can be used but normally the default for regression is a linear one.

In the case of binary classification, our targets can take values $y \in \{0, 1\}$. In that case, the maximum-likelihood approach is to define a Bernoulli distribution over y conditioned on $\boldsymbol{x}$. Therefore, it is sufficient to predict only $p(y = 1|\boldsymbol{x})$ as $p(y = 0|\boldsymbol{x}) = 1 - p(y = 1|\boldsymbol{x})$. Sigmoid units can be used in this case, because they produce values bounded in $[0, 1]$ which is the range for valid probabilities. Thus, we can have a single sigmoid unit as output that predicts $p(y = 1|\boldsymbol{x})$, and if $p(y = 1|\boldsymbol{x}) > 0.5$, $\boldsymbol{x}$ is estimated as belonging to the first class else to the second one.

In multi-class classification, we need a probability distribution over a discrete variable with n possible values, which correspond to n different classes. We now need our network to produce a vector of estimates $\boldsymbol{f}(\boldsymbol{x})$ with $f(\boldsymbol{x})_c = p(y = c|\boldsymbol{x})$. We require not only that each element of $f(\boldsymbol{x})_c$ be between 0 and 1 , but also that the entire vector sums to 1 so that it represents a valid probability distribution. The same approach that worked for the Bernoulli distribution generalizes to the categorical distribution. We model the categorical distribution with a softmax activation function:

$$o(a_i) = \text{softmax}(a_i) = \frac{\exp a_i}{\sum_j \exp a_c}. \tag{2.10}$$

Probability distributions based on exponentiation and normalization are common throughout the statistical modeling literature, where we normalize over unnormalized probability distributions. Here, the unnormalized probability distribution is a_i, that is the preactivation of the output neurons. A similar procedure of exponentiating and normalizing is followed to model sigmoids as Bernoulli distribution in the case of binary classification.

Now, we will continue our discussion for the hidden units of a neural network. A typical problem in optimizing deep neural networks is the *vanishing gradients* problem. The vanishing gradient problem is that as we back-propagate the gradients to our network (we talk about back-propagation and how is related to the vanishing gradient problem in Section 2.6), the gradients take very small values, and as we reach to the shallower layers of the network the gradients become nearly zero. This has the effect that the information vanishes and the network does not learn anything useful. Part of this problem is due to the activation functions of the hidden units.

Rectified linear units are an excellent default choice for hidden units. ReLUs are easy to optimize because they are similar to linear units. The only difference between a linear unit and a rectified linear unit is that a rectified linear unit outputs zero across half of domain. This has the effect that whenever the preactivation is positive, ReLU outputs a strong signal and also results in large gradients. ReLus were designed to overcome the vanishing gradient problem which as will see below, sigmoidal units were suffering from vanishing gradients. As long as the preactivation is positive ReLUs gradients are large and back-propagate normally through the network. For hidden units that suffer from the vanishing gradient problem, we also say that these hidden units *saturate*, that is, at big part of their domain their gradients become zero. ReLU is a non-saturating hidden unit and for this reason is preferred in modern deep model architectures.

Several generalizations of rectified linear units exist. Most of these generalizations perform comparably to rectified linear units and occasionally perform better. One drawback of rectified linear units is that they cannot learn via gradient-based methods on examples for which their preactivations are negative. A variety of generalizations of rectified linear units guarantee that they receive gradients everywhere.

Three generalizations of rectified linear units are based on using a non-zero slope β when $a < 0 : g(a, \beta) = max(0, a) + \beta \min(0, a)$. Absolute value rectification fixes $\beta = -1$ to obtain $g(a) = |a|$. It is used for object recognition from images [18], where it makes sense to seek features that are invariant under a polarity reversal of the input illumination. Other generalizations of rectified linear units are more broadly applicable. A leaky ReLU [19] fixes β to a small value like 0.01 while a parametric ReLU or PReLU treats β as a learnable parameter [20].

Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function or the hyperbolic tangent activation function. These activation functions are closely related because $\tanh(a) = 2\sigma(2a) - 1$. We have already seen sigmoid units as output units, used to predict the probability that a binary variable is 1 . Unlike piecewise linear units, sigmoidal units saturate across most of their domain-they saturate to a high value when a has a high positive value, saturate to a low value when a has a high negative value, and are only strongly sensitive to their input when a is near 0. The widespread saturation of sigmoidal units can make gradient-based learning very difficult. For this reason, their use as hidden units in feedforward networks is now discouraged.

2.4 Loss functions

In section 2.3 we mentioned that we can model the output of a neural network as a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and train it with maximum likelihood. Consider a set of m examples $\boldsymbol{X} = \{(\boldsymbol{x}^{(1)}, y^{(1)}), ..., (\boldsymbol{x}^{(m)}, y^{(m)})\}$. Given $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$, the conditional maximum likelihood is:

$$\boldsymbol{\theta}_{ML} = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{i=1}^{m} \log p(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}). \tag{2.11}$$

We convert it into a minimization problem by changing the sign in the last equation. Hence, we are minimizing the negative log-likelihood of our data which results in the cost function:

$$J(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \log p(\boldsymbol{y}^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{\theta}). \tag{2.12}$$

We can express $J(\boldsymbol{\theta})$ as an expectation over the dataset $\boldsymbol{X}$ by dividing the last equation with m which results in:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{\boldsymbol{X}} \log p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta}). \tag{2.13}$$

In regression, where we set $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \boldsymbol{f}(\boldsymbol{x}), \boldsymbol{I})$, our objective function takes the form:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\mathbb{E}_{\boldsymbol{X}} \|\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})\|^2, \tag{2.14}$$

where we omitted the constant term that arises from the normalization factor of the Gaussian distribution. Hence, the per-example loss function is:

$$L(f(\boldsymbol{x}), \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta})\|^2. \tag{2.15}$$

In classification, the output units model directly the conditional probability distributions, that is $p(\boldsymbol{y}|\boldsymbol{x}) = f(\boldsymbol{x})$. Substituting in equation (2.13) the loss function now becomes:

$$L(f(\boldsymbol{x}), y) = -\sum_c \mathbb{1}_{(y=c)} \log f(\boldsymbol{x})_c = -\log \boldsymbol{f}(\boldsymbol{x})_y, \tag{2.16}$$

that is the negative log-likelihood of our model for the true class y of $\boldsymbol{x}$, or it can be seen as the cross entropy between the distribution $\mathbb{1}_{(y=c)}$ (identity function) which gives 1 for the true class of $\boldsymbol{x}$ and 0 elsewhere, and our model distribution $\boldsymbol{f}(\boldsymbol{x})$.

We saw previously that sigmoidal units saturate, and hence their use as hidden units is discouraged. Nevertheless, they can be used as output units when an appropriate cost function can undo the saturation of the sigmoid or tanh in the output layer. This is why modern deep network training is modeled under a maximum likelihood estimation framework. The logarithm in the cost function cancels the exponential in sigmoidal units or the softmax, thus with large values these output units will not saturate and the gradients will not vanish.

2.5 Empirical risk minimization

The goal of a machine learning algorithm is to reduce the expected generalization error that is:

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{p(\boldsymbol{x},y)} L(f(\boldsymbol{x}), y) = \int L(f(\boldsymbol{x}), y) p(\boldsymbol{x}, y), \tag{2.17}$$

where $p(\boldsymbol{x}, y)$ is the true underlying data distribution. This quantity is known as the risk. Ideally, we would like to minimize the risk, but the true underlying distribution of our data is not known. To this end, we minimize the empirical risk:

$$\mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x}), y))] = \frac{1}{m} \sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)}), y^{(i)})), \tag{2.18}$$

where $\hat{p}(\boldsymbol{x}, y)$ is the empirical distribution defined by our training set and m is the number of training examples.

Ideally, we would like to optimize the classification error, which is a very hard function to optimize as it has discontinuity at zero and everywhere else its gradient is zero which means that a gradient-based optimization algorithm is not feasible. As a result we minimize a surrogate loss function which in our case is the negative log-likelihood.

2.6 Back-propagation algorithm

As explained in Section 2.2, forward propagation consists of providing an input vector $\boldsymbol{x}$ to the input layer, which propagates it to the subsequent layers, where the hidden units compute their preactivations and activations until the output layer computes $\boldsymbol{f}(\boldsymbol{x})$, i.e. the prediction of the network for $\boldsymbol{x}$. Forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. Given $J(\boldsymbol{\theta})$, *back-propagation algorithm* allows the information to flow backwards, from the output to the input of the network, in order to compute the gradients of the network parameters w.r.t. $J(\boldsymbol{\theta})$. Of course, it would be possible to compute analytical expressions for the gradient, but the evaluation of these expressions would be very expensive. Back-propagation algorithm performs these computations in a very computationally efficient way.

Many times the back-propagation algorithm is misunderstood as being the full training procedure for neural networks. In fact, back-propagation is just the procedure for computing the gradients of the neural network, which will be used by an optimization procedure to minimize our cost function.

Making the full derivation of the parameter gradients of the neural network is a complicated procedure; hence, we will break it in multiple steps. Finally, we will give the back-propagation algorithm which is composed from all these single steps.

We start by deriving the gradient of the loss w.r.t. the output of the neural network. In this example, we make the derivation for the classification loss. It is fairly simple to derive it for regression loss as well. It is:

$$\frac{\partial}{\partial f(\boldsymbol{x})_c}\left[-\log f(\boldsymbol{x})_y\right] = -\frac{1_{(y=c)}}{f(\boldsymbol{x})_y}, \tag{2.19}$$

where $1_{(y=c)}$ is the indicator function, c a given class and y the true class of $\boldsymbol{x}$. The numerator is multiplied by the indicator function because if $y \neq c$, then $-\log f(\boldsymbol{x})_y$ is constant w.r.t. $f(\boldsymbol{x})_c$. Hence its gradient is given by:

$$\nabla_{\boldsymbol{f}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right] = \frac{-\boldsymbol{e}(y)}{f(\boldsymbol{x})_y}, \tag{2.20}$$

where $\boldsymbol{e}(y)$ is the one-hot vector of y which is everywhere 0 except the position where $y = c$.

Now, we will derive the gradient of the loss w.r.t. the preactivations of the output layer. Its expression is given by:

$$\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}[-\log f(\boldsymbol{x})_y] = -(1_{(y=c)} - f(\boldsymbol{x})_c), \tag{2.21}$$

which is simple and elegant, although it is derived after several intermediate steps which we illustrate next:

$$\begin{aligned}
&\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}[-\log f(\boldsymbol{x})_y] = \frac{-1}{f(\boldsymbol{x})_y}\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c} f(\boldsymbol{x})_y \\
&= \frac{-1}{f(\boldsymbol{x})_y}\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}\text{softmax}(\boldsymbol{a}^{(L+1)}(\boldsymbol{x}))_y = \frac{-1}{f(\boldsymbol{x})_y}\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}\frac{\exp(a^{(L+1)}(\boldsymbol{x})_y)}{\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})} \\
&= \frac{-1}{f(\boldsymbol{x})_y}\left(\frac{\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}\exp(a^{(L+1)}(\boldsymbol{x})_y)}{\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})}\right. \\
&\left. - \frac{\exp(a^{(L+1)}(\boldsymbol{x})_y)\left(\frac{\partial}{\partial a^{(L+1)}(\boldsymbol{x})_c}\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})\right)}{\left(\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})\right)^2}\right) \\
&= \frac{-1}{f(\boldsymbol{x})_y}\left(\frac{1_{(y=c)}\exp(a^{(L+1)}(\boldsymbol{x})_y)}{\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})} - \frac{\exp(a^{(L+1)}(\boldsymbol{x})_y)}{\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})}\frac{\exp(a^{(L+1)}(\boldsymbol{x})_c)}{\sum_{c'}\exp(a^{(L+1)}(\boldsymbol{x})_{c'})}\right) \\
&= \frac{-1}{f(\boldsymbol{x})_y}\left(1_{(y=c)}\text{softmax}(\boldsymbol{a}^{(L+1)})_y - \text{softmax}(\boldsymbol{a}^{(L+1)})_y\text{softmax}(\boldsymbol{a}^{(L+1)})_c\right) \\
&= \frac{-1}{f(\boldsymbol{x})_y}\left(1_{(y=c)}f(\boldsymbol{x})_y - f(\boldsymbol{x})_y f(\boldsymbol{x})_c\right) = -\left(1_{(y=c)} - f(\boldsymbol{x})_c\right).
\end{aligned}$$

Equivalently, the gradient is:

$$\nabla_{\boldsymbol{a}^{(L+1)}(\boldsymbol{x})}[-\log f(\boldsymbol{x})_y] = -(\boldsymbol{e}(y) - \boldsymbol{f}(\boldsymbol{x})). \tag{2.22}$$

We continue with the derivation of the partial derivative of the hidden units. We could follow the same procedure as before and obtain an analytic expression for each neuron of each hidden layer. In that case, things are getting complicated as we have to derive multiple expressions for each neuron and as we are moving to the lower layers of the network the expressions are getting more complicated. Thus, we need a more general formulation. The chain rule of calculus is used for this reason, which obtains the gradients of each layer in a highly efficient way.

Let a be a real number, and let p, q: $\mathbb{R} \to \mathbb{R}$. Suppose that $b = q(a)$ and $c = p(q(a)) = p(b)$. Then the chain rule states that:

$$\frac{dc}{da} = \frac{dc}{db}\frac{db}{da}. \tag{2.23}$$

We can generalize this beyond the scalar case. Suppose that $\boldsymbol{a} \in \mathbb{R}^m$, $\boldsymbol{b} \in \mathbb{R}^n$, q: $\mathbb{R}^m \to \mathbb{R}^n$ and p: $\mathbb{R}^n \to \mathbb{R}$. If $\boldsymbol{b} = q(\boldsymbol{a})$ and $c = p(\boldsymbol{b})$, then:

$$\frac{\partial c}{\partial a_j} = \sum_i \frac{\partial c}{\partial b_i}\frac{\partial b_i}{\partial a_j}. \tag{2.24}$$

In our setting, let a_j to be a hidden unit, b_i the preactivation in the layer above and c our loss function. Then the partial derivative of j-th hidden unit in layer k w.r.t. the loss is derived as:

$$\begin{aligned}\frac{\partial}{\partial h^{(k)}(\boldsymbol{x})_j}\left[-\log f(\boldsymbol{x})_y\right] &= \sum_i \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k+1)}(\boldsymbol{x})_i}\frac{\partial a^{(k+1)}(\boldsymbol{x})_i}{\partial h^{(k)}(\boldsymbol{x})_j} \\ &= \sum_i \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k+1)}(\boldsymbol{x})_i} W_{i,j}^{(k+1)} \qquad (2.25) \\ &= \left(\boldsymbol{W}_{\cdot,j}^{\top}\right)\left(\nabla_{\boldsymbol{a}^{(k+1)}(\boldsymbol{x})_i}\left[-\log f(\boldsymbol{x})_y\right]\right), \qquad (2.26)\end{aligned}$$

where $\boldsymbol{W}_{\cdot,j}$ is the j-th column of $\boldsymbol{W}$. Equation (2.25) follows from $a^k(\boldsymbol{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\boldsymbol{x})_j$.
The gradient is given by:

$$\nabla_{\boldsymbol{h}^{(k)}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right] = \boldsymbol{W}^{(k+1)^{\top}}(\nabla_{\boldsymbol{a}^{(k+1)}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right]). \tag{2.27}$$

In equation (2.27), we've seen how to acquire the partial derivative of the loss w.r.t. the hidden units of a given layer, where we used the chain-rule of calculus by summing over all hidden units in the layer above. Now, we will use the chain-rule to obtain the partial derivatives of the loss w.r.t. a hidden unit's preactivation. The difference is that now we will not sum over several different units, as the preactivation of a hidden unit depends only on its activation. The steps to obtain it are the following:

$$\frac{\partial}{\partial a^{(k)}(\boldsymbol{x})_j}\left[-\log f(\boldsymbol{x})_y\right] = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial h^{(k)}(\boldsymbol{x})_j}\frac{\partial h^{(k)}(\boldsymbol{x})_j}{\partial a^{(k)}(\boldsymbol{x})_j} = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial h^{(k)}(\boldsymbol{x})_j} g'(a^{(k)}(\boldsymbol{x})_j), \tag{2.28}$$

where the last equality is true because $h^{(k)}(\boldsymbol{x})_j = g(a^{(k)}(\boldsymbol{x})_j)$. Similarly, the gradient is:

$$\begin{aligned}&\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right] \\ &= \nabla_{\boldsymbol{h}^{(k)}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right] \odot [..., g'(a^{(k)}(\boldsymbol{x})_j), ...] \qquad (2.29) \\ &= \left(\nabla_{\boldsymbol{h}^{(k)}(\boldsymbol{x})}\left[-\log f(\boldsymbol{x})_y\right]\right)^{\top} \nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})}\boldsymbol{h}^{(k)}(\boldsymbol{x}), \qquad (2.30)\end{aligned}$$

where $\odot$ is the element-wise product. Equations (2.29) and (2.30) are equivalent since $\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})}\boldsymbol{x}^{(k)}(\boldsymbol{x})$ in equation (2.30) is the Jacobian matrix of the activations with respect to the preactivations which is a diagonal matrix.

The only missing element in equation (2.28) is the definition of the activation functions' gradients. Below you can see the gradients of the activation functions we defined in Section 2.3.

Activation functions gradients

- Linear: $g'(a) = 1$
- Tanh: $g'(a) = 1 - g^2(a)$
- Sigmoid: $g'(a) = g(a)(1 - g(a))$
- ReLU: $g'(a) = \begin{cases} 0 & x < 0, \\ 1 & x > 0 \end{cases}$

In equation (2.29), we can see that we have an element-wise multiplication with the derivative of the activation function. If we observe the derivative of the sigmoid and tanh activation functions, we can see that if their preactivations have very large positive or negative values, the values of the gradients are pushed towards zero. Thus, the whole expression of the gradient of the loss function w.r.t. the preactivations in equation (2.30) have values near zero. As we will see below, this gradient is involved in the gradient of the loss function w.r.t. the weights. This will result in driving the weight gradients toward zero. Thus, when we will perform updates to learn the value of the parameters, the values of the weights will not change, which means that we do not learn any useful information. This is actually the vanishing gradient problem and this is why sigmoidal units are saturating units.

Finally, we will obtain expressions for the gradient of the loss w.r.t. the model weights and biases at each layer, which we will use for performing updates with an optimization algorithm in order to learn the set of our parameters. We start with the partial derivative of the loss w.r.t. the weights that is:

$$\frac{\partial}{\partial W_{i,j}^{(k)}}[-\log f(\boldsymbol{x})_y] = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k)}(\boldsymbol{x})_i}\frac{\partial a^{(k)}(\boldsymbol{x})_i}{\partial W_{i,j}^{(k)}} = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k)}(\boldsymbol{x})_i} h^{(k-1)}(\boldsymbol{x})_j. \tag{2.31}$$

The last part of the expression is true as $a^k(\boldsymbol{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\boldsymbol{x})_j$. The gradient is:

$$\nabla_{\boldsymbol{W}^{(k)}} \left[-\log f(\boldsymbol{x})_y\right] = \left(\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right]\right) \boldsymbol{h}^{(k-1)}(\boldsymbol{x})^\top, \tag{2.32}$$

which is a matrix of size (#hidden units in k-th layer) $\times$ (#inputs in $(k-1)$-th layer).

Finally, lets derive the partial derivative of the loss w.r.t. the biases of a hidden layer, that is:

$$\frac{\partial}{\partial b_i^{(k)}} \left[-\log f(\boldsymbol{x})_y\right] = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k)}(\boldsymbol{x})_i} \frac{\partial a^{(k)}(\boldsymbol{x})_i}{\partial b_i^{(k)}} = \frac{\partial[-\log f(\boldsymbol{x})_y]}{\partial a^{(k)}(\boldsymbol{x})_i}. \tag{2.33}$$

Thus, the gradient is:

$$\nabla_{\boldsymbol{b}^{(k)}} \left[-\log f(\boldsymbol{x})_y\right] = \nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right]. \tag{2.34}$$

Now, we have all the necessary gradients in order to describe the back-propagation algorithm. You may noticed in the expressions for the gradients in the hidden layers that the left part of each expression depends on the gradients of another part of the neural network. Specifically, in equation (2.27) the gradient of the activation depends on the gradient of the preactivation of the layer above, similarly the gradient of a layer's weights and biases depends on the the gradient of the layer's preactivation, and the gradient of a layer's preactivation depends on the gradient of the layer's activation. This suggests that putting these expressions in specific order we can compute these gradients using at each step precomputed gradients of previous steps that we don't need to evaluate again. This is actually the main idea of back-propagation algorithm which is described in algorithm 2.1. In a different problem (e.g. regression), the only part that changes is the computation of the gradient of the output preactivation.

The computations for performing forward propagation and back-propagation can be described using the notion of a *computational graph*. A computational graph is an acyclic graph. Each node in the graph indicates a variable, where the variable may be a scalar, vector, matrix, tensor, or even a variable of another type. To formalize our graphs, we also need to introduce the idea of an *operation*. An operation is a simple function of one or more variables. Our graph is accompanied by a set of allowable operations. Functions more complicated than the operations in this set may be described by composing many operations together. If a variable y is computed by applying an operation to a variable x, then we draw a directed edge from x to y.

A computational graph for the case of a simple feedforward neural net with one hidden layer is depicted in figure 2.3. As you can see, the computational graph is

Algorithm 2.1 Back-propagation algorithm

```
Require: A forward propagation, to obtain $\boldsymbol{f}(\boldsymbol{x})$, $\boldsymbol{h}^{(k)}(\boldsymbol{x})$, $\forall k \in \{1, ..., L+1\}$
 1: ▷ Compute the gradient of the output preactivation
 2: $\nabla_{\boldsymbol{a}^{(L+1)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y] \Longleftarrow -(\boldsymbol{e}(y) - \boldsymbol{f}(\boldsymbol{x}))$
 3: for $k$ from $L+1$ to 1 do
 4:     ▷ Compute the gradients of hidden layer parameters (weights and biases)
 5:     $\nabla_{\boldsymbol{W}^{(k)}} [-\log f(\boldsymbol{x})_y] \Longleftarrow \left(\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y]\right) \boldsymbol{h}^{(k-1)}(\boldsymbol{x})^\top$
 6:     $\nabla_{\boldsymbol{b}^{(k)}} [-\log f(\boldsymbol{x})_y] \Longleftarrow \nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y]$
 7:     ▷ Compute the gradients of the hidden layer below
 8:     $\nabla_{\boldsymbol{h}^{(k-1)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y] \Longleftarrow \boldsymbol{W}^{(k)\top} (\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y)]$
 9:     ▷ Compute the gradients of preactivation of the hidden layer below
10:     $\nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y] \Longleftarrow \left(\nabla_{\boldsymbol{h}^{(k-1)}(\boldsymbol{x})} [-\log f(\boldsymbol{x})_y]\right)^\top \nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} \boldsymbol{h}^{(k-1)}(\boldsymbol{x})$
11: end for
```

composed of different variables/nodes. We have the input nodes $\boldsymbol{x}$ and y, which provide the input data and the labels to the network, respectively. Next, we have the preactivation nodes $\boldsymbol{a}^{(k)}(\boldsymbol{x})$, where given its children, namely the weight nodes and biases nodes $\boldsymbol{W}^{(k)}$ and $\boldsymbol{b}^{(k)}$, respectively, as well as the node below it performs the operation of inner product. The activation nodes $\boldsymbol{h}^{(k)}(\boldsymbol{x})$ perform a nonlinear operation, i.e. they compute the layer's nonlinearity given the practivation nodes $\boldsymbol{a}^{k}(\boldsymbol{x})$. The output node $\boldsymbol{f}(\boldsymbol{x})$ computes the network's prediction and finally the loss node $L(\boldsymbol{f}(\boldsymbol{x}), y)$ computes the loss given the nodes $\boldsymbol{f}(\boldsymbol{x})$ and y. The nodes $\boldsymbol{x}$, y, $\boldsymbol{W}^{(k)}$, and $\boldsymbol{b}^{(k)}$ are not accompanied with any operations, they just store numerical values for the input variables.

Of course, more complicated computational graphs can be constructed by adding more elements to the network. If for example, we add regularization terms there will be extra nodes that represent these terms as well as edges connecting these nodes with other parts of the network that the regularization term affects. We avoid the illustration of such graphs for simplicity.

Computational graphs similarly to symbolic mathematical expressions, operate on *symbols*, namely variables that do not have a specific value. These represenatations are called symbolic representations and they compute symbolic expressions. Each node of the graph is a symbol, which does not have a numerical value. Symbols together with operations of each node, define the symbolic expression that a graph represents.

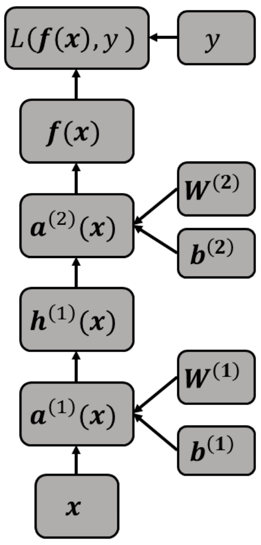

$L(\boldsymbol{f}(\boldsymbol{x}), y)$
y
$\boldsymbol{f}(\boldsymbol{x})$
$\boldsymbol{W}^{(2)}$
$\boldsymbol{a}^{(2)}(\boldsymbol{x})$
$\boldsymbol{b}^{(2)}$
$\boldsymbol{h}^{(1)}(\boldsymbol{x})$
$\boldsymbol{W}^{(1)}$
$\boldsymbol{a}^{(1)}(\boldsymbol{x})$
$\boldsymbol{b}^{(1)}$
$\boldsymbol{x}$

Figure 2.3: Computational graph of a feedforward neural network with one hidden layer.

Providing an input with numerical values to the graph, the symbols are evaluated through the operations in the nodes and we obtain numerical values for the variables of the graph.

Forward propagation is simply the evaluation of all the symbols in the computational graph that defines our neural network. We traverse the graph from the input to the loss node and at each step we compute the operation that a node performs given its children, denoted as **fprop** operation.

Back-propagation can also be computed using a computational graph. This can be accomplished by adding nodes to the graph, that provide symbolic description of the desired derivatives. For simplicity of illustration, we show an example of a simple computational graph for performing back-propagation in figure 2.4. We avoid showing the graph for the gradients of a neural network as it gets very complicated, and hence it would be tedious for someone to read. At the left part of the figure, you can see the graph that corresponds to the forward propagation. The computational graph for the back-propagation is at the right part. Each node computes its derivative given its parent. Then, multiple derivative nodes are combined together through a product operation and form the chain-rule of calculus. The back-propagation algorithm applies the chain-rule multiple times to obtain expressions for the derivatives given pre-computed derivatives. This graph formulates exactly this procedure. Any subset of the graph may then be evaluated using specific numerical values at a later time.

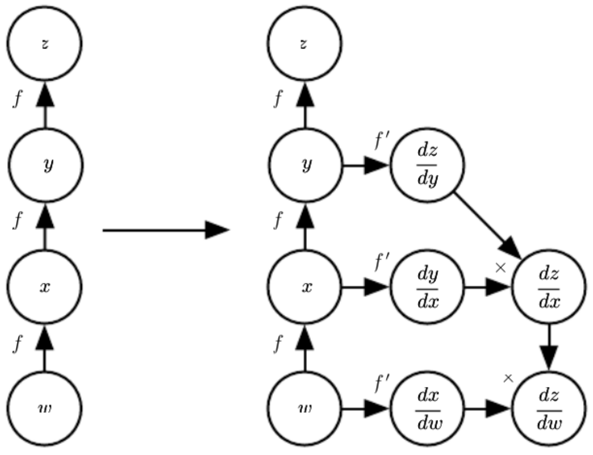


Figure 2.4: Computational graph for performing back-propagation with additional nodes that represent the gradients. Left: The computational graph for the forward propagation. Right: The computational graph with additional nodes for performing back-propagation. The nodes have a bprop operation where they compute the gradients using the chain rule. Figure reproduced from [1].

This allows us to avoid specifying exactly when each operation should be computed. Instead, a generic graph evaluation engine can evaluate every node as soon as its parents' values are available.

Now, each node has also a **bprop** operation. We traverse the computational graph in reverse order starting from the node z. At each step, we can compute the gradient of node z w.r.t. each parent of the loss node by recursively applying the chain-rule. We continue this procedure for each node until we reach the input node $\boldsymbol{x}$. Thus, the bprop operation related with each node is responsible for computing the gradient-jacobian product which defines the chain-rule for the gradient of each node. This is how the back-propagation algorithm is able to achieve great generality. Each operation is responsible for knowing how to back-propagate through the edges in the graph that it participates in. This is called *automatic differentiation*.

Above, we constrained the description of computational graphs and the back-propagation algorithm for feedforward neural networks. Modern deep learning libraries implement a general form of back-propagation algorithm, that is based on the idea of constructing a computational graph for both forward propagation and back-propagation. This general form of back-propagation, makes it suitable for training several different types of networks, such as convolutional networks or recurrent neural networks. Here, we wanted only to give the general idea of how back-propagation works by building a computational graph for constructing symbolic representations of its derivatives, but we will not go in further implementation details as is out of the scope of this work.

2.7 Optimization

In Section 2.5, we saw how learning is casted as an optimization problem. The cost function that is used for optimization is the empirical risk (2.18). Now, the goal is to search for the best set of parameters that minimize our cost function $J(\boldsymbol{\theta})$, such that:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, J(\boldsymbol{\theta}). \tag{2.35}$$

Our cost function, or objective function may be decomposed as a sum over the training examples. As we will see, we can leverage this special form of objective function and design optimization algorithms scalable to large datasets.

2.7.1 Optimization methods

Gradient descent

Gradient descent algorithm is the basic algorithm in which several algorithms for training neural networks are based on. Gradient descent presents the main idea of how to perform parameter updates as an iterative optimization procedure. Typically, numerical optimization methods are iterative, where at each iteration they move the parameter vector in a descent direction, i.e. a direction where the objective function decreases. Gradient descent proposes to move in the direction in which our objective function decreases faster. To explain which is this direction, we first need to explain the notion of a **directional derivative**.

The directional derivative in direction $\boldsymbol{u}$ (a unit vector) is the slope of a function f in direction $\boldsymbol{u}$. In other words, the directional derivative is the derivative of the function $f(\boldsymbol{x} + \alpha\boldsymbol{u})$ w.r.t. α, evaluated at $\alpha = 0$. Using the chain-rule, we can see that $\frac{\partial}{\partial\alpha} f(\boldsymbol{x} + \alpha\boldsymbol{u})$ evaluates to $\boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x})$ when $\alpha = 0$.

Hence, the direction in which f decreases the fastest is given by:

$$\min_{\boldsymbol{u},\boldsymbol{u}^\top\boldsymbol{u}=1} \boldsymbol{u}^\top \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = \min_{\boldsymbol{u},\boldsymbol{u}^\top\boldsymbol{u}=1} \|\boldsymbol{u}\|_2 \|\nabla_{\boldsymbol{x}} f(\boldsymbol{x})\|_2 \cos\theta, \tag{2.36}$$

where θ is the angle between $\boldsymbol{u}$ and the gradient. Substituting in $\|\boldsymbol{u}\|_2 = 1$ and ignoring factors that do not depend on $\boldsymbol{u}$, this simplifies to $\min_u \cos\theta$. This is minimized when $\theta = 180°$, that is when $\boldsymbol{u}$ points in the opposite direction to the gradient. This suggests that we can decrease f by moving in the direction of the negative gradient. This is known as the method of steepest descent or gradient descent.

Gradient descent proposes a new point:

$$\boldsymbol{x}' = \boldsymbol{x} - \alpha \nabla_{\boldsymbol{x}} f(\boldsymbol{x}), \tag{2.37}$$

where α is the *step size* or *learning rate* in the machine learning literature and it is a positive scalar. The learning rate can be defined as a small constant or it can change in each iteration using several strategies such as *line search* methods. Gradient descent converges when the gradient of our objective function is zero.

The gradient descent algorithm is known in the machine learning literature as *batch gradient descent*. In the case of training a neural network, where the gradient of the empirical risk is given by:

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}), \tag{2.38}$$

batch gradient descent perform updates using the following formula:

$$\begin{align} \boldsymbol{\theta}^{(t+1)} &= \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] \tag{2.39} \\ &= \boldsymbol{\theta}^{(t)} - \alpha \frac{1}{m} \sum_{i=1}^{m} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)}), \tag{2.40} \end{align}$$

where t denotes the iteration. Batch gradient descend converges towards a local minimum provided sufficiently small learning rates α.

Each iteration of batch gradient descent requires the computation of $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)]$, which is an average over the gradients of the entire training set. The computational cost of this computation is $O(m)$, where m is the size of our training set. For large training sets, the time to take a single gradient step becomes prohibitively long. Besides that, batch gradient descent becomes intractable for training sets that do not fit in memory.

Stochastic gradient descent

Stochastic gradient descent (SGD) is a drastic simplification of batch gradient descent. In (2.38), $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)]$ is an expectation of gradients over the entire training dataset. Instead of computing this expectation, each iteration of stochastic gradient descent approximates $\nabla_{\boldsymbol{\theta}} \mathbb{E}_{\hat{p}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}),y)]$ by choosing an example $(\boldsymbol{x}^{(t)}, y^{(t)})$ at random, and updating the parameters $\boldsymbol{\theta}^{(t)}$ by using the following formula:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(t)};\boldsymbol{\theta}), y^{(t)}). \tag{2.41}$$

Thus, it performs an update by estimating the average gradient of the loss over the entire training set with the gradient of the loss of a single randomly chosen training example. Averaging the stochastic gradient descent update rule over all possible choices of training examples $(\boldsymbol{x}^{(t)}, y^{(t)})$ results in the batch gradient descent algorithm. The stochastic gradient descent simplification relies on the hope that the random noise introduced by this procedure will not perturbate the average behavior of the algorithm.

Stochastic gradient descent does not need to remember the examples that where visited in previous iterations. Hence, it is capable of processing examples in an online fashion, where examples are drawn from a stream of continuously created examples. In such situation, stochastic gradient descent directly minimizes the expected risk, as examples are drawn randomly from the true data generation distribution $p(\boldsymbol{x}, y)$.

For computing an unbiased estimate of the expected gradient, the examples should be independent and subsequent examples should not be correlated with each other. This suggests that at each iteration of SGD, examples should be sampled randomly. In large training sets, it is impractical to sample examples uniformly at random each time we perform an update. A practical solution to that is to shuffle the order of the dataset on each *epoch*. An epoch is defined as a pass of each example in the dataset once. Picking examples without shuffling the dataset might result in decreased performance of SGD as examples might come in particular order or grouped by class. Thus, shuffling is crucial for the performance of SGD.

Another motivation of using an estimation of the gradient instead of using the average gradient over the entire training set is that the training set might contain redundant information. This means that a big number of training examples might make very similar contributions to the gradient, and hence batch gradient descent would do unnecessary gradient computations. SGD eliminates this potential redundancy by performing one update at a time.

The convergence of stochastic gradient descent has been studied extensively in the stochastic approximation literature. Convergence results usually require learning rates satisfying the conditions:

$$\sum_t \alpha_t \to \infty, \tag{2.42}$$

and

$$\sum_t \alpha_t^2 < \infty. \tag{2.43}$$

These conditions propose that learning rates should decrease over the iterations of SGD. Thus, from now on we denote the learning rate on iteration t as α_t instead of using a constant learning rate α.

The convergence speed of stochastic gradient descent is in fact limited by the noisy approximation of the true gradient. This noisy approximation introduces variance in our parameter estimates $\boldsymbol{\theta}^{(t)}$. If the learning rate is decreased slowly, then the variance decreases slowly while with a fast decrease of the learning rate the parameter estimates need more time to reach a local optimum. Thus, careful decreasing strategies are needed. We will examine some popular learning rate decreasing strategies later in this section. Also, very important is the initial learning rate which is usually selected with hyperparameter optimization.

Finally, a very important property of SGD is that the computational cost per update does not increase with the sise of the training set. This makes feasible the application of SGD in large datasets and allows convergence at a reasonable amount of time.

Mini-batch stochastic gradient descent

A very similar algorithm to stochastic gradient descent is the mini-batch stochastic gradient descent algorithm, which performs updates using the formula:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \frac{1}{b} \sum_{i=1}^{b} \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), y^{(i)}). \tag{2.44}$$

Here, an estimation of the gradient is computed on average over b training examples where $b << m$. As b grows, equation (2.44) approximates better the batch gradient descent update. At the same time, the variance of our parameters estimation is reduced; hence, we can reach faster a local optimum. Moreover, when b increases we can get more multiply-add operations per second by leveraging highly optimized matrix-matrix multiplications which makes the computation of the gradient in batches very efficient. Typical numbers for b range from 50 to several hundreds. Using mini-batches of examples for optimization has became the default choice for training neural networks as it has more stable convergence from SGD and at the same time retains the nice scalability properties of SGD.

Momentum

The surface of several objective functions has the form of a ravine, i.e. areas which are shallow and long in the direction towards the optimum and steep on the other direction. Alternatively, you can see these areas as a ”quadratic bowl”. In such areas, SGD will tend to oscillate across the steep direction as the negative gradient will point to the steepest direction rather that the shallow direction across the optimum. Thus, in such cases SGD oscillates across the ravine or performs a ”zig-zag” effect without making much progress to the local optimum.

The objective functions of deep networks have this form near local optima, and thus standard SGD can lead to very slow convergence. Momentum [21] is a method for accelerating in the direction towards the minimum and dampen oscillations towards the steep direction.

The intuition of momentum is derived from a physical interpretation. Imagine that we have a ball on the surface of our objective function. The location of the ball on the horizontal plane represents our parameter vector while the location of the ball on the vertical plane is our objective function. The ball has initial velocity $\boldsymbol{v} = 0$ and starts moving down the hill. The movement here represents the updates of our parameters. Initially, the ball will follow the direction of steepest descent as it doesn't have initial velocity. As long as it will start gaining velocity, it will not keep following the same direction with the gradient. That is because its momentum moves it towards the previous direction. Furthermore, as we are moving towards the local minimum we need to loose some energy so that our parameter vector converges. Thus, we need to introduce some sort of viscosity, that is the velocity of our parameter vector diminishes on each update.

Momentum algorithm accumulates an exponentially decaying moving average of past gradients and continue by moving in their direction. Momentum algorithm is

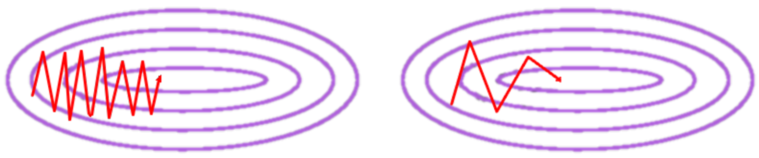


Figure 2.5: Left: Standard SGD. It is clear that SGD oscillates highly across the ravines. Right: SGD with momentum. The oscillations across the steep direction are dampened and momentum moves faster towards the relevant direction. Figure reproduced from [2]

combined with mini-batch stochastic gradient descent. We denote by $\nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta})$ the gradient of the loss over a mini-batch of b training samples. An update of the mini-batch stochastic gradient descent with momentum algorithm is given by the formula:

$$\boldsymbol{v}^{(t)} = \gamma \boldsymbol{v}^{(t-1)} - \alpha \nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta}^{(t)}) \tag{2.45}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \boldsymbol{v}^{(t)}, \tag{2.46}$$

where $\boldsymbol{v}^{(t)}$ is the velocity of our parameter parameter vector at time t, α is as before the learning rate and γ is the momentum term. Momentum is a misnomer as it refers actually to viscosity. Momentum term regulates how fast we are moving towards the optimum and is set $\gamma < 1$. It exponentially decays the velocity of our parameter vector over updates as we are moving towards a local optimum.

Now we are not moving to the direction of the gradient but to the direction of our velocity vector. The gradient increments the velocity of previous steps. The velocity increases for dimensions whose gradients point in the same direction and decreases for dimensions whose gradients change direction. Thus, it dampens oscillations in directions of high curvature by canceling accumulated gradients with opposite signs and it accelerates in directions with small but consistent gradients that are in the shallow direction of the ravine and point toward the local optimum. Hence, the large gradients across the ravine are cancel out while the small gradients along the ravine accumulate velocity towards that direction as they are pointing on the same direction.

In figure 2.5, there is a graphical illustration of how SGD updates differ from SGD with momentum updates. You can see that SGD with momentum reduces oscillations comparing to standard SGD which leads in faster training.

If momentum algorithm always observes the same gradient $\nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta})$, it will follow the direction of $-\nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta})$ until it finally reaches its terminal velocity, that is the velocity at ∞ which is given by:

$$\boldsymbol{v}(\infty) = \frac{1}{1-\gamma}\left(-\alpha \nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta})\right). \tag{2.47}$$

The term $\frac{1}{1-\gamma}$ suggests that if γ is close to 1 momentum becomes much faster than SGD. For example setting $\gamma = 0.99$ corresponds to going 100 faster than standard SGD. Common values for γ are 0.5, 0.9 and 0.99. Furthermore, γ can be adapted through time. A typical momentum annealing procedure is to start with $\gamma = 0.5$ for a few updates and then anneal it smoothly over 0.9 or 0.99 or so. Nevertheless, it turns out that adapting the momentum term over time is less important than adapting the learning rate, and thus constant momentum terms are very common in practice.

Nesterov momentum

The standard momentum method first computes the gradient at the current location and then performs a big jump in the direction of the updated accumulated gradient.

In [22], a variant of the momentum algorithm that was inspired by Nesterov's accelerated gradient method [23, 24] was introduced. The formula of the updates of Nesterov momentum is very similar to momentum method and is given by:

$$\boldsymbol{v}^{(t)} = \gamma \boldsymbol{v}^{(t-1)} - \alpha \nabla_{\boldsymbol{\theta}} J_b(\boldsymbol{\theta}^{(t)} - \gamma \boldsymbol{v}^{(t-1)}) \tag{2.48}$$

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + \boldsymbol{v}^{(t)}. \tag{2.49}$$

The difference with standard momentum method is that now we do not compute the gradient at position $\boldsymbol{\theta}$ of our parameter vector, but we first move our parameter vector to the position $\boldsymbol{\theta}^{(t)} - \gamma \boldsymbol{v}^{(t-1)}$ and then compute the gradient. Therefore, we first make a big jump in the direction of previously accumulated gradients and in that position we measure the gradient and make a correction. In the standard momentum method the current gradient is added to the previously accumulated gradients and then the parameter vector moves to the direction of previsously accumulated gradients. In Nesterov momentum, the previously accumulated gradients are used to perform first a jump to that direction, measure the gradient and perform a correction based on that position.

The quantity $\boldsymbol{\theta}^{(t)} - \gamma \boldsymbol{v}^{(t-1)}$ can be seen as an approximation of the position of our parameters in the direction of the updated accumulated velocity. Therefore, we first make an initial estimate of where our parameter vector is going to be and then we

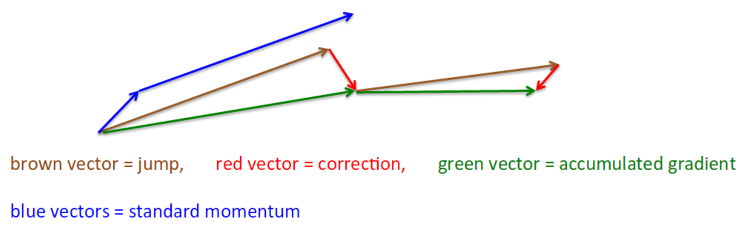

brown vector = jump,
red vector = correction,
green vector = accumulated gradient
blue vectors = standard momentum

Figure 2.6: The difference between a standard momentum step and a Nesterov momentum step. As you can see the blue vectors correspond to a step of standard momentum method and the others correspond to a step of Nesterov momentum. Nesterov momentum takes bigger steps, and hence it moves faster towards the local minimum. Figure reproduced from [3]

compute the gradient there, where we combine this gradient with our initial estimate and perform a jump to that direction.

In figure 2.6, the difference between standard momentum and Nesterov momentum is clearly depicted. The blue vectors correspond to a jump of the standard momentum method, where the small vector is the direction of the current gradient and the big vector is the jump of our parameter vector in the direction of updated accumulated gradients. In Nesterov momentum on the other hand, first we perform a jump in the direction of previously accumulated gradients which is the brown vector, then we measure the gradient in that position which direction is denoted by the red vector, and then we compute the updated accumulated gradients, by adding the gradient in that position to the previously accumulated gradients, which results in the green vector and finally we perform the final step of Nesterov momentum in the direction of the updated accumulated gradients which result in the new green vector.

Nesterov momentum with mini-batch stochastic gradient descent has become a default option for training deep neural nets.

There are also other very famous optimization methods in the deep learning literature, namely Adagrad [25], Adadelta [26], RMSprop (unpublished) and Adam [27] which automatically adapt the learning rate on each update and they use a separate learning rate for each parameter dimension. They have shown very good performance in training deep networks and they converge faster than momentum or nesterov momentum in general. We will not perform an analysis of these algorithms in this work. Lastly, we do not study second-order methods as they require the computation of the inverse of the Hessian matrix which becomes intractable for large models with large vectors of parameters.

2.7.2 Learning rate schedules

As we described before, it is required that equations (2.42) and (2.43) are satisfied such that SGD converges. Hence, learning rate schedules are necessary for decreasing the learning rate over training. The most straightforward option is to keep a constant learning rate which generally works well in several cases. Thus, even without satisfied convergence conditions, our model might be able to learn a task well as the goal of a machine learning algorithm is not to optimize perfectly the error in the training set but to be able to perform well on new data. Nevertheless, a lot of times decreasing the learning rate results in increased performance because the oscillations are reduced towards approaching the local optimum. Below, we present several different heuristics on decaying the learning rate over training epochs.

Two similar learning rate schedules are the following:

$$\alpha_t = \frac{\alpha_0}{1 + \delta t}, \tag{2.50}$$

and

$$\alpha_t = \frac{\alpha_0}{t^\delta}, \tag{2.51}$$

where α_0 is the initial learning rate, t is the current iteration, and δ is a decay constant that controls how rapidly the learning rate decays. They belong to the family of $O(1/t)$ learning rate schedules. A typical setting for $O(1/t)$ learning rate schedules is to maintain the learning rate constant for the first few updates and then start decreasing it using a given schedule. The intuition is that we allow the optimizer to perform big steps at the first iterations such that it reaches faster near the local optimum, and then we are decreasing the learning rate so that our updates perform small steps in order to converge to the local optimum. A heuristic for adaptively setting the iteration at which we start decreasing the learning rate can be, to monitor the training error over consecutive training epochs and start decreasing the learning rate when the error stops improving significantly.

One popular option of learning rate schedules, that works fairly well in practice is **step decay**, that is to decrease the learning rate every a fixed number of epochs by a constant, e.g. to decrease the learning rate by a factor of 0.5 every 20 epochs.

The advantage of this approach is that its hyperparameters, i.e. the fraction of decay and the time steps in unit of epochs are more naturally interpretable than δ in equations (2.50), (2.51), that might be the case that we may have a better notion of how to tune them and which values might work well for a given problem.

The schedules described above have the disadvantage that they introduce new hyperparameters that need to be tuned. Methods that are hyperparameter-free are preferable as tuning the hyperparameters of a model might be a tedious task especially when the number of hyperparameters is getting bigger. One hyperparameter-free learning rate schedule, that is actually my favorite choice, is to decrease the learning rate when validation error ceases. Thus, in that case we hold a validation set and measure its performance after each epoch. When the error function stops improving we decay our learning rate by a constant which is usually set to 0.5. We can perform this process several times by going back on the last best performing epoch and successively decrease the learning rate until the validation error starts improving. In that case, a maximum allowed number of times must be set of going back and refining the learning rate. Moreover, this is a natural option of learning rate schedule as we wish our model to perform well on the validation set and not on the training set; thus, it is natural to decay the learning rate based on the performance on the validation set.

2.7.3 Parameter initialization

Deep learning training algorithms have strong dependence on the initial values of the parameters of the model. The initialization of the parameters can determine whether our learning algorithm will converge. If convergence is ensured, the initialization might affect seriously the convergence speed of the algorithm. Furthermore, the initial point can determine whether we converge in an area with high or low cost, or even in case of the same cost, the initial point can even affect generalization performance.

Modern initialization strategies are simple and heuristic. The design of advanced initialization strategies is a very difficult task, as neural network optimization is not yet fully understood. Another difficulty arises from the fact that an initial point might be beneficial from an optimization point of view, but might be disadvantageous for the generalization performance. Our knowledge of how the initial point affects generalization is few to none, and thus we have no insight of how to choose our initial parameters concerning generalization.

A fairly well known property is that the initial parameters need to break the symmetry between hidden units of the same layer. Symmetry means that when the hidden units of a layer share the same input and output parameter vectors, a deterministic algorithm with a deterministic model and a deterministic cost will compute the same outputs in these hidden units and it will update all of them constantly in the same way. In other words, all the hidden units across the layer will compute the same activation functions and will receive the same gradients, and thus it will perform the same updates and will remain identical. It is preferable that each hidden unit computes a different function. By computing each hidden unit a different function, the layer learns more powerful representation that do not contain redundant information. Thus, the forward propagation and back-propagation ”signals” are enhanced. Hence, the need for the hidden units to compute different functions suggests random parameter initialization.

Usually, the biases are set to heuristically chosen constants and we initialize randomly the weights. The weights are almost always sampled from a normal or a uniform distribution. The choice between normal or uniform does not seem to have significant effect on the performance of the algorithm. Nevertheless, the initial scale of the distribution plays a crucial role in the optimization as well as the generalization performance of our model.

Large initial weights will result in a strong symmetry breaking effect, i.e. the hidden units will compute much different activations. This will ensure that the forward propagation and the gradient signals are strong, and thus the information is flowing in the network. On the other hand, strong symmetry breaking might result in overfitting, as the model becomes highly non-linear. Furthermore, a strong symmetry breaking effect results in exploding values in the network, hence exploding gradients. Finally, too large weights may cause several hidden units to saturate especially the sigmoidals, which will block the flow of the forward information as well as the gradients through several hidden units.

Another consideration is the trade-off between initializing our parameters towards better optimization against initializing our parameters towards better regularization. Optimization suggests larger weights so that the gradient flows normally through the network, while regularization prefers smaller weights.

We do not know in prior the true distribution of our parameters. If we perform proper data normalization, we can assume that our parameters are drawn from a Gaussian distribution $\mathcal{N}(0, \sigma)$ with zero mean and standard deviation σ. Similarly, we can assume that the distribution of our parameters is a zero centered uniform distribution $\mathcal{U}[-r, r]$. The range of the distributions is crucial.

A typical setting is to sample in a very small range around zero e.g., to assume a uniform distribution $\mathcal{U}[-0.01, 0.01]$. This option breaks the symmetry between hidden units of the same layer and at the same time has a beneficial effect on regularization; our model is initialized with a set of very small weights close to zero, thus our initial model is not highly-nonlinear due to large weights. Another intuition towards small ranges around zero is that the main activity of our hidden units is around zero. In sigmoid and hyperbolic tangents, the main activity of the neuron is around zero and as moving away the neuron saturates. Thus, it is a fairly reasonable choice to choose ranges around zero in the non-saturating part of the neuron which is also the most informative, as most of the nonlinearity of a neuron resides there. Similarly, for a ReLU hidden unit it is again reasonable to set a range around zero, as in the negative part saturates and in the positive part, even though with a strong signal, it is linear. Thus, a zero center distribution is convenient so that the ReLU initializes to its nonlinear part which will result in diverse hidden units with different behaviors.

One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that we can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in, i.e. its number of inputs. The formula of this commonly used heuristic for initializing our weights is given by:

$$W_{i,j} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right], \tag{2.52}$$

where n is the size of the previous layer. This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence.

In [28], it is suggested to use the normalized initialization:

$$W_{i,j} \sim U\left[-\sqrt{\frac{6}{n+m}}, \sqrt{\frac{6}{n+m}}\right], \tag{2.53}$$

where n,m are the layer's fan-in and fan-out respectively. This latter heuristic is designed to compromise between the goal of initializing all layers to have the same activation variance and the goal of initializing all layers to have the same gradient variance.

It is sufficient to use biases of zero since our weight initialization breaks the symmetry in the hidden units. Nevertheless, sometimes is desirable to ensure that the bias will not saturate the neuron on its initialization, and thus the bias should be moved away from zero towards a non-saturation point. In a ReLU for example, a heuristic that sometimes people use is to initialize the biases to small values e.g. 0.1 such that we ensure that the neuron is active and does not block the gradients from passing through.

One main drawback of equations (2.52) and (2.53) is that the weights become extremely small when the layers become large. This can result in a weak gradient signal which diminishes as it flows through the network. In [29], an alternative initialization scheme is introduced which is called sparse initialization. It proposes that each hidden unit is initialized to have k non zero weights which are initialized using (2.52). The motivation is to keep a constant amount of inputs to each neuron which does not depend on the size of the previous layer n, without making the magnitude of individual weights diminish depending on n. This can also be seen as a symmetry breaking rule.

2.8 Regularization

The goal of a machine learning algorithm is to be able to perform well on previously unseen examples, i.e. on examples that the algorithm didn't observe during training. This is also called the generalization performance of the algorithm. The generalization of a model is measured by the generalization error which is the expected risk we introduced in equation (2.17). As we do not know the true underlying data distribution we cannot measure the true generalization error. To this end, we use two separate sets, the training set and the test set. In the training set, we compute the training error which we minimize to train our model. We then, estimate the generalization error of our model in the test set where we compute the test error. As we train our model in the training set, it is natural that the training error is smaller than the test error. Nevertheless, we wish our model to have low test error as well, in order to be able to perform well on new examples.

Thus, the goal of a machine learning algorithm is twofold. Firstly, the algorithm should be able to minimize the training error of a given model sufficiently. Secondly, the gap between the training error and test error should be small.

More formally, the generalization error decomposes as a sum of two terms, the *bias* term and the *variance* term. The bias term measures how far is the expected model from the true function that maps inputs to targets. The expectation is taken across all possible datasets we can sample from the underlying data distribution. Variance is the deviation from the expected model of all possible models that can be constructed by sampling different datasets from the underlying data distribution. In other words, variance is to what extent do we get very different models with small perturbations in the dataset.

The capacity or complexity of a model measures the model's ability to fit a wide set of functions. High capacity means that our model can fit a bigger set of functions while low capacity means that our model can fit a smaller set of functions.

When the model has high bias we are in an **underfitting** situation, where the capacity of our model is not enough, so that our model can learn the distribution that generates the data. In that case the set of all the possible functions that our model can select is quite small and far from the true solution. Underfitting also occurs when the optimization algorithm is not efficient enough. In both cases, we result in high training error. Thus, underfitting in general is when we are in a situation that we cannot obtain a sufficiently small training error. On the other hand, when the model has high variance we are in an **overfitting** situation, where the capacity of our model is high. In that case, the set of all possible functions that our model can select is quite large. This makes the model being able to learn the distribution that generates the data. However, because the model is too complex it also **memorizes** very specific properties of the dataset, such as the distribution that generates the noise. Thus, with different datasets we obtain quite different models which makes the model not being able to generalize well on new examples. In this case, the training error is quite small as the model learned "very well" properties of the training set, but the the test error is high as our model does not have good generalization performance.

In figure 2.7, we can see examples of several different situations w.r.t. the bias and the variance of a model. The rings represent the sets of all possible functions. The red circle represents the true function that maps inputs to targets while the blue circles represent all the possible functions that our model can select. The vertical

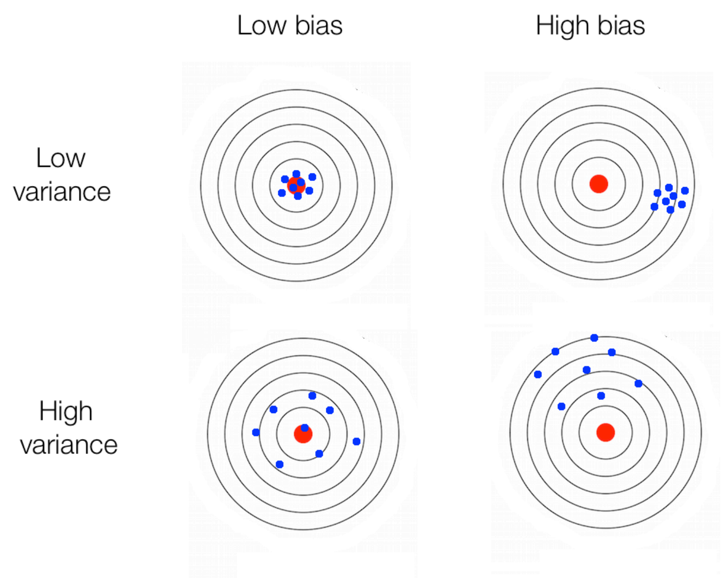


Figure 2.7: Bias and variance. The rings represent the space of all possible functions. The red circle is the true solution in a given task. The blue circles represent the set of functions that our model is able to select. The horizontal axis represent the bias while the vertical axis represent the bias. We can see intuitively what happens in several different situations. Figure reproduced from [4].

axis is the variance while the horizontal axis is the variance. Most interesting and common cases are high bias/low variance and high variance/low bias. In the first case, you can see that our model has low capacity, and thus the range of functions that can approximate is small and quite far from the true solution. We may not get very different solutions as we vary the dataset but the solutions we get are far from the true, thus the error is high. In the second case, the capacity of our model is high; hence, the range of functions that our model is able to select is big. As you can see, among several solutions we can select the true solution because it is in the range of the functions that our model can select but at the same time with small perturbations on the dataset we may obtain very different solutions.

Very complex tasks need models with high capacity. Nevertheless, as we said higher capacity overfits our model. As you can see, there is a trade-off between variance and bias. Thus, we should find ways to control the capacity of a given model, so that there is a good balance of bias and variance and we do not end up in extreme situations.

We can control the capacity of a model by choosing its hypothesis space, that is the family of functions that our model is able to select. For example, the linear regression algorithm has the set of all linear functions of its input as its hypothesis space. We can generalize linear regression to include polynomials rather than just linear functions in its hypothesis space. Doing so, the model's capacity is increased. Another example is that we can control the capacity of a neural network by adding layers or bigger number of hidden units to the neural network. Doing so, we allow our network to be able to fit a much broader set of functions.

Certain functions in a given family may result in high error. Thus, choosing only the hypothesis space is not sufficient. We need to find a way to express preference to certain type of functions from its hypothesis space, while penalizing functions that result in high variance, and hence overfit our model. This process is called *regularization*. Regularization is intended to enforce constraints on a model so that the model does not memorize the dataset.

There has been extensive research on developing regularization techniques for deep learning algorithms, as deep learning models are high capacity models which are very prone to overfitting. Here, we will describe the most common regularization techniques for deep models, which are *parameter norm penalties* and *dropout*. Several other methods that have been incorporated for regularization will not be described here, but the interested reader can look for regularization with ensembles, multi-task learning and semi-supervised learning among others. Lastly, one more technique for regularizing deep models is called *data augmentation* and will be described in Chapter 3.

2.8.1 Parameter norm penalties

Parameter norm penalties approaches limit the capacity of a model by adding penalty terms $\varOmega(\boldsymbol{\theta})$ in the cost function related with the parameter norms of the model. The new cost function is given by:

$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda_{reg}\varOmega(\boldsymbol{\theta}), \tag{2.54}$$

where λ_{reg} is a positive scalar that controls the amount of regularization. $\varOmega(\boldsymbol{\theta})$ controls the magnitude or the number of our model parameters. Different choices for $\varOmega(\boldsymbol{\theta})$ prefer different solutions.

We regularize only the weights of our model as the biases need less data to fit well. That is because the weights define interactions between two variables; hence to fit the weights the algorithm need to observe these two variables under several different circumstances of interactions, while biases control a single variable, and thus less observations are needed to fit well.

The first regularization technique in this family, that is widely used for neural networks, is called L^2 parameter norm penalty. It adds a term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{w}\|_2^2$ in the cost function which now becomes:

$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda_{reg}\frac{1}{2}\|\boldsymbol{w}\|_2^2. \tag{2.55}$$

The above is for the case of a parameter vector. In the case of a neural network, for enforcing constraints in a layer this becomes $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{W}^{(k)}\|_F^2$ where $\|\|_F$ is the Frobenius norm of a matrix and $\boldsymbol{W}^{(k)}$ is the weight matrix of layer k. L^2 penalties drive the weights toward zero and it can be seen as MAP Bayesian inference with a Gaussian prior with zero mean. We could regularize towards a different value than zero, but since we do not know if the weights should be positive or negative, regularization towards zero is a fair choice.

If we make a quadratic approximation to the regularized objective function and an eigen-decomposition to its Hessian matrix, it turns out that the i-th component of the weight vector $\boldsymbol{w}$ is rescaled by the factor $\frac{\lambda_i}{\lambda_i+\lambda_{reg}}$, where λ_i is the i-th eigenvalue of the Hessian matrix. Eigenvalues with $\lambda_i \gg \lambda_{reg}$ will have no regularization effect, while eigenvalues with $\lambda_i \ll \lambda_{reg}$ will shrink w_i to have nearly zero magnitude. A small eigenvalue conveys that a movement in that direction will not significantly increase the gradient. These unimportant directions are decayed. Only directions that contribute significantly to the weight updates remain.

The above analysis is for a general quadratic function. To study the effect of L^2 regularization in machine learning we can study the problem of linear regression where its cost function is truly quadratic. The closed form solution for the weights in a linear regression problem $\boldsymbol{X}\boldsymbol{w} = \boldsymbol{y}$ is $\boldsymbol{w} = \left(\boldsymbol{X}^\top\boldsymbol{X}\right)^{-1}\boldsymbol{X}^\top\boldsymbol{y}$, where $\boldsymbol{X}$ is the dataset matrix and $\boldsymbol{y}$ is the targets vector. With L^2 regularization the solution becomes $\boldsymbol{w} = \left(\boldsymbol{X}^\top\boldsymbol{X} + \lambda_{reg}\boldsymbol{I}\right)^{-1}\boldsymbol{X}^\top\boldsymbol{y}$. Diagonal entries of matrix $\left(\boldsymbol{X}^\top\boldsymbol{X}\right)$ are the variances of the inputs. Thus, L^2 regularization for the problem of linear regression makes the data having higher variance which result in shrinking the weights for data whose covariance with y is lower than their variance.

Another technique of parameter norm penalties is the L^1 regularization. It takes the form $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{w}\|_1 = \sum_i |w_i|$. The regularized cost function now becomes:

$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \lambda_{reg}\|\boldsymbol{w}\|_1. \tag{2.56}$$

Again, we will study the effect of L^1 regularization on the linear regression problem. Like before, λ_{reg} is the regularization strength. The sub-gradient of the regularized objective function (because absolute value is not a differentiable function) is:

$$\nabla_{\boldsymbol{w}} J_{reg} = \lambda \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} J, \tag{2.57}$$

where

$$\text{sign}(x) = \begin{cases} +1 & x > 0, \\ -1 & x < 0 \end{cases}.$$

L^1 has a much different effect on the weights than L^2 regularization. Now, the regularization contribution to the gradient does not scale linearly with each w_i. Instead, it is a constant and its sign is given by sign(w_i). A consequence of that is that we cannot have clean algebraic expressions to the quadratic approximation as before.

By making some assumptions for the Hessian matrix of the objective function, we can derive a quadratic approximation which will have the following closed form solution:

$$w_i = \text{sign}(w_i^*) \max\{|w_i| - \frac{\lambda_{reg}}{H_{i,i}}, 0\}, \tag{2.58}$$

where w_i^* is the point at which we evaluate the quadratic approximation and $H_{i,i}$ is the i-th diagonal element of the Hessian matrix. It is clear that L^1 regularization provides sparse solutions, i.e. weight vectors where several elements are zero. The behavior of L^1 regularization is much different than L^2 regularization. While the first selects the most important components of the weight vector resulting in a sparse weight vector with zeros in all the unimportant components, the second rescales all the components relatively to their importance resulting in diffuse weight vectors with small magnitude. In L^1 regularization, λ_{reg} controls the sparsity of the weight vector. L^1 regularization has been used extensively as a feature selection mechanism. L^1 regularization can be seen as MAP Bayesian inference with an isotropic Laplace prior.

2.8.2 Dropout

Dropout [30] is a computationally efficient and powerful method for regularizing deep neural networks. As it is well known, bagging can be used for the regularization of models, as averaging several models reduces the variance in the predictions; hence, it reduces the overfitting effect. Bagging trains several models and evaluate the test examples on each model, and then computes the average over all models. Each model is trained with a dataset sampled from the training set with replacement. This is impractical for deep neural networks, as the training of a deep neural network is a very computationally expensive procedure and to train a big number of models requires a lot of computational resources.

Dropout is a computationally inexpensive approximation of training and evaluating a bagged ensemble of exponentially many neural networks. Dropout trains the ensemble of all possible sub-networks that can be formed by removing hidden units from a single neural network. Instead of removing hidden units dropout multiply the units that are about to be removed with zero.

Each time we sample a mini-batch to perform parameter updates, we sample a different random binary mask $\boldsymbol{m}^{(k)}$ for the k-th layer and the size of a mask is the same as the number of hidden units in layer k. As we said, instead of dropping hidden units, hidden units are multiplied with the binary mask $\boldsymbol{m}^{(k)}$. Now, equation (2.3) which computes the activations of hidden units of layer k becomes:

$$\boldsymbol{h}^{(k)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{a}^{(k)}(\boldsymbol{x})) \odot \boldsymbol{m}^{(k)}. \tag{2.59}$$

As the activation of hidden units are involved in some of the back-propagation algorithm steps, some gradients are also affected. Specifically:

$$\nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right] \Longleftarrow \left(\nabla_{\boldsymbol{h}^{(k-1)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right]\right)^{\top} \nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} \boldsymbol{h}^{(k-1)}(\boldsymbol{x}),$$

now becomes

$$\nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right] \Longleftarrow \left(\nabla_{\boldsymbol{h}^{(k-1)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right]\right)^{\top} \nabla_{\boldsymbol{a}^{(k-1)}(\boldsymbol{x})} \boldsymbol{h}^{(k-1)}(\boldsymbol{x}) \odot \boldsymbol{m}^{(k-1)},$$

and

$$\nabla_{\boldsymbol{W}^{(k)}} \left[-\log f(\boldsymbol{x})_y\right] \Longleftarrow \left(\nabla_{\boldsymbol{a}^{(k)}(\boldsymbol{x})} \left[-\log f(\boldsymbol{x})_y\right]\right) \boldsymbol{h}^{(k-1)}(\boldsymbol{x})^{\top},$$

contains $\boldsymbol{h}^{(k-1)}$ which is multiplied with the binary mask. Thus, on each forward propagation and back-propagation the dropped out hidden units are zero as well as their gradients. This is the same as removing this hidden units and train at each step the sub-network that is formed by removing these hidden units. The mask of each hidden unit is sampled independently. The probability of sampling a mask of 1, or in other words the probability of leaving a hidden unit ”turned on” is a hyperparameter that should be tuned. Nevertheless, a standard default is 0.5 which means that at each pass half of the hidden units are removed.

In bagging, all of the models are trained and consequently all of them participate in inference. In dropout, all the possible models are 2^d where d is the number of hidden units that may be dropped from the network. This becomes intractable for deep networks. Hence, only a tiny fraction of the possible subnetworks are trained, each one for a single step of the learning procedure. Also, all these sub-networks share the same parameters. Parameter sharing causes the subsequent subnetworks to arrive at good settings of the parameters. Thus, the basic differences of bagging and dropout is that in dropout we do not train all the possible models of our ensemble and the models share the same parameters while in bagging each model has its own parameters and trained independently from the others. One more difference is that in bagging the models are trained till convergence while in dropout each model is trained for a single step of a minibatch-based algorithm such as mini-batch SGD. Apart from that, dropout follows bagging. For example, the training set that is encountered by each subnetwork is a subset of the original dataset sampled with replacement.

In bagging, all models of the ensemble are voting so that the ensemble makes a prediction. If each of the models produces a probability distribution $p^{(i)}(y|\boldsymbol{x})$ and the ensemble consists of n models, the prediction of the ensemble is given by:

$$p_{ensemble}(y|\boldsymbol{x}) = \frac{1}{n}\sum_{1}^{n} p^{(i)}(y|\boldsymbol{x}). \tag{2.60}$$

The predictions of the ensemble in dropout that is produced by sampling random masks $\boldsymbol{m}$ is given by:

$$p_{ensemble}(y|\boldsymbol{x}) = \sum_{\boldsymbol{m}} p(\boldsymbol{m})p(y|\boldsymbol{x},\boldsymbol{m}). \tag{2.61}$$

In the case of deep neural networks, this sum contains and exponential number of terms, thus is intractable to compute. There is an effective approach that approximates the predictions of the entire ensemble in a single forward propagation pass. This approach incorporates the geometric mean instead of the arithmetic mean of the predictions of the ensemble's members and it can be shown that the geometric mean performs comparably to the arithmetic mean. The geometric mean of the ensemble's members predictions is given by:

$$p_{ensemble}(y|\boldsymbol{x}) = \sqrt[2^d]{\prod_{\boldsymbol{m}} p(y|\boldsymbol{x}, \boldsymbol{m})}. \tag{2.62}$$

It turns out that $p_{ensemble}(y|\boldsymbol{x})$ can be approximated with a single model: the neural network where all hidden units participate and the binary masks are replaced with their expectation. For example, if we use an inclusion probability of 0.5 the expectation of sampling a binary mask will be 0.5; thus, we replace all the entries in the binary mask vector with 0.5. The motivation is to make sure that the expected total input to a unit at test time is roughly the same as the expected total input to that unit at train time. Therefore, at the end of training we replace the binary masks with their expectations and we perform inference from that model, approximating the geometric average over the whole ensemble. There is no theoretical justification for the accuracy of this approximate inference but in practice it works very well. For networks that do not have nonlinearities this approximation is exact. For networks that have nonlinearities it is only an approximation, but works well.

Dropout trains an ensemble of models that share hidden units. This means that each hidden unit must perform well regardless of the other hidden units. This stands because in each model of the ensemble different set of units are present; hence, training pushes hidden units towards being able to perform well in the absence of other hidden units. Hidden units cannot co-adapt to each other as they don't rely to each other. This provides better generalization as hidden units learn more generally useful features and not very complex ones that fit perfectly the training set. Hidden units cannot develop very complicated co-adaptation patterns because they are not combined all together.

Chapter 3

Convolutional Networks

3.1 Introduction

Convolutional networks (ConvNets) [31], also known as convolutional neural networks (CNNs), are very successful deep learning models. ConvNets are a specialized kind of neural network for processing data that has a known, grid-like topology. Examples include time-series data, which can be thought of as a 1D grid taking samples at regular time intervals, and image data, which can be thought of as a 2D grid of pixels. In this work, we study the application of ConvNets to images as it is described in Chapter 1. The name is derived from the fact that the network deploys the mathematical operation of convolution in one or more of its layers instead of classical matrix multiplications of the input with the layer's weights for reasons which will be further described. In short, convolutional networks are designed in a way, such that they can handle very high-dimesnional inputs (an image might have thousands or millions of pixels). The idea is that while an image is usually high-dimensional, we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that computing the output (prediction) requires fewer operations. Moreover, convolutional networks provide invariance in certain variations such as translation. ConvNets leverages the aforementioned ideas by using three principles that are well-known in the deep learning community as *local connectivity*, *parameter sharing* and *equivariant represantations*.

3.2 Convolution

Since convolution is the fundamental operation of convolutional networks, it will be described here as well as some of its practical aspects. It is defined as the integral of the product of the two functions after one is reversed and shifted:

$$(x * w)(t) = \int x(\tau)w(t-\tau)d\tau. \tag{3.1}$$

While the symbol t is used above, it need not represent the time domain. But in that context, the convolution formula can be described as a weighted average of the function $x(\tau)$ at the moment t where the weighting is given by $w(-\tau)$ simply shifted by amount t. As t changes, the weighting function emphasizes different parts of the input function. In other words it expresses the amount of overlap of w as it is shifted over x. It therefore "blends" the two functions.

There is also **discrete convolution**, which is used when x and w are defined on the set $\mathbb{Z}$ of integers. Computers cannot represent real-valued intervals, thus we sample values at regular intervals, i.e the intervals are discretized. Examples are pixels of an image or time points of a song. Thus, we use discrete convolution which is defined as:

$$(x * w)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)w(t-\tau). \tag{3.2}$$

In convolutional network terminology, the first argument (in this example, the function x) to the convolution is often referred to as the input and the second argument (in this example, the function w) as the *kernel*. The output is sometimes referred to as the *feature map*.

Convolution can be extended to more dimensions, thus we can have 2D convolution. As we know from computer vision applications, this is useful for images because of their 2D structure. If we have a 2D image I as our input and a 2D kernel K, then 2D convolution of them is:

$$(I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{3.3}$$

Convolution is commutative, meaning we can equivalently write:

$$(K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n). \tag{3.4}$$

The expression (3.4) is more efficient for implementation, as kernels are in general smaller than images, hence summing over all kernel elements require fewer operations comparing to summing over all image pixels. Consequnetly, (3.4) is preferred for implementation. It can also be written as:

$$(K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(r-m, r-n), \tag{3.5}$$

where r is the number of rows and columns of the kernel assuming that the kernel is square. In (3.5), we flip the kernel and compute the inner-products at shifted locations of the original image, which is equivalent to flipping the image and computing the inner-products at shifted image locations using the original kernel, that is the case for (3.4).

In convolutional networks, we want the original weight matrix to operate on the inputs, and not a flipped version of it. To accomplish this, we use as kernel the weight matrix with flipped rows and columns, and the convolution (for instance (3.5)) re-flips the kernel; hence we obtain the original weight matrix. Alternatively we can apply **cross-correlation**:

$$(K \star I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n), \tag{3.6}$$

without flipping the weight matrix in prior. Machine learning libraries are using either (3.5) or (3.6) for the implementation of convolution. Since in (3.5) the weight matrix is actually flipped in prior, it is also cross-correlation in essence. Therefore, convolutional networks actually apply cross-correlation, but they are called **convolutional**, because they can be implemented using convolutions.

As it is well-known from computer vision, applying convolution to an image can be used for the extraction of image features such as edges, using the proper kernel. Convolution measures the amount of overlap of two functions, say an image and a filter, and wherever there is maximum overlap the image feature in question is detected, given that we use a proper kernel. Alternatively, as convolution applies inner products at multiple image locations (image patches), we can think the image patches and the kernel as vectors, where we get a maximum value when their inner product is maximum, i.e the angle between the two vectors is zero, and thus the image and the kernel overlap perfectly.

As we will see later, the difference between classical computer vision and convolutional networks is that in classical computer vision we use predefined kernels that are able to detect specific image features, whereas in convolutional networks, kernels are multidimensional arrays of parameters that are adapted by the learning algorithm.

3.3 Local connectivity

Given an image as input (this generalizes to other types of high-dimensional data), a classical fully-connected neural net would require an unmanageable number of parameters, since each hidden unit in the first layer would be connected to each image pixel. To deal with this, convolutional networks introduce the idea of local connectivity. Hidden units in a convolutional network are not connected to each unit of the layer below but just to a local neighborhood of the input.

Typically, a square patch is used as the local neighborhood of the layer below. The patch of the image that a hidden unit is connected with, is called *receptive field* of the hidden unit. A graphical illustration of local connectivity in ConvNets can be seen

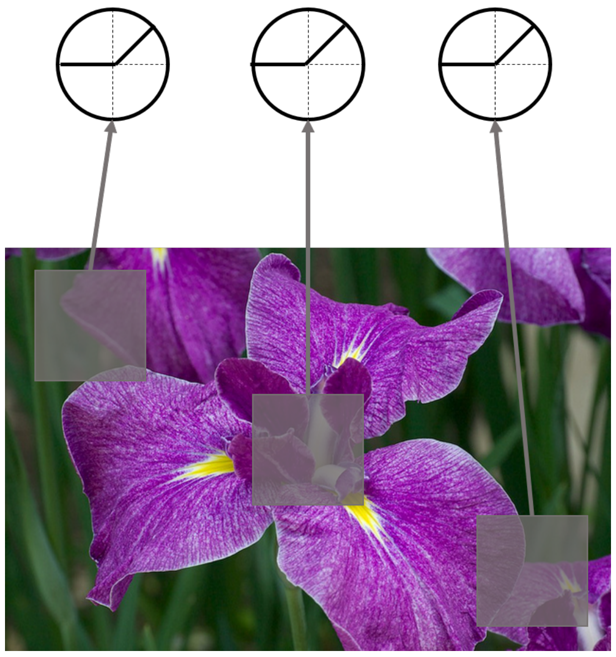


Figure 3.1: The idea of local connectivity in convolutional networks. Gray squares are the image patches that hidden units are connected with, or the receptive fields of hidden units. The circles on top of the image represent the hidden units and the arrows show which hidden unit is connected with which patch of the image. In this example, it is clear that hidden units have local connections and that different hidden units are used for different local neighborhoods, where we use three different hidden units to show the latter.

in figure 3.1. Gray squares represent the receptive fields of different hidden units. Hidden units are represented with the circles above the image and the arrows denote which hidden unit is connected with which image patch. In this example, it is clear that hidden units have local connections and that different hidden units are used for different local neighborhoods.

Moreover, in ConvNets a hidden unit is connected to all input channels, e.g in RGB images a hidden unit has connections with three different channels (R, G, B channels) while in gray-scale images it has connections with a single channel. Different hidden units are connected to different patches of the input image, such that the image is covered with the receptive fields of all hidden units.

Local connectivity is accomplished by performing convolutions to an image using a kernel smaller than the image. The correspondence of performing convolutions to an image, with a neural network perspective is that the result of convolution at a

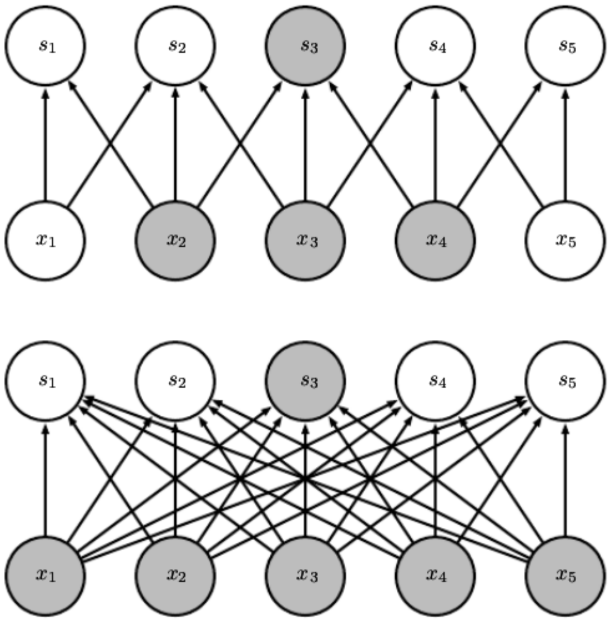


Figure 3.2: The difference between a fully-connected neural network and a convolutional network. The fully-connected net is shown at the bottom while a 1D ConvNet is shown at the top. The colors denote the hidden units from the layer below that a hidden unit is connected with. Each hidden unit in the fully-connected net is connected with each unit from the layer below. The ConvNet has a receptive field of size three; hence, each hidden unit is connected only to three neighboring units from the layer below. Figure reproduced from [1].

particular location is the preactivation of a hidden unit, while convolutions at multiple shifted image locations correspond to different hidden units. The image patch where the convolutional kernel overlaps is the receptive field of the respective hidden unit. Using a kernel with a size equal to the image size, we result in a classical fully-connected network. In figure (3.2), we see the difference between a fully-connected neural net and a ConvNet for the 1D case. The fully-connected net is shown at the bottom while the ConvNet is shown at the top. The colors denote the hidden units from the layer below that a hidden unit is connected with. While the hidden units of the fully-connected net are connected with each unit of the layer below, the ConvNet uses a receptive field of size three and each hidden unit is connected only to three neighboring units from below. Here, we want to emphasize that while a ConvNet can be implemented by performing convolutions, it can also be represented and implemented with a classical neural network representation but with sparse connections. Thus, the correspondence between performing convolutions at shifted image locations and a neural net with sparse connections is direct.

Local connectivity reduces significantly the number of parameters of the model, since each hidden unit is connected with a small local subset of the input, hence the weights are fewer comparing to a fully-connected net. This tackles two major problems. Firstly, the memory requirements of the model are significantly reduced since the parameters are fewer, and secondly fewer operations are required for computing the pre-activation of a hidden unit since the connections are fewer; hence the computation time is reduced.

3.4 Parameter sharing

A second idea that is added on top of local connectivity is called parameter sharing. As we explained in Section 3.3, in a ConvNet a hidden unit is connected to a patch of the image and different hidden units are connected to different patches of the image, covering the whole image. The hidden units that cover the whole image are grouped together in a, so called, *feature map*. Parameter sharing means that all hidden units in a feature map share the same parameters, i.e. they use the same weight matrix to compute their preactivations w.r.t. the local image patch they are connected.

In convolutional networks, parameter sharing is implemented by convolving the image with the same kernel, over all possible image locations. The result of this convolution is the layer's feature map which again can be seen as an image as it has a 2D structure and this is convenient in the context of ConvNets, since local connectivity and parameter sharing can also be applied to subsequent layers, as they take images as inputs (the 2D feature maps). We obtain several different feature maps by using different kernels. Thus, we convolve the image with different kernels to obtain different feature maps, where the hidden units within a feature map (feature map spatial locations) share the same kernel, i.e. a single feature map is the result of convolving the input with the same kernel over all spatial locations of the input. A graphical illustration of both local connectivity and parameter sharing can be seen in figure 3.3.

With parameter sharing, the number of parameters is reduced even more, as in a feature map all hidden units share the same kernel. Another big benefit of convolving the image with the same kernel over all spatial locations, is that we obtain

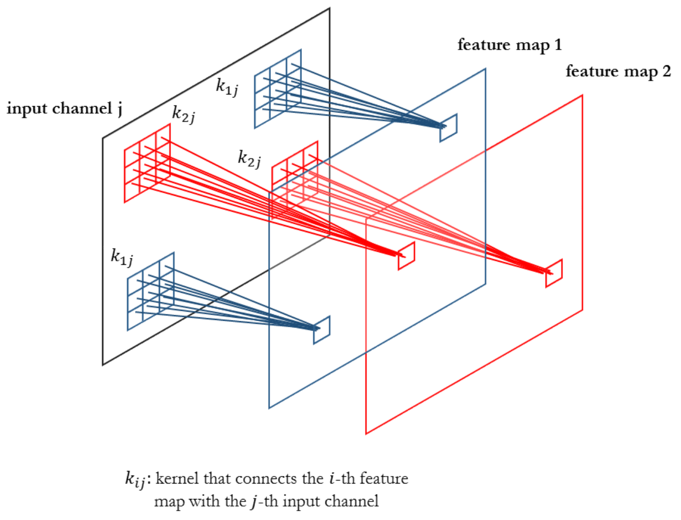

feature map 1
feature map 2
input channel j
k_{1j}
k_{2j}
k_{2j}
k_{1j}
k_{ij}: kernel that connects the i-th feature map with the j-th input channel

Figure 3.3: Local connectivity and parameter sharing in convolutional networks. Different colors correspond to different feature maps. You can see that within each feature map, hidden units are connected only to a subset of their input (local connectivity) and all the hidden units within a feature map share the same kernel (parameter sharing). Different feature maps have different kernels.

a property called equivariance to translation. A function is equivariant means that if the input changes, the output changes in the same way. Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, then the convolution function is equivariant to g. For example, let I be an image. Let g be a function that translates an image, such that $I' = g(I)$ is the image with $I'(x, y) = I(x-1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I, then applied the transformation g to the output. Convolution creates a feature map of the locations of detected features in the image, where the feature is specified by the kernel that we are using. Equivariance means that if we move an object in an image, its detected features will be moved by the same amount at the feature map. This suggests that if we have a kernel that detects, for example horizontal edges and convolve an image with this kernel, in the output feature map, edges will be detected in all possible positions. Thus, in each feature map specific features based on the kernel that is used are detected, e.g. the first kernel detects horizontal edges, the second vertical edges, the third diagonal edges and so forth.

3.5 Basic structure of a Convolutional Network

In this section, we will put together all the principles that we analyzed previously in this chapter, and we will describe all the building blocks for constructing a convolutional network. A ConvNet is composed of different types of layers. Typically, two are the main type of layers that exist almost always in a ConvNet, convolutional layers and pooling layers. The last layer is the output layer where we obtain the predictions of the model. Between the input layer, where we feed images to the model, and the output layer, there are several convolution and pooling layers. Usually a pooling layer follows a convolutional layer. Each component of such an archtecture is described below in details. Finally, we describe some famous ConvNet architectures which gave impressive results in given classification tasks.

3.5.1 Convolutional layer

A convolutional layer takes a set of input channels and produces as output a set of feature maps. The number of feature maps to be produced is arbitrarily defined by the user. Similarly to fully-connected layers, a convolutional layer first computes its preactivations which are subsequently given to a non-linear function to produce the layer's activations. While the activations are computed identically as in a fully-connected layer, the way the preactivations are computed changes, since now the operation of convolution is employed. Therefore, a convolutional layer's computations are performed in two steps as:

$$\boldsymbol{a}^{(n)}(\boldsymbol{x})_{ij} = \boldsymbol{h}^{(n-1)}(\boldsymbol{x})_i * \boldsymbol{k}_{ij}^{(n)} \tag{3.7}$$

$$\boldsymbol{h}^{(n)}(\boldsymbol{x})_j = g(\sum_i \boldsymbol{a}^{(n)}(\boldsymbol{x})_{ij} + b_j^{(n)}), \tag{3.8}$$

where $\boldsymbol{x}$ is the input image, n is the current convolutional layer, $\boldsymbol{h}^{(n-1)}(\boldsymbol{x})_i$ is the i-th input channel, $\boldsymbol{k}_{ij}^{(n)}$ is the kernel of the current convolutional layer that connects the i-th input channel of the $(n-1)$-th layer with the j-th feature map of the n-th layer, $b_j^{(n)}$ is the bias of the j-th feature map and $g(\cdot)$ is an activation function. Finally, $\boldsymbol{a}^{(n)}(\boldsymbol{x})_{ij}$ are the preactivations of the current convolutional layer which result from the i-th input channel and participate in the j-th feature map, while $\boldsymbol{h}^{(n)}(\boldsymbol{x})_j$ is the j-th feature map of the current convolutional layer.

To produce a single feature map $\boldsymbol{h}^{(n)}(\boldsymbol{x})_j$, the convolutional layer computes the preactivations $\boldsymbol{a}^{(n)}(\boldsymbol{x})_{ij}$ for each input channel $\boldsymbol{h}^{(n-1)}(\boldsymbol{x})_i$, independently. In (3.7), the preactivation $\boldsymbol{a}^{(n)}(\boldsymbol{x})_{ij}$ is computed by convolving the i-th input channel with $\boldsymbol{k}_{ij}^{(n)}$. Firstly, the preactivations are computed using (3.7), for each input channel independently, and subsequently the activations for the j-th feature map are computed using (3.8), by summing the preactivations from each input channel plus a bias $b_j^{(n)}$ for the j-th feature map and giving the result as input to an activation function. The summation is performed element-wisely, i.e. each spatial location (i, j) in the summed preactivation map is the result of summing the preactivation from each input channel at spatial location (i, j). It can be seen as computing a weighted combination of the input channels, where the weighted combination for each spatial location is not computed w.r.t. a single element from each input channel, but w.r.t. a local neighborhood around that element. In other words, instead of multiplying each input channel with a single element and summing the results which would be the classical definition of weighted combination, we convolve each input channel with a kernel and then sum the results which is a weighted combination that takes into consideration the neighboring values at each spatial location.

The above process is performed as many times as the number of feature maps that the layer produces, using different set of kernels $\boldsymbol{k}_{ij}$ for each feature map. The total number of kernels that a convolutional network utilizes is (# input channels $\times$ # feature maps), while for a single feature map (# input channels) feature maps are used. Lastly, because of parameter sharing, a single bias is used for each feature map instead of using different biases for each hidden unit of the feature map.

A graphical demonstration of performing the computations of a convolutional layer by applying equations (3.7) and (3.8) can be seen in figure 3.4. In this example, the convolutional layer takes three input channels (red, green and blue) and produces a single feature map (the purple one). With this graphical example, we want to emphasize that each spatial location of the resulting feature map is the result of convolving each input channel with a different kernel at the corresponding spatial locations and subsequently summing the results. This is demonstrated in this example with the small gray element in the resulting feature map which is connected with the corresponding convolved gray areas in the input channels. Note that on top of the gray areas in the input channels, we show the corresponding kernel to emphasize that a different kernel is used for each input channel. A nonlinearity and a bias follow

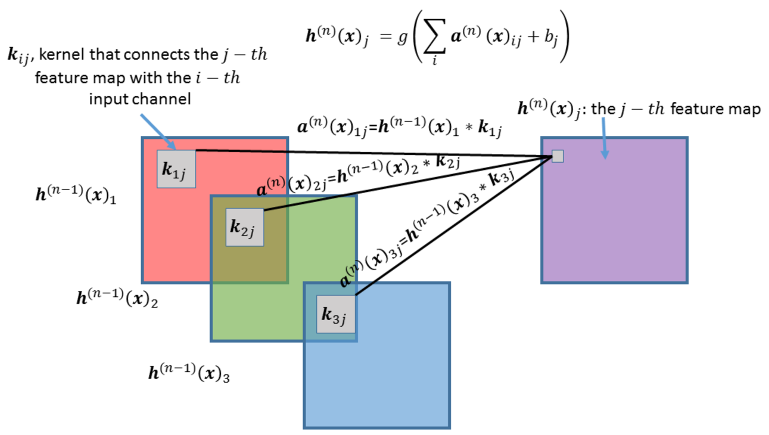

$h^{(n)}(x)_j = g\left(\sum_i a^{(n)}(x)_{ij} + b_j\right)$
k_{ij}, kernel that connects the $j-th$ feature map with the $i-th$ input channel
$h^{(n)}(x)_j$: the $j-th$ feature map
$a^{(n)}(x)_{1j}=h^{(n-1)}(x)_1 * k_{1j}$
$a^{(n)}(x)_{2j}=h^{(n-1)}(x)_2 * k_{2j}$
$a^{(n)}(x)_{3j}=h^{(n-1)}(x)_3 * k_{3j}$
$h^{(n-1)}(x)_1$
$h^{(n-1)}(x)_2$
$h^{(n-1)}(x)_3$
k_{1j}
k_{2j}
k_{3j}

Figure 3.4: A typical convolutional layer. This convolutional layer takes three input channels (red, green and blue) and produces a single feature map (the purple one). Each position in the resulting feature map is the result of performing convolutions over all possible input channels and then summing the results. The resulting position in the feature map is the small gray element while the areas that are to be convolved and then subsequently summed are the bigger gray squares in the input channels. We show the kernel that is used on top of each convolved area to emphasize that a different kernel is used for each input channel. Finally, nonlinearities are applied to compute the feature map activations. The bias is shared across the hidden units of the feature map (parameter sharing).

the summation of the convolutions.

Note, that the feature maps that are produced at the current layer are used as input channels for the subsequent layer. Also at the first convolutional layer, the input channels from the previous layer are the input image channels, i.e. $\boldsymbol{h}^{(0)}(\boldsymbol{x})_i = \boldsymbol{x}_i$ where $\boldsymbol{x}_i$ is the i-th channel of the input image. An RGB image has three channels, and thus for each feature map three kernels are needed, while for a gray-scale image only one is needed.

3.5.2 Pooling layers

On top of convolutional layers, we usually add another operation that is called pooling and subsampling. A pooling function aggregates activations of feature maps in a local neighborhood by replacing the activations on that neighborhood with a summary of

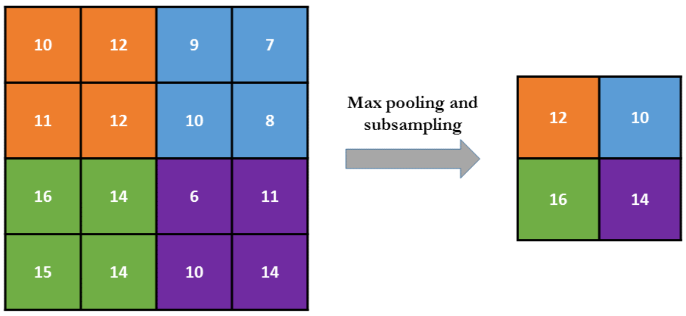


Figure 3.5: The max pooling operation. Left: The feature map before pooling and subsampling. Right: The pooled and subsampled feature map. In the unpooled feature map, different colors denote the different non-overlapping neighborhoods while the same colors to the right show the correspondence between the neighborhoods and their pooled values.

their statistics. Two main pooling operations are used in convolutional networks, *max pooling* and *average pooling*. Both are implemented as layers of the ConvNet.

In max pooling, we define small local neighborhoods of feature maps, i.e. square patches of the feature map, and we compute the maximum of the activations on each patch. The new, pooled feature map of the max pooling layer is constructed by replacing the neighborhoods with their respective maximum (i.e. the pooled value).

The max pooling operation is defined as:

$$h_{ijk}^{(n)}(\boldsymbol{x}) = \max_{p,q} h_{i,j+p,k+q}^{(n-1)}(\boldsymbol{x}), \tag{3.9}$$

where p and q are indexes of a patch of the feature map $\boldsymbol{h}_i^{(n-1)}$ of the previous layer, which is a convolutional layer and $h_{ijk}^{(n)}(\boldsymbol{x})$ is the corresponding pooled value for the i-th resulting feature map at position (j,k) of the current max pooling layer. Additionally, in general, max pooling is performed in non overlapping regions. This means that if we pool $l \times l$ regions, neighboring regions are l pixels apart which results in feature maps with reduced size. It is called pooling, because we pool the max value over local neighborhoods and subsampling because the size of the feature map is reduced by replacing the whole neighborhood with the max value. In figure 3.5, you can see a graphical depiction of max pooling. The feature map before max-pooling is shown at the left and the pooled and subsampled feature map is shown at the right. In the unpooled feature map, different colors denote the different non-overlapping neighborhoods while the same colors to the right show the correspondence between the neighborhoods and their pooled values.

Similar to max pooling, another pooling operation is called average pooling. The idea is the same as before with the only difference that now we are computing the average of the local neighborhood:

$$h_{ijk}^{(n)}(\boldsymbol{x}) = \frac{1}{m^2} \sum_{p,q} h_{i,j+p,k+q}^{(n-1)}(\boldsymbol{x}), \tag{3.10}$$

where m is the size of the neighborhood.

There are two main advantages of performing pooling and subsampling. The first is that it reduces the size of a feature map. This improves the computational efficiency of the network because the next layer has fewer inputs to process. The second advantage of pooling (mainly of max-pooling) is that it helps to make the representation invariant to small translations of the input. This can be explained as, if we perform small translations to the input and the position of the max remains in the same pooling neighborhood, max pooling will result to the same value as if we wouldn't translate the input, as the maximum of that neighborhood is still the same. This introduces invariance as a pooling and subsampling layer detects the same features on both an input and a translated version of it, given that the translation is relatively small.

3.5.3 Forward propagation in a convolutional network

As the fundamental elements of a convolutional network were presented, we will describe a typical forward propagation of information in a ConvNet.

A convolutional network consists mainly of convolutional, pooling and fully connected layers. The model is composed of two main parts, the feature extraction part and the classification part, in respective order. The feature extraction part consists mainly of convolutional and pooling layers. Typically, a pooling layer follows a convolutional layer, but it is not necessary to place a pooling layer after each convolutional layer. Generally, the layout of the network is arbitrary and we will discuss later in this chapter about design patterns and famous ConvNet layouts. In this part, we learn data-driven representations of our dataset, which provide discrimination across

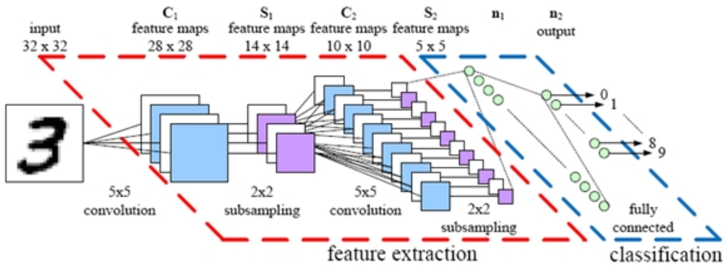


Figure 3.6: The typical structure of ConvNets. A convolutional network consists of two parts, the feature extraction part and the classification part. The red part of the network is the feature extraction part where the model learns image representations while the blue part is a fully-connected neural net which learns to classify images. Figure reproduced from [5].

different classes. The representations are learned through learning the convolutional kernels with back-propagation. Data-driven represantations means that the learned features are tied with the objects represented in our dataset, e.g. if a dataset contains cats and dogs the ConvNet will learn features that discriminate well between these two categories. In the classification part of the model, we use the learned features from the previous part to classify our data. This part of the model is in general a fully-connected neural net. In other words, we connect a fully-connected neural net with the last convolutional or pooling layer of the feature extraction part by vectorizing the feature maps of the last layer of the feature extraction part and use this vector as input for the classification part of the model.

This idea is clearly visible in figure 3.6. The red part is the feature extraction part which consists of two convolutional layers, each one followed by a pooling layer. The blue part is the classification part where in the specific example consists of a hidden layer and the output layer, both fully-connected. As you can see, the feature maps from the last pooling layer are connected to the hidden layer of the classification part. In this example, the model is trained on images that contain digits; hence, the output layer has ten output units.

A forward propagation in a ConvNet consists of providing an image as input to the convolutional network, and traversing such a structure described above, in order to finally compute class predictions at the output layer of the network. While traversing the convolutional network, convolutional and pooling layers compute the learned feature maps corresponding to the input, while the fully-connected layers compute

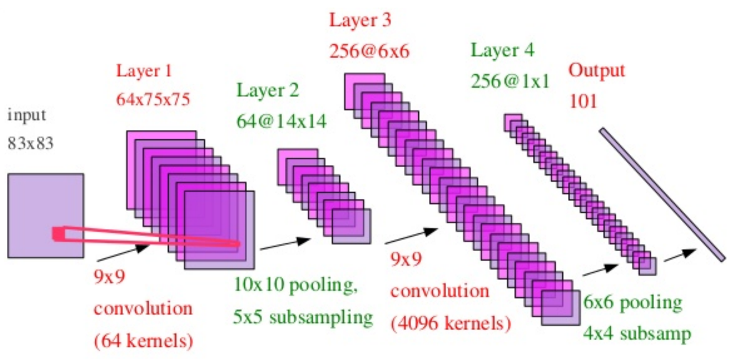


Figure 3.7: A typical forward propagation in convolutional networks. The network takes as input an image and produces in the output layer class predictions for 101 classes. It takes as input an 83×83 image. It consists of two convolutional layers each one followed by a pooling layer. The first convolutional layer has 64 kernels of size 9×9 and produces 64 feature maps of size 75×75. Then, the pooling layer reduces the size of the feature maps and a similar procedure is applied to the subsequent convolutional and pooling layer. The output of the last pooling layer is connected to the output layer that classifies images in 101 categories (Caltech 101 dataset [6]). Figure reproduced from [7].

the hidden and output units activations. An example of a typical forward propagation in a ConvNet is illustrated in figure 3.7. We can see that the first convolutional layer produces 64 feature maps by applying 64 convolution kernels to the input image. Since the input image has a single channel, a single kernel is used for each feature map. After the convolution, the size of the feature maps is reduced to 75×75 from 83×83. The first pooling layer, or Layer 2 at the specific example, reduces the size of each feature map to 14×14. The reduction in the size of the feature maps that is caused by convolutional and pooling layers depends on specific design choices which will be described in Section 3.5.4. The second convolutional layer (Layer 3), consists of 4096 convolution kernels, in total. That's because it is connected with 64 input channels from the previous layer; thus, we need 64 kernels to produce a single feature map, plus we want to obtain 256 feature maps, which means $256 \times 64 = 4096$ kernels in total. After the last pooling layer (Layer 4), we result in 256 feature maps of size 1×1 which we vectorize, and connect it with a fully-connected layer, to obtain class predictions.

As it is the case generally for deep learning, ConvNets learn hierarchical feature representations, which means that more complicated concepts are build from simpler

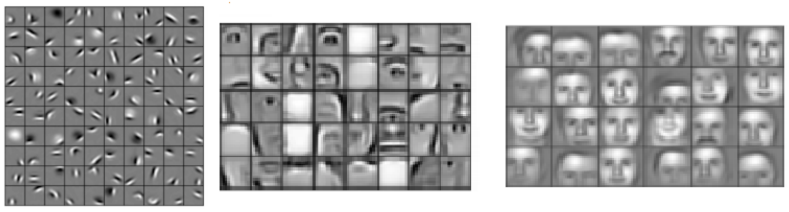


Figure 3.8: Hierarchical feature representations learned by a ConvNet. Here the model is applied to face recognition. Figure reproduced from [8]

ones. In ConvNets where the inputs are images, these concepts correspond to image features. Thus, the first convolutional layer learns simple features such as edges, the second layer combines these edges to construct more complicated features such as parts of an object and as we go on, we add more levels of abstraction, and eventually in the last convolutional layer we obtain highly abstract features, i.e. representations of the object in question. For example, in figure 3.8 where we classify faces, in the first layer the model learns edges that correspond to edges in the face, the second learns parts of the face such as eyes, mouths, noses, and the last one learns abstract representations of the face. This happens because a convolutional layer computes each feature map as a weighted combination of the input channels. Hence, the convolutional kernels are learned in a way, such that the weighted combination of the input channels can provide meaningful higher level representations. For example, the weighted combination of edges can form a part of an object. In digit recognition, a weighted combination of edges can form a digit. The reason that these weighted combinations provide meaningful representations, is that the problem is supervised; hence, we know what is represented in each image and we are able to learn the weights taking into consideration class attributes.

3.5.4 Design patterns

In this section, we will describe design patterns that affect the size of the feature maps at the layers of a convolutional network. To do so, first we will explain the notion of *zero-padding* and *stride*.

Convolution can be implemented in two different ways. The first way is that, we convolve the image at all positions where the kernel fits into the image, i.e there is no part of the kernel lying outside the image borders. This reduces the size of the feature map, as if for example we use a 3×3 kernel, we cannot convolve it at the image borders because part of the kernel falls outside the image. The second way, is to add the required number of columns and rows of zeros around the image (zero-padding), so that the kernel can fit in the whole original image. For example using a 3×3 kernel we need to zero-pad the image with padding $= 1$ element around the borders.

Stride refers to the amount of translation of our filters in order to perform convolutions at different image positions. If the stride size is s, we move our filters s pixels apart from the current position to perform the subsequent convolution. This can again affect the feature map size, as if $s > 1$ less convolutions are performed, and thus the activations in the feature map are fewer. As in pooling we pool and subsample in non overlapping neighborhoods, s is as big as the neighborhood size, e.g. if we pool and subsample over 2×2 neighborhood, then $s = 2$.

Given that, we have a feature map of size $w_1 \times h_1$, kernel size f, zero-padding p and stride s, the size of the new feature map after a convolutional layer, can be computed as:

$$\begin{aligned} w_2 &= \frac{w_1 - f + 2p}{s} + 1 \\ h_2 &= \frac{h_1 - f + 2p}{s} + 1, \end{aligned} \tag{3.11}$$

and after a pooling layer:

$$\begin{aligned} w_2 &= \frac{w_1 - f}{s} + 1 \\ h_2 &= \frac{h_1 - f}{s} + 1, \end{aligned} \tag{3.12}$$

since we don't use padding in pooling layers.

Lastly in this section, we will give some practical advices on designing layer sizing patterns which are described in [32].

First of all it is preferable to use a stack of convolutional layers with small filters to one convolutional layer with large receptive field. Suppose that we stack three convolutional layers with kernel size 3×3 on top of each other. In this arrangement, hidden units in the first convolutional layer have a receptive field of size 3×3 w.r.t the input image. Hidden units in the second convolutional layer have a receptive field of size 3×3 w.r.t. the first convolutional layer and by extension a receptive field of size 5×5 w.r.t. the input. Hidden units in the third convolutional layer have a receptive field of size 3×3 w.r.t. the second hidden layer; thus a receptive field of size 7×7 w.r.t. the input. An illustration of that can be seen at Figure 3.9. Suppose

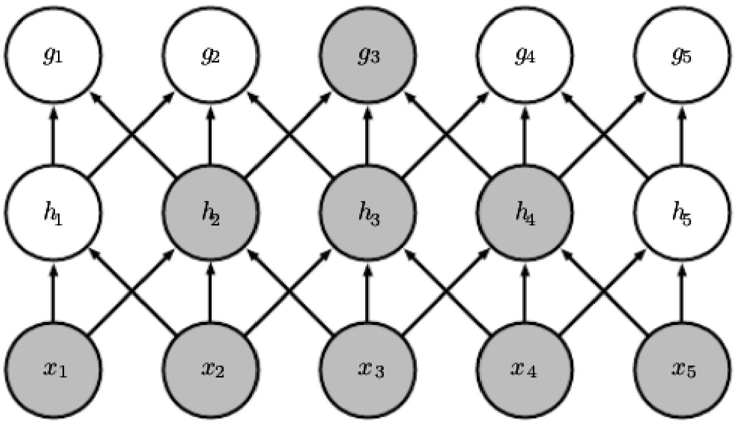


Figure 3.9: Stacking convolutional layers with small receptive fields results in the hidden units of the deeper layers to have large receptive fields. Each convolutional layer in the scheme has a receptive field of three w.r.t. the layer below. The hidden units of the first convolutional layer has a receptive field of 3 w.r.t. the input layer. The hidden units of the second convolutional layer has a receptive field of 3 w.r.t. the first convolutional layer but a receptive field of 5 w.r.t. the input layer. Figure reproduced from [1].

that instead of these three layers of 3×3 convolutions, we only wanted to use a single convolutional layer with 7×7 receptive fields. These neurons would have a receptive field size of the input that is identical in spatial extent (7×7), but with several disadvantages. First, the neurons would be computing a linear function over the input, while the three stacks of convolutional layers contain non-linearities that make their features more expressive. Second, if we suppose that all convolutional layers have C channels, then it can be seen that the single 7×7 convolutional layer would contain $C \times (7 \times 7 \times C) = 49C^2$ parameters, while the three 3×3 convolutional layers would only contain $3 \times (C \times (3 \times 3 \times C)) = 27C^2$ parameters. Intuitively, stacking convolutional layers with tiny filters as opposed to having one convolutional layer with big filters allows us to express more powerful features of the input, and with fewer parameters.

As arises from above, convolutional layers should have small kernels such as 3×3 or 5×5. Some architectures are incorporating 7×7 kernels but this only common to see in the first convolutional layers. One possible justification is that the design philosophy of the latter architectures is to allow the hidden units of deeper layers to indirectly have very large receptive fields on the input, which may arise in increased accuracy. Nevertheless, stacking several convolutional layers of 3×3 kernels works very well in practice as you will see in some popular ConvNet architectures that we will describe in the next section. Moreover, it is crucial to use a stride of $s = 1$ and zero-pad the inputs accordingly such that convolutional layers does not alter the feature maps size. For a kernel of size $f = 3$, $p = 1$ retains the feature maps spatial extent while for $f = 5$, $p = 2$. For a general f, it can be seen that $p = \frac{f-1}{2}$ preserves the input size. The pooling layers are in charge of downsampling the spatial dimensions of the input. The most common setting is to use max-pooling with 2×2 receptive fields, and with a stride of 2 which results in non-overlapping neighborhoods.

The scheme presented above is pleasing because all the convolution layers preserve the size of their input, while the pooling layers alone are responsible for for subsampling the inputs. By using $s = 1$ convolutional layers perform ”dense” convolutions which is more informative than performing convolutions at ”sparse” locations. Zero-padding preserves the information at the image borders. If we use convolutional layers without zero-padding, the size of the feature maps is reduced after each convolutional layer and the information at the image borders is trimmed and cannot be exploited. Thus, in the above setting the role of convolutional layers is to transform their inputs and output all the possible information and the role of pooling layers is to downsample their inputs.

Finally, as we usually subsample the input by a factor of 2 at pooling layers (convolutional layers preserve the spatial size), it is convenient for the input layer (that contains the image), to be divisible by 2 several times. Common numbers include 32 (e.g. CIFAR-10 [33]), 64, 96 (e.g. STL-10 dataset [34]), or 224 (e.g. common ImageNet [35] ConvNets), 384, and 512.

3.6 Gradients of convolutional and pooling layers

In Section 2.6, we have seen how to apply backpropagation in order to compute the gradient of the weights for fully-connected layers in feedforward neural networks. For applying backpropagation in a ConvNet, the only missing parts are the computation of the gradients of convolutional and pooling layers which we will derive in this section. The backpropagation of gradients for fully-connected layers in convolutional networks is identical with that of feedforward neural networks; hence they will not be restated here.

For brevity, we denote the loss function by L. Furthermore, for the simplicity of illustration we consider the case of using a stride of $s = 1$ and zero-padding $p = 0$ for the convolutional layers. Hence, the resulting feature maps from the convolutional layers have size $(w-f+1)\times(h-f+1)$ according to (3.11), where w and h are the width and height of the input respectively and f is the size of the kernel. Lastly, we derive the gradients of convolutional layers for the simple case where the convolutional layer takes as input a single channel and produces as output a single feature map. Hence, we drop the i, j indices from the kernel $\boldsymbol{k}_{ij}$ which connects the i-th input channel with the j-th feature map. k_{ij} now refers to a particular spatial location of kernel $\boldsymbol{k}$, that is its i-th row and the j-th column. Equations (3.7) and (3.8) now become:

$$a_{ij}^{(n)} = \boldsymbol{h}_{ij}^{(n-1)} * \boldsymbol{k}_{ij}^{(n)} = \sum_p \sum_q h_{i-p,j-q}^{(n-1)} k_{pq}^{(n)} \tag{3.13}$$

$$\boldsymbol{h}^{(n)} = g(\boldsymbol{a}^{(n)} + b^{(n)}), \tag{3.14}$$

where in (3.13) we employed the definition of convolution given in (3.4) and in (3.14) the sum from (3.8) is dropped since we consider the case of a single input channel. Moreover, we dropped the dependence on the input $\boldsymbol{x}$ which in (3.7) and in (3.8) is denoted in parentheses, for the simplicity of illustration.

We start by deriving the derivative of the loss w.r.t. each weight $k_{pq}^{(n)}$ of kernel $\boldsymbol{k}^{(n)}$. We use the chain rule where according to (2.24) we must sum the contributions of all expressions in which $k_{pq}^{(n)}$ occurs. It is given by:

$$\frac{\partial L}{\partial k_{pq}^{(n)}} = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}} \frac{\partial a_{ij}^{(n)}}{\partial k_{pq}^{(n)}} = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}} h_{i-p,j-q}^{(n-1)}. \tag{3.15}$$

In the above expression, we can easily derive that $\frac{\partial a_{ij}^{(n)}}{\partial k_{pq}^{(n)}} = h_{i-p,j-q}^{(n-1)}$ by looking at (3.13). Also an important observation is that while in (2.31) for the derivative of the weights of a fully-connected layer we do not sum over different expressions in which the weights participate, here we do. The reason is that in a fully-connected layer each weight occurs exactly once and participates in the inner product for producing a single hidden unit. In convolutional layers where we have the property of parameter sharing, each weight participates in all convolutions for producing each hidden unit of a feature map; hence, in the computation of the derivative of the weight we sum over all hidden units' preactivations for which the weight occurs. Equation (3.15) is a convolution, where the rows and columns of $h_{i,j}^{(n-1)}$ are flipped. Hence, (3.15) can be computed as:

$$\frac{\partial L}{\partial k_{pq}^{(n)}} = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}} h_{i-p,j-q}^{(n-1)} = \frac{\partial L}{\partial a_{ij}^{(n)}} * h_{-i,-j}^{(n-1)} = \frac{\partial L}{\partial a_{ij}^{(n)}} * \text{flip}(h_{i,j}^{(n-1)}), \tag{3.16}$$

where flip$(\cdot)$ flips the rows and columns of a matrix. In gradient form it is written as:

$$\nabla_{\boldsymbol{k}^{(n)}} L = \nabla_{\boldsymbol{a}^{(n)}} L * \text{flip}(\boldsymbol{h}^{(n-1)}), \tag{3.17}$$

where the convolution is performed over all possible locations.

The partial derivative of the loss w.r.t. the feature map shared bias is computed as:

$$\frac{\partial L}{\partial b^{(n)}} = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}} \frac{\partial a_{ij}^{(n)}}{\partial b^{(n)}} = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}}. \tag{3.18}$$

For consistency of illustration, we show (3.18) in gradient form (even if it is the same with the partial derivative since it is a single element), which is:

$$\nabla_{b^{(n)}} L = \sum_i \sum_j \frac{\partial L}{\partial a_{ij}^{(n)}}. \tag{3.19}$$

Prior to the computation of (3.15) and (3.18), we need to compute the term $\frac{\partial L}{\partial a_{ij}^{(n)}}$, i.e. the partial derivative of the loss w.r.t. the convolutional layer's preactivation. We also use the chain rule:

$$\frac{\partial L}{\partial a_{ij}^{(n)}} = \frac{\partial L}{\partial h_{ij}^{(n)}} \frac{\partial h_{ij}^{(n)}}{\partial a_{ij}^{(n)}} = \frac{\partial L}{\partial h_{ij}^{(n)}} \frac{\partial g(a_{ij}^{(n)})}{\partial a_{ij}^{(n)}} = \frac{\partial L}{\partial h_{ij}^{(n)}} g'(a_{ij}^{(n)}), \tag{3.20}$$

where we have seen in Section 2.6, the gradients $g'(\cdot)$, of various activation functions. Note that (3.20) is identical to (2.28) that is the partial derivative of the loss w.r.t. a fully-connected layer's preactivation. Hence, the gradient is identical to the corresponding gradient of (2.28), which we derive in (2.29), with the only difference that now the element-wise multiplication is between two matrices and not two vectors.

Finally, for a convolutional layer, we need the partial derivative of the loss w.r.t. the layer's activation:

$$\frac{\partial L}{\partial h_{ij}^{(n)}} = \sum_p \sum_q \frac{\partial L}{\partial a_{i+p,j+q}^{(n+1)}} \frac{\partial a_{i+p,j+q}^{(n+1)}}{\partial h_{ij}^{(n)}} = \sum_p \sum_q \frac{\partial L}{\partial a_{i+p,j+q}^{(n+1)}} k_{pq}^{(n+1)}, \tag{3.21}$$

where we can easily see from (3.13) that

$$a_{i+p,j+q}^{(n+1)} = \sum_p \sum_q h_{ij}^{(n)} k_{pq}^{(n+1)} + b^{(n+1)},$$

and

$$\frac{\partial a_{i+p,j+q}^{(n+1)}}{\partial h_{ij}^{(n)}} = k_{pq}^{(n+1)}.$$

It is apparent that (3.21) is a cross-correlation. Again, we can compute it as a convolution by flipping the rows and the columns of $\frac{\partial L}{\partial a_{i+p,j+q}^{(n+1)}}$. Alternatively, if we use the equivalent definition for the convolution which is described in (3.5), we can compute (3.21) as a convolution by flipping the kernel. Hence:

$$\begin{aligned} \frac{\partial L}{\partial h_{ij}^{(n)}} &= \sum_p \sum_q \frac{\partial L}{\partial a_{i+p,j+q}^{(n+1)}} k_{pq}^{(n+1)} = \text{flip}\left(\frac{\partial L}{\partial a_{ij}^{(n+1)}}\right) * k_{ij}^{(n+1)} \\ &= \frac{\partial L}{\partial a_{ij}^{(n+1)}} * \text{flip}\left(k_{ij}^{(n+1)}\right). \end{aligned} \tag{3.22}$$

Practically, the definition in (3.5) is used for the implementation of convolution; hence, in (3.22) we flip the kernel instead of the gradient of the loss w.r.t. the preactivations of the layer above. Since we use a zero-padding of $p = 0$ for the convolution, the size of $\boldsymbol{a}^{(n+1)}$ will be reduced comparing to $\boldsymbol{h}^{(n)}$. Hence, $\nabla_{\boldsymbol{h}^{(n)}} L$ will be bigger than $\nabla_{\boldsymbol{a}^{(n+1)}} L$. As a solution, we zero-pad $\nabla_{\boldsymbol{a}^{(n+1)}} L$ accordingly. Using a kernel of size f, we zero-pad $\nabla_{\boldsymbol{a}^{(n+1)}} L$ with $2(f-1)$ zeros. Therefore, the gradient of the loss w.r.t. a convolutional layer's activation is given by:

$$\nabla_{\boldsymbol{h}^{(n)}} L = \nabla_{\boldsymbol{a}^{(n+1)}} L \circledast \text{flip}(\boldsymbol{k}^{(n+1)}), \tag{3.23}$$

where $\circledast$ denotes a zero-padded convolution.

We derived the necessary gradient formulas for backpropagating the gradients through a convolutional layer, assuming that we use a stride of $s = 1$ and a zero-padding of $p = 0$. While we will not enter into technical details on how to backpropagate the gradients in the general case of any zero-padding and stride, we mention that proper zero-padding should be used for the backpropagation if we use a zero-padding of $p > 0$ for performing convolutions during forward propagation, as well as the gradient of the feature maps should be upsampled accordingly in the case of a stride of $s > 1$ (by filling the intermediate positions in which convolutions did not occur with zeros). For the case where the convolutional layers take as input multiple channels and produce as output multiple feature maps, the above formulas can easily be generalized by taking into consideration in the chain rule the contributions of multiple input channels and multiple produced feature maps wherever they occur.

Now, we will derive the gradients of pooling layers. No learning takes place in pooling layers; therefore the gradient from the layer above just passes through and gets propagated to the layer below and there is not update rule for pooling layers.

During forward propagation, $m \times m$ blocks are reduced to a single value (according to the pooling function), that is the value of the "winning unit". During backpropagation, this single value "winning unit" acquires an error computed from propagating back the error from the layer above. To keep track of the "winning unit" its index is noted during the forward pass and used for gradient routing during back-propagation. Gradient routing is done in the following ways:

- For max pooling, the derivative for $h^{(n)}_{ijk}$ is computed by using the chain rule and simply propagating back the error as:

$$\frac{\partial L}{\partial h^{(n)}_{ijk}} = \frac{\partial L}{\partial h^{(n+1)}_{ijk}} \frac{\partial h^{(n+1)}_{ijk}}{\partial h^{(n)}_{ijk}} = \frac{\partial L}{\partial h^{(n+1)}_{ijk}} \frac{\partial}{\partial h^{(n)}_{ijk}} \max_{p,q} h^{(n)}_{i,j+p,k+q}. \tag{3.24}$$

We set $h^{(n)}_{ijk} = h^{(n)}_{i,j+p',k+q'}$ and (3.24) becomes:

$$\frac{\partial L}{\partial h^{(n)}_{i,j+p',k+q'}} = \frac{\partial L}{\partial h^{(n+1)}_{ijk}} \frac{\partial}{\partial h^{(n)}_{i,j+p',k+q'}} \max_{p,q} h^{(n)}_{i,j+p,k+q}. \tag{3.25}$$

We can easily derive that:

$$\frac{\partial}{\partial h^{(n)}_{i,j+p',k+q'}} \max_{p,q} h^{(n)}_{i,j+p,k+q} = \begin{cases} 1 & \text{if } p', q' = \underset{p,q}{\text{argmax}}\, h^{(n)}_{i,j+p,k+q}, \\ 0 & \text{elsewhere} \end{cases}.$$

Thus:

$$\frac{\partial L}{\partial h^{(n)}_{i,j+p',k+q'}} = \begin{cases} \frac{\partial L}{\partial h^{(n+1)}_{ijk}} & \text{if } p', q' = \underset{p,q}{\text{argmax}}\, h^{(n)}_{i,j+p,k+q}, \\ 0 & \text{elsewhere} \end{cases}. \tag{3.26}$$

Similarly, in gradient form:

$$\nabla_{h^{(n)}_{i,j+p',k+q'}} L = \begin{cases} \nabla_{h^{(n+1)}_{ijk}} L & \text{if } p', q' = \underset{p,q}{\text{argmax}}\, h^{(n)}_{i,j+p,k+q}, \\ 0 & \text{elsewhere} \end{cases}. \tag{3.27}$$

In other words, the error is just assigned to where it comes from, i.e. the ”winning unit”, because all the other units in the previous layer’s pooling blocks did not contribute to the error; hence, they are assigned values of zero.

- For average pooling, the derivative for $h_{ijk}^{(n)}$ using the chain rule is:

$$\frac{\partial L}{\partial h_{ijk}^{(n)}} = \frac{\partial L}{\partial h_{ijk}^{(n+1)}} \frac{\partial h_{ijk}^{(n+1)}}{\partial h_{ijk}^{(n)}} = \frac{\partial L}{\partial h_{ijk}^{(n+1)}} \frac{\partial}{\partial h_{ijk}^{(n)}} \frac{1}{m^2} \sum_{p,q} h_{i,j+p,k+q}^{(n)} = \frac{\partial L}{\partial h_{ijk}^{(n+1)}} \frac{1}{m^2}, \tag{3.28}$$

where

$$\frac{\partial}{\partial h_{ijk}^{(n)}} \frac{1}{m^2} \sum_{p,q} h_{i,j+p,k+q}^{(n)} = \frac{1}{m^2}.$$

From (3.28) we obtain a single value, yet we want to fill the whole pooling neighborhood with the gradient of the corresponding pooled unit. In max-pooling, we filled the pooled neighborhood with zeros everywhere except the ”winning unit” since the other units do not contribute to the gradient. In average pooling, each unit has the same contribution to the gradient; hence, each unit should receive the same gradient error. Thus, we upsample the error gradient by filling each pooling neighborhood with the gradient of the corresponding pooled unit. The gradient of an average pooling layer is:

$$\nabla_{\boldsymbol{h}^{(n)}} L = \frac{1}{m^2} \text{upsample}(\nabla_{\boldsymbol{h}^{(n+1)}} L), \tag{3.29}$$

where upsample$(\cdot)$ inverts the subsampling operation by filling each $m \times m$ neighborhood with the error of the corresponding pooled unit. As you can see, the error is multiplied by $\frac{1}{m^2}$ and assigned to the whole pooling block which means that all units in the pooling block get the same value.

3.7 Popular convolutional network models

Here, we will briefly describe some popular and successful convolutional network architectures. Most of them won ILSVRC [35] which is a very challenging image recognition and detection competition. ILSVRC provides a very big datataset, namely ImageNet, with approximately 1 million images and 1000 image categories. ImageNet played a crucial role in the history of ConvNets. Moreover, ILSRVC is a very good

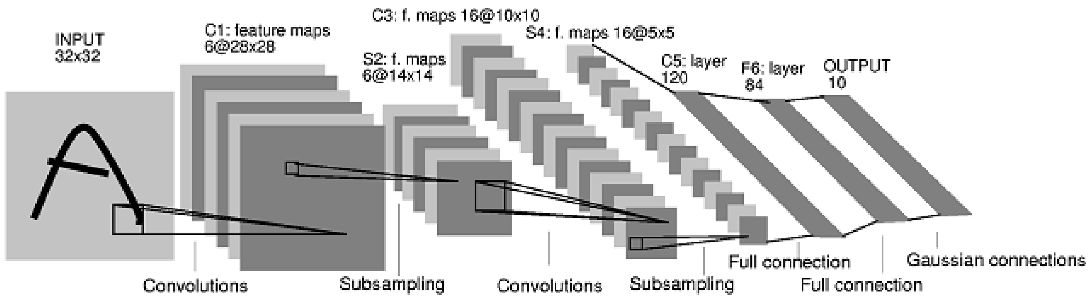


Figure 3.10: The pioneering convolutional architecture LeNet-5 [9] for handwritten digit recognition. Figure reproduced from [10].

benchmark for comparing classification and detection models, due to the difficulty of the given tasks, the variabilty of the data in the dataset and the big number of classes/examples. Thus, winning an ILRSVC contest indicates a powerful model.

LeNet-5 is a pioneering convolutional network architecture by Yann LeCun et al. [9]. LeCun introduced the basic idea of convolutional networks previously in [31], without calling them convolutional networks at that time. Later in [9], LeCun et al. introduced the notion of convolutional networks and established well their principles such as parameter sharing and local connectivity. Furthermore, they designed LeNet-5, the first well-known convolutional layer architecture. It consists of 3 convolutional layers, where only the first 2 are followed by pooling and subsampling, and then follows a fully-connected layer which is connected with the output layer. A graphical representation of LeNet-5 can be seen at Figure 3.10. The authors employed LeNet-5 on the task of handwritten character recognition, where they also created the *MNIST* dataset, a dataset that is used until today as a benchmark on image classification. In the experiments, they tested several different classifiers (Nearest Neighbors, SVMs, etc) and they stated that LeNet-5 was superior in performance over all the other classifiers. Moreover, they experimented with two more shallow ConvNets, where the performance was downgraded comparing to LeNet-5, still comparable and most of the times superior with the rest of the classifiers though. This findings suggested that ConvNets are suitable models for image classification, and that deeper networks can perform better.

AlexNet was the first work that popularized convolutional networks in computer vision, developed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton [11]. The AlexNet was submitted to the ImageNet ILSVRC challenge in 2012 and significantly outperformed the second runner-up (top 5 error of 16% compared to the runner-up whose error was 26%).

In their work, they proposed several novelties concerning design strategies and the training of their architecture. First of all they proposed the use of ReLUs than classical sigmoids or tanh, which helped the network to be trained several times faster. Furthermore, they proposed a new type of layer which can been added after one ore more convolutional layers. It is named *local respone normalization* layer, and it normalizes the activation of a hidden unit at a given position (i, j) in a feature map by taking into consideration the activations of hidden units at the same position (i, j) at neighboring feature maps. Following their notation, if $a^i_{x,y}$ is the activation of the i-th feature map at position (x, y), then the response-normalized activity $b^i_{x,y}$ is given by the expression:

$$b^i_{x,y} = a^i_{x,y} / (k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N,i+n/2)} (a^j_{x,y})^2)^\beta \tag{3.30}$$

where the sum runs over n "adjacent" kernel maps at the same spatial position, and N is the total number of kernels in the layer. The constants k, n, α, and β are hyper-parameters whose values are determined using a validation set. A similar normalization layer was proposed in [18]. Here a feature map activation is normalized in two different ways, across adjacent elements of the same feature map or at the same spatial location across different feature maps. The normalization takes the form of subtracting and dividing with locally computed means and standard deviations respectively. The idea of these normalization schemes is to enforce a sort of local competition between adjacent features in a feature map, and between features at the same spatial location in different feature maps. Nevertheless, these layers didn't have a drastical impact on performance, and thus they have been rejected in more recent architectures.

In AlexNet, max-pooling with overlapping neighborhoods is used, opposite to most architectures, and the authors report reduction in error rates. They employed a parallelization scheme to spread the network across two GPUs for faster training, by splitting the kernels of each layer across two diferrent GPUs. The model is depicted in figure 3.11. The network consists of 5 convolutional layers, where the first two are followed by max-pooling and the following three are stacked on top of each other with max-pooling after the last one. This is the first work that stacks multiple convolutional layers on top of each other, before it was common to place a max-pooling layer after each convolutional layer. Lastly, they utilized some more techniques regarding the

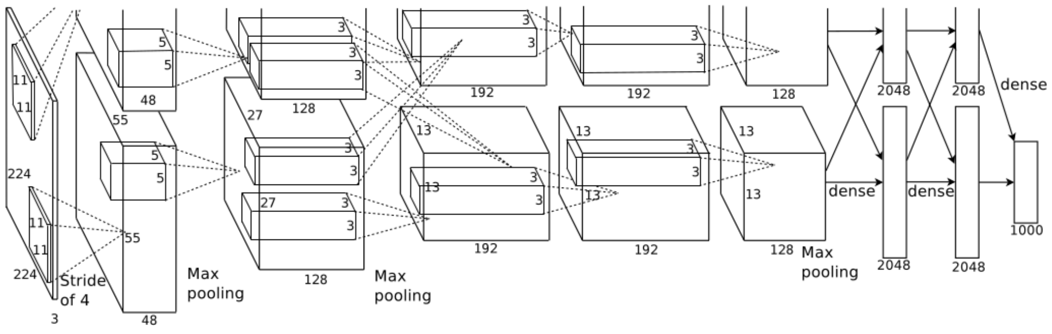


Figure 3.11: AlexNet architecture. The network is splitted in two parts, where each part contains half of the feature maps of each layer. In this way the network is parallelized in two GPUs. Note that all convolutional layers are connected only with the part of the network that is in the same GPU while the third convolutional layer is connected with both parts of the second convolutional layer. Figure reproduced from [11].

training of their architecture which we 'll see in the next chapter where we talk about training.

The ILSVRC 2014 winner was a convolutional network named **GoogleNet** from Szegedy et al. [12] from Google.

The main contribution in this architecture is the *inception module*, which replaced simple convolutional layers in the network, except the first layers where normal convolutional layers were placed. The motivation for the inception module is based on the idea of finding an efficient way to approximate an optimal local sparse structure. Optimal local sparse structure, refers to the optimal way of connecting the hidden units in a feature map, locally with the inputs of the layer before. In other words, you can see it as the optimal local connectivity of layers which we analyzed in section 3.3. Convolution, proposes a local structure but might not be the optimal. The basic idea of the inception module is to combine multiple convolutional layers to approximate an optimal local structure. To this end, they firstly developed, a naive version of the inception module which is depicted in Figure 3.12a. As it can be seen, they used three convolutional layers in parallel, which are connected to the previous layer, where the first performs 1×1 convolutions, the second 3×3 and the third 5×5. Moreover, they added also a max-pooling layer for additional beneficial effect in performance. Note, that after this operations, the outputs from each different building block are merged in a single output vector to form the input for the next layer. They observed though, that the aforementioned module, could be prohibitely computationally expensive when the number of filters in the previous layer is very large. This led to the development of a sophisticated version of the inception module, where they applied dimensionality reduction before convolutions and max-pooling, that is, they apply 1×1 convolutions which result in a smaller number of feature maps. The second version of the inception module is illustrated in Figure 3.12b. The benefits of this module are the following:

- It reduces the computational requirements of the model.
- The incorporation of 1×1 convolutions introduces multiple levels of non-linearity which may result in more powerful feature representations.
- It processes the information at various scales and then aggregates it so that the next stage can abstract features from different scales simultaneously, which may result in superior performance.

Instead of fully-connected layers at the last part of the model, GoogleNet applies average pooling, which firstly reduces further the number of parameters of the model and secondly the authors state increased performance. The overall architcture is a 22 layer architecture, with the first layers to be classical convolutional layers, the following layers are layers based on the inception module where some of them are followed by max-pooling and finally follows average pooling with a fully-connected layer as output. Finally, as this architecture is very deep, finding an efficient way to propagate the gradients through the whole network is a major problem. To this end, they added auxiliary classifiers to the architecture, which take the form of small convolutional networks placed on top of the output of specific inception modules. Their loss gets added to the total loss of the network with a discount weight and this increases the gradient signal that gets propagated back. You can see a depiction of GoogleNet architecture in figure 3.13.

To sum up, the hallmark of GoogleNet architecture is the improved use of computational resources with the utilization of the inception module which allows for increasing both the width of each layer as well as the number of layers without getting into computational difficulties.

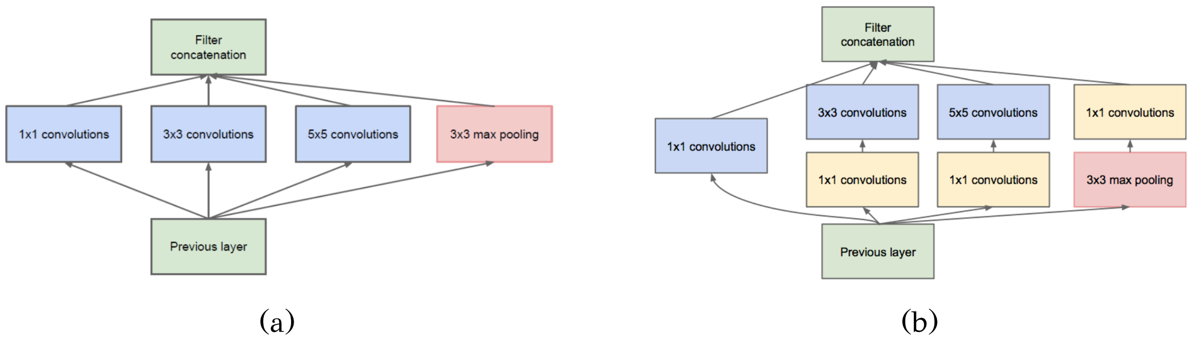


Figure 3.12: Inception module. (a) The naive form of inception module. (b) The inception module with dimensionality reductions in the form of 1×1 convolutions. Figures reproduced from [12].

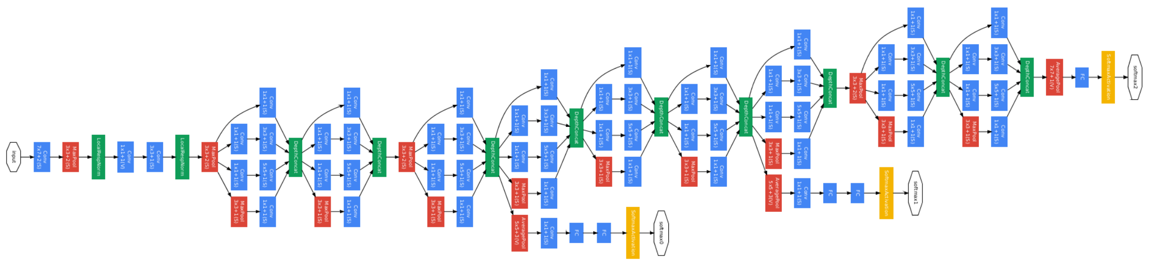


Figure 3.13: GoogleNet architecture. Figure reproduced from [12].

In ILSVRC 2014, **VGG-Net** from Karen Simonyan and Andrew Zisserman [36], won the first and the second position in localization and classification tracks respectively. Their main contribution is a thorough evaluation of networks of increasing depth. To this end, they constructed multiple architectures with increasing depth from 11 to 19 layers including convolutional and fully-connected layers. The key novelty of the VGG-Net is the use of convolutional layers with small receptive fields of size 3×3, stride 1 and pooling 1, and max-pooling layers with 2×2 receptive fields and a stride of 2 across the whole network. This is a minimalistic and very homogeneous architecture which has all the nice properties we described in Section 3.5.3. Furthermore, multiple convolutional layers are stacked on top of each other, which as we described in Section 3.5.3 diminishes the need for large receptive fields. They were the first that constructed such a homogeneous architecture, while previously different kernel sizes and stride/padding sizes were used across different layers of the network. They showed that this architecture outperforms several other previous

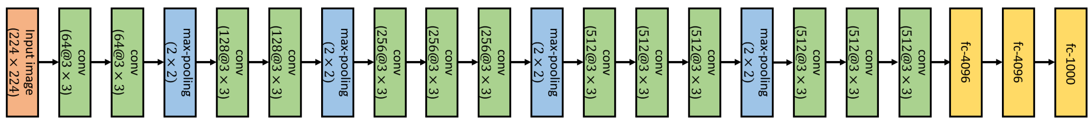


Figure 3.14: VGG-Net architecture. Different colors are used to discriminate between the input layer, convolutional layers, max-pooling layers and fully-connected layers. The architecture consists of 13 convolutional layers, 4 max-pooling layers and 3 fully-connected layers from which the last is the output layer. Convolutional layers are stacked on top of each other before a max-pooling layer. Stacks of convolutional layers are called convoutional groups. The number of feature maps in successive convolutional groups is doubled, where the first convolutional group has 64 feature maps and the last convolutional group has 512 feature maps. The hidden fully-connected layers have 4096 hidden units while the output layer has 1000 output units.

successful architectures. Very importantly, a very deep VGG-Net (16 trainable layers) yielded the best performance, which suggests that the increased ConvNet depth is beneficial in terms of classification accuracy. A graphical illustration of their best performing ConvNet architecture can be seen in figure 3.14.

3.8 Dataset augmentation

The best way to reduce overfitting and increase the generalization performance of a machine learning model is to train it on more data. In practice, the amount of available data is limited. A solution is to create artificial data instances and add it to the training set. This is particularly useful for deep convolutional networks and generally deep neural models which they are more prone to overfitting.

The most common way of creating new instances of data is called *dataset augmentation*, where new examples are generated by randomly transforming the original dataset examples. Dataset augmentation in images is pretty straightforward, where they exhibit geometric and photometric transformations. The goal is to make our model invariant to these transformations. Hence, the original images are transformed by random geometric transformations such as translation and rotation and by random color perturbations. In classification, dataset augmentation is a label preserving process, that is, while the original images are transformed, their corresponding labels are unchanged, such that, the model learns invariant mappings from the images to the corresponding labels. In regression, in some cases the target values should be changed correspondingly with the image transformation. For example, in hand pose estimation, if the model regresses for joint locations, a geometric image transformation should also be applied to the target values, such that, in the transformed image the transformed target values match the new joint locations.

Most commonly, data augmentation is applied in an online scheme with mini-batch stochastic gradient descent. For each mini-batch of images, random transformations are applied to the whole mini-batch and the model is trained on the transformed images. In such a way, in each epoch the algorithm observes different versions of the original images and the generalization performance of the model is improved since it observes a bigger amount of data. Data augmentation is especially useful when training on small datasets.

The most common image transformations for dataset augmentation are the following:

- **Horizontal flips**: During training, flip the image horizontally with probability 0.5. Vertical flips are not common, since the vertically flipped version of objects is not common in nature. During testing, either the predictions are based on the original image or they result from the average of the original image and its flipped version.
- **Random crops/scales**: Resize the training image with random scale and subsequently sample a random patch and crop the image. In such way, we obtain zoomed versions of the original image and the random crops are similar to random image translations but in a computationally cheaper way. During testing, the images are scaled using multiple scales, and for each scale the images are cropped using multiple crops that span the image space. These crops and scales are identical across the test set. The predictions are obtained as an average of the predictions of the multiple transformed versions of the image.
- **Color jitter**: There are two ways of jittering the color values of images:
    1. Randomly contrast jittering: Add small Gaussian noise to the pixels.
    2. PCA jittering [11]: Before training, PCA is applied on the RGB vectors of the pixels of the training set. During training, at each training image, add

multiples of the found principal components, with magnitudes proportional to the corresponding eigenvalues times a random variable drawn from a Gaussian distribution with zero mean and standard deviation 0.1.

In each case, we observe that during training the images are transformed randomly, while during testing a deterministic procedure is followed, where either the predictions are averaged on multiple fixed image transformations, or the transformations are not applied at all. This follows a general regularization scheme where random noise is added to the training procedure and during testing the noise is marginalized out. Similarly, in dropout, during training we sample random binary masks for the hidden units in each iteration while during testing we leave all hidden units active.

Of course, more transformations can be applied in the training images such as translation, rotation and shearing, but they are less common since they are more computationally expensive.

3.9 Pretrained models

Deep ConvNets are preferred over shallow ones since they learn more expressive and highly discriminant features. Yet, the limited amount of training data prevents the employment of large ConvNets due to overfitting concerns. Since it is rare to have a dataset of sufficient size to train a large ConvNet, it is usual to pretrain a ConvNet on a very large dataset (e.g. ImageNet [35]) and subsequently use the pretrained model for the task of interest. This falls in the category of transfer learning where a learned model in a task is used for the task of interest. There are two major scenarios of transfer learning with pre-trained ConvNets.

In the first scenario, the pretrained ConvNet is used as a fixed feature extractor. The output layer is removed and features for the images of the new dataset are extracted from the last hidden layer of the ConvNet. It is important to extract the features after the activation functions of the hidden units since the network is trained to classify images based on the values of the activations functions. Once features are extracted for each image from the pretrained ConvNet, a linear classifier such as a simple softmax fully-connected layer or a linear SVM is trained on these ConvNet based image features. This scenario is particularly convenient when the dataset is small where training the ConvNet would lead to overfitting.

In the second scenario, the pretrained weights are used as an initialization of the weights of the ConvNet. The original output layer is replaced with an output layer suitable for the task of interest (e.g. for classification the number of the output units would be equal to the number of classes of the new task). Then, the weights of the network are *fine tuned* by applying optimization on the new dataset. It is possible to train only part of the network while keeping the weights of the rest of the network unchanged. The motivation for that, is that ConvNets learn hierarchical features and the shallow layers learn more generic features (e.g. edge detectors or color blob detectors) that should be useful to many tasks, but deeper layers of the ConvNet becomes progressively more specific to the details of the classes contained in the original dataset. Hence, it is typical to keep the first layers of the ConvNet fixed and train the later layers. The amount of layers that will be fine tuned depends on the amount of training data due to overfitting concerns. In the case of a medium size dataset, it is typical to train the fully-connected layers or even some convolutional layers too. In the case of a large dataset, more convolutional layers can be fine tuned or even the whole network. Usually, the learning rate during fine tuning is smaller than the initial learning rate since we expect that the pretrained weights are already good and we do not wish to change them fast and much.

Since, the training of deep ConvNets on large datasets can be very time consuming, it is common that researchers release their pretrained models (e.g. VGG [36] trained on ImageNet [35]). These models are used by the community for fine tuning on new tasks. The limitation is that in order to employ the pretrained weights, someone is restricted to use the corresponding model. Since successive layers are connected to each other, the structure of the network cannot be changed by removing convolutional layers or reducing the number of kernels in convolutional layers. Nevertheless, the utilization of pretrained models is very common as it has proven beneficial for the performance of ConvNets.

Chapter 4

Hand pose estimation with convolutional networks using RGB-D data

4.1 Introduction

Articulated pose estimation refers to the problem of estimating all the kinematic parameters of the skeleton of an articulated object, expressed in joint angles or joint positions. The problem of pose estimation of 3D articulated objects such as human body and hand has aroused a lot of attention in the computer vision community for long, as their solution can provide support to several important applications such as human computer interaction (HCI), augmented reality, gesture recognition, robots learning by demonstration and gaming.

In this work we study the problem of 3D hand pose estimation. Articulated hand pose estimation shares a lot of similarities with the popular 3D body pose estimation.

Despite their similarities, proven approaches in body pose estimation [37] cannot be repurposed directly to hand articulations, due to the unique challenges of the task. The human hand has far more complex articulations; hence, self occlusions between joints are prevalent which make the pose prediction a very challenging task. The projected image of a human hand is much smaller than that of a human body which results in low resolution hand images. Moreover, hand motion exhibits much larger variations in both camera viewpoints and finger articulations. This is because a human body can be assumed to be upright but a hand can take any orientation; the hand has much more meaningful configurations.

The literature is classified in two major categories: discriminative methods (appearance-based or data-driven) and generative methods (model-based). In model-based approaches [38,39,40,41,42], hypotheses are generated from a 3D hand model and poses are tracked by fitting the model to the test data using optimization. While these model-based approaches inherently deal with the kinematic constraints, joint articulations and viewpoint changes, their performance heavily relies upon accurate pose initialization and structural correlation between the synthetic model and testing subject (i.e. hand width and height). Furthermore, generative methods, rely upon accurate initialization, and at each frame the pose is initialized from the previous pose estimation which can lead to error accumulation and drift from the ground truth pose. Lastly, these methods require a hand-crafted carefully designed cost function to avoid local minima and a proper optimization method.

Discriminative methods [13, 14, 15, 37, 43, 44, 45, 46, 47, 48] deploy a classifier or regressor and use labeled datasets to learn a mapping from image features to hand poses. Data-driven approaches are more advantageous as they do not require complex model calibration and are robust to poor initialization. Moreover, discriminative methods have state-of-the-art performance in human body and hand pose estimation. Therefore, we are focusing on discriminative methods for hand pose estimation.

In the past few years, with the popularization of consumer depth sensors, human body pose estimation [37,45] have seen rapid progress and significant success. Depth images provide 3D information which is directly correlated with the task of 3D pose estimation, and hence the regressor or classifier can learn easier mappings from depth images to 3D poses, which results in increased performance. Since the widespread success of real-time human body pose estimation, the area of hand pose estimation has received much attention within the computer vision community.

The first successful data-driven approaches to human body pose estimation are based on random forests [37] and depth images, in which a random forest classifies pixels to body parts and a subsequent step estimates the position of the joints in each respective part. This combination of depth images with random forests that employ very simple depth features, enabled real-time performance and achieved state-of-the-art results in human pose estimation.

This paradigm has been applied to hand pose estimation [44] but with less success. That is, because the body is mostly near-frontal and there is less occlusion between limbs. However, hand motion exhibits much larger variations in both camera viewpoints and finger articulations. This can lead to occluded parts, and consequently to wrong labeling of hand parts from the classifier. As a result the subsequent step of joint position inference in each hand part is affected by the classification errors and the estimation accuracy is limited. Furthermore, in this scheme the predictions on each joint are independent, and thus the estimator does not consider the dependency between joints which may result in poses that violate kinematic constraints.

The aforementioned problems that arise from classification methods applied in hand pose estimation are tackled by the regression based approaches [13, 14, 15, 43, 45,46,47,48]. This approaches incorporate a regressor which directly learns mappings from depth images to 3D hand poses. Such methods are more principled since their learning is directly guided by the task. Regression based methods do not rely on a subsequent estimation step, they learn a direct mapping from a depth image to a 3D hand pose; hence, they can mitigate the problem of self-occlusion and recover the hand pose from occluded joints since they can learn meaningful mappings from occluded joints to the ground truth pose.

Regression forests were applied extensively for hand pose estimation [45,46,47,48], and have shown state-of-the-art performance. The authors usually propose a variant of random forests particularly designed taking into consideration the hand topology. With the emergence of deep learning most recent hand pose estimation methods turned their attention towards convolutional networks [13, 14, 15, 16, 43] where they outperform previous state-of-the-art regression forest based methods.

We propose a convolutional network based approach for 3D hand pose estimation. Since the structure of convolutional networks plays a crucial role in their performance, in the first part of our proposed methodology, we design and evaluate several different convolutional network architectures by alternating the depth of the network as well as other parameters that change the structure of the network. The depth of a network plays a crucial role in its performance. The amount of pooling is also essential since redundant pooling can lead to information loss and decreased performance. Having these and other important design considerations in mind, we carefully designed each one of our evaluated architectures in order to investigate different cases and find the one that performs best for the problem of 3D hand pose estimation. Our experimental analysis shows that our deepest convolutional network outperforms all the other considered architectures and provide state-of-the-art performance. Previous convolutional network based methods employ much shallower convolutional networks with reduced capacity. It turns out that hand pose estimation is a problem of high complexity and requires high capacity models, such that the network can learn good feature representations and have good generalization performance in new 3D poses. This is the main reason that our large convolutional network outperforms the state-of-the-art. This convolutional network is our proposed architecture that we use in the rest of our experiments.

Most methods use single depth images for 3D hand pose estimation. Depth images are noisy with quantization errors that result in missing parts around the hand boundaries, yet they provide useful 3D information. RGB images provide an accurate description of objects with color and texture information, but they lack of 3D infromation. We conjecture that the combination of RGB and depth images can improve the performance of convolutional networks. Based on these observations, in the second part of our methodology we propose fusion methods of RGB and depth information using convolutional networks. We propose three different approaches, input fusion, score level fusion and double-stream architecture fusion. Input level fusion aggregates RGB-D data and trains a convolutional network with images that contain both RGB and depth channels, while score level fusion trains two different convolutional networks with RGB and depth images respectively and fuses their predictions. Finally, double-stream fusion architecture, is based on training two separate convolutional networks in parallel and fuse their feature maps at an intermediate layer using fusion functions. From our experiments we conclude that fusion of RGB and depth information do not leverage further useful information towards more accurate 3D hand pose estimation. The fused nets perform quite comparably with the state-of-the-art, but still our proposed convolutional network which is trained only with depth images outperform the fusion techniques and provide state-of-the-art performance.

4.2 Related work

The problem of hand pose estimation has been studied in the computer vision literature for decades. We refer the reader to [49] for a detailed survey of earlier hand pose estimation methods. Here we focus on more recent methods. After the popularization of low cost depth sensors, e.g. Kinect [50], 3D hand pose estimation as well as 3D body pose estimation have received much attention. More recent methods are classified into two different categories: the *generative (or model-based)* approaches and the *discriminative (or appearance-based)* approaches.

In generative hand pose estimation methods [38, 39, 40, 41, 42], hypotheses are made from a 3D articulated hand model e.g. a 3D hand mesh and poses are tracked by fitting the model to input image observations. Typically, an optimization method is responsible for searching in the parameter pose space and find the pose parameters that best explains the available observations. The objective function to be optimized measures the discrepancy between the visual cues that are expected due to a model hypothesis and the actual ones. The synthetic model is derived from well-known computer animation and graphics concepts. One downside of these methods is that they rely on very careful initialization, and hence they depend on previous pose estimation. In that case, output poses may drift away from the ground truth when the error accumulates over time. The objective function has to be designed very carefully in order to avoid local minima, or else proper optimization methods have to be incorporated that search the entire pose space. Oikonomidis et. al. [40] proposed the use of particle swarm optimization to track the hand pose in real-time from RGB-D images. De La Gorce et al. [41] incorporated shading and texture information into a model-based tracker. Ballan et al. [38] used color images and deployed a multi-camera setup to alleviate color ambiguitites. To this end, they used salient points like finger tips and they proposed an objective function that takes into account edges, optical flow and salient points. They formulated the optimization problem as a nonlinear least squares problem and they used the Levenberg-Marquard algorithm for local optimization. In [39], a novel optimization method is proposed, that effectively and efficiently explores the high-dimensional space of human hands. It is an evolutionary-based optimization method that deploys quasi-random sampling for more uniform coverage of the hand parameter space. Qian et. al. [42] introduced a very simple hand model that is approximated by a set of spheres and a very fast cost function that allowed real-time performance. Furthermore, they proposed a hybrid optimization algorithm which combines the Iterative Closest Points algorithm (ICP) and particle swarm optimization for faster convergence and better resistance to local optima.

Generative approaches are very accurate and can handle articulation and view-point changes. On the other hand, they are very computationally expensive methods as hypotheses are generated online and the optimization method searches for the best observed pose at each frame. Furthermore, in a generative method, joint angle constraints need to be imposed to the optimization method so that the estimations do not violate the allowed range of poses. Finally, they need very carefully designed objective functions, proper selection of optimization algorithm and they rely on initialization, which can lead to drift from the original pose from error that accumulates from previously tracked poses.

Discriminative methods [13, 14, 15, 16, 37, 43, 44, 45, 46, 47, 48, 51, 52] learn a mapping from visual features to target parameter space such as joint locations. Oppositely to generative methods, they do not need an explicit specification of joint angles constraints, motion constraints or a synthetic model as these informations are encoded in the training data. Typically, they use a regressor or a classifier to infer joint locations. Discriminative methods are single-frame methods, that is they estimate the pose at each frame independently of previous frames; hence they do not require initialization and they are more robust to previous errors.

With the advance of Kinect and other similar depth sensors, several discriminative methods have been developed and gave impressive results in the task of pose estimation. Shotton et. al. [37], tackled the problem of human pose estimation using an intermediate body part representation. A depth image is first segmented into body parts which are spatially centered to the skeletal joints. They treated the segmentation in body parts as pixel-wise classification where they trained a deep Randomized Decision Forest to classify pixels to body parts. For training, they generated a large realistic synthetic dataset with depth images of human bodies. Each tree in the forest takes pixel data as input, and after branching left or right based on the decision in the split nodes, pixels end up in the leaf nodes where probability distributions are constructed that classify pixels to body parts. At the splitting nodes, the comparison is based on:

$$f_\theta(I, \boldsymbol{x}) < \tau \tag{4.1}$$

where f_θ is a feature function on the input parameterized by θ, I is the input image, $\boldsymbol{x}$ is the pixel that is on the splitting node and τ is a threshold. τ and θ are learned during training. For the feature function they propose:

$$f_\theta(I, \boldsymbol{x}) = d_I(\boldsymbol{x} + \frac{\boldsymbol{u}}{d_I(\boldsymbol{x})}) - d_I(\boldsymbol{x} + \frac{\boldsymbol{v}}{d_I(\boldsymbol{x})}), \tag{4.2}$$

where $d_I(\boldsymbol{x})$ is the depth of image I at pixel location $\boldsymbol{x}$ and the parameters $\theta = (\boldsymbol{u}, \boldsymbol{v})$ are image offsets. These features that simply compare the depth between offsets of the current pixel have shown very good performance at pose estimation tasks using depth images. The normalization with $\frac{1}{d_I(\boldsymbol{x})}$ ensures depth invariance such that at a given body point the length of the offsets scales with the depth of the point. Now that the body is segmented into parts, the mean shift algorithm is used at each part to find local modes in the distribution of the part, i.e. to estimate the centers of each part which correspond to a skeletal joint.

In [45], a similar approach with [37] is followed. They used a regression forest where they employed the same test features at the split nodes like [37]. The difference is that now pixels do not vote for body parts but for offsets from the joints. Votes are accumulated to the leaf nodes where distributions are formed that represent the relative 3D offset for each body joint. Thus, now pixels vote for their joint offsets and the forest directly produces continuous outputs of the estimated joint locations.

A big part of the hand pose estimation literature based on discriminative approaches were inspired by [37, 45] and used RDFs or regression forests in combination with depth images. In [51], the idea of [37] was adopted and trained an RDF to classify pixels to hand parts and finally infer the 3D hand pose estimation using the mean shift algorithm. In [44], the idea of Shape Classification Forests (SCF) was introduced. They first cluster the dataset, where each cluster represents a different hand shape. Then they train the SCF to infer the cluster for each pixel where a separate pose estimator is trained on each cluster forming a network of experts. In [52], Hough regression forests were used. In the first step a Hough forest provides an initial estimation of the hand orientation and 3D location. In the second step the parameters of the orientation of the hand from the first step are used to form a new Hough Forest which with the use of hand orientation parameters transforms the classical pixel features used in [37] and provides features invariant to rotations. This second Hough Forest produces a set of candidate poses where a final optimization step selects the best among the candidates. Tang et al. [48] introduced the Semi-supervised Transductive Regression forest, which learns associations between partially labeled realistic data and fully labeled synthetic data and leverages merits from both domains. The forest handles viewpoint changes along with poses by using a hierarchical classification scheme, where in the first step it performs viewpoint classification, it follows a classification step of individual joints and in the final step the forest regresses the joint location and infer the pose. In [47], the hand topology is learned in an unsupervised manner by using a Latent Tree Model. Then the authors introduced the Latent Regression Forest, which given the learned Latent Tree Model, it trains binary classification trees which iteratively divide the image in two sub-regions until each sub-region corresponds to a skeletal hand joint. Sun et. al. [46] proposed a cascaded hand pose regression scheme, where it deploys a sequence of weak regressors that are regression forests and progressively each weak regressor is parameterized with the pose estimation of the previous regressor, and learns the residual error. With this procedure the pose estimations are updated iteratively by the new estimations of the weak regressors. Furthermore they further improved their approach by introducing a hierarchical approach which regresses the pose of different parts sequentially in the order of their articulation complexity, such that easier parts are first estimated such as the palm, and then conditioned on this estimation more complex joints are estimated such as the fingers.

The widespread use of convolutional networks and their outstanding performance in several computer vision tasks influenced also the hand pose estimation literature. Tompson et al. [13] were the first that employed convolutional networks for hand pose estimation. They trained a convolutional network to predict 2D heat-maps for each joint location where the intensity of the heat map represents the probability that the joint is present in that location. Afterwards, they used an inverse kinematics optimization algorithm to recover the 3D pose of the hand from the 2D heat-maps. The drawback is that their inverse kinematic procedure relies on the depth map for the inference of the depth, which may result in large errors under occlusions of joints. They stated, that they used the intermediate heat-map representation because it reduces the complexity of the learning problem; it is very difficult for the network to directly map depth images to poses. The input image is downsampled twice and produces a multi-resolution image pyramid. Then, a multi-resolution convolutional network is deployed and benefits from both local and global features depending on the scale. Each image at a different scale is fed as input to a convolutional module, with two layers of convolutions followed by max-pooling. Finally, the feature maps from the three modules are concatenated in a vector and they are connected two a neural network with two fully-connected layers and at the output of the network predicts the heat-maps.

Oberweger et al. [14] first considered several different convolutional network architectures to examine which one performs best for the problem of hand pose estimation. Subsequently, they used a refinement step that for refining the positions of the joints. They first examined two classical architectures, a shallow network with one convolutional layer followed by max pooling and one fully-connected hidden layer, and a deeper network with three convolutional layers followed by max-pooling layers and finally two fully-connected layers. Similarly to [13], they investigated a multi-scale architecture where at each scale the image is downsampled. Their results showed that the multi-scale architecture performed better than the deep architecture which performed better than the shallow architecture.

Their last architecture was motivated by the idea that a low dimensional embedding is sufficient for parameterizing the hand pose. To this end, they wanted instead of predicting directly the hand pose to predict first the parameters of the hand in a lower dimensional space which will impose a prior on the physical constraints of the hand. For the prior they used a ”bottleneck” layer at the end of the network before the output layer with size smaller than the number of joints and this layer predicts the parameters of the hand in the low dimensional space. They initialized the weights of this layer with some of the principal components of PCA. In their experiments they showed that this network with the prior layer, performed best among the architectures they evaluated.

Finally, in [14], a pose refinement step was proposed, which used a multi-scale approach. They centered patches of different scales on each joint and they used an architecture similar to their multi-scale architecture described before, but now centered on each joint separately. For further improvement they iterated this refinement step more than once. Their results outperformed previous state-of-the-art.

Oberweger et al. [15] adopted a generative approach without using a hand model or an optimization method accompanied with a carefully designed cost function. Instead they learned to generate images from training data. Very interestingly, each component of this approach is a convolutional network. A typical convolutional network is employed as the predictor, which makes an initial pose estimation. The second part consists of the synthesizer. This network takes poses as inputs and learn to generate the corresponding depth images. It does so by successively performing convolutional and *unpooling* operations. Unpooling is the inverse operation of pooling, that is the feature map is expanded. Unpooling has been used by other researchers too, for image generation with convolutional networks [53, 54]. Finally, the updater network is an architecture with two parallel streams of convolutions and max-pooling operations that share weights and they are merged at the fully-connected layers of the network. Given a depth image, and a corresponding generated image from synthesizer for the current pose estimate, the updater learns to correct the errors from the initial pose estimate that are seamlessly integrated in the generated image from the synthesizer. Their method significantly outperformed previous state-of-the-art methods.

Zhou et al. [16] introduced a model based deep learning approach that considers the geometry of a hand model. A new layer was proposed that maps joint angles to joint locations. The layer is differentiable and can be trained along with the network in an end-to-end fashion. The layer implements a forward kinematic function that maps pose parameters to joints. Their network is similar to the baseline architecture of [14] (three convolutional layers followed by max-pooling and two fully-connected hidden layers and the output layer). The key difference is that after the second fully connected layer they introduced a new fully-connected layer that outputs the pose parameters which is connected with the hand model layer that uses the forward kinematic function to output 3D joint locations and take geometry into account which ensures that the output poses do not violate any physical constraint. Lastly, they added in the loss function a term that enforces physical constraints on the rotation angle range. Their approach had performance comparable to state-of-the-art.

In [43], a multi-view CNN approach was used, which can better exploit depth cues and recover 3D information of hand joints without the need for model fitting. Firstly, the point cloud of the input depth image is projected onto three orthogonal planes and then each view is fed in a separate convolutional network to generate a set of heat-maps. They used the same convolutional network as in [13]. As the heat-map in each view encodes the 2D distribution of joint locations, the combination from all views contains information about the 3D location of hand joint. By fusing heat-maps from each of the three views they finally obtain the 3D joint location. The approach of [43] has several advantages over [13]. Firstly, in [13] the depth joint location is inferred by the depth value at the estimated 2D heat-map position, which may result in large errors even with small deviation from the true 2D joint location. On the contrary, in [43] heat-maps in multiple viewpoints are estimated, from which the 3D hand pose can be estimated more robustly. Moreover, differently to [13], multi-view CNN do not require a hand model for fitting the pose with optimization, but they learn inherently hand kinematic constraints from data.

Convolutional networks have shown outstanding performance in the task of hand pose estimation [14,15,43] where the ConvNet based methods outperformed the previous state-of-the-art random forest based methods. In [55], a thorough review for depth-based hand pose estimation was presented, which we strongly recommend the interested reader to study. In their analysis, Supančič et. al compared deep models, random forests, deformable part models and a nearest neighbor method which they introduced for hand pose estimation. Their results showed clearly that deep models outperform all the other methods in the problem of articulated hand pose estimation. As deep models have proven to be the best estimators in the task, we are turning our attention towards developing a ConvNet based method for addressing the problem of 3D hand pose estimation.

Our first major concern is to find which architecture performs best for the problem of hand pose estimation. To this end, we evaluate several different architectures, by alternating the depth of the network as well as the size and the number of the kernels in the convolutional layers.

Very few works in the literature exploit the use of both RGB and depth information for hand pose estimation. Oikonomidis et al. [40] incorporated both RGB and depth data, by adding terms in the objective function that measure the discrepancy between observed color and depth hand images and the color and depth hand images that are rendered for a given hand hypothesis. The same authors followed a similar approach in [56] but they applied pose estimation on two hand interacting with each other. Sridhar et. al [57] proposed a hybrid approach that combines a generative and a discriminative hand pose estimator. The generative hand pose estimator performs local optimization to find the hypothesis that best explains the observed RGB image. The discriminative pose estimator first detects finger tips and uses the finger tips along with finger databases to generate multiple pose hypotheses, and finally, the pose with the least discrepancy in observed and estimated fingertips is chosen. This pose is used then as an initialization for the local optimization. Apart from these methods, to our knowledge, there are not other approaches to hand pose estimation that combine RGB and depth cues.

We propose ConvNet based fusion techniques, that combine RGB and depth information in an end-to-end learning paradigm. Our intention is to investigate whether the fusion of RGB and depth information can leverage benefits of both domains and provide more accurate hand pose estimation. We propose three fusion techniques, input fusion, score level fusion, and double-stream architecture fusion. We compare these methods with our best performing deep architecture trained only with depth images. The ConvNet that is incorporated in each fusion technique has the same structure as our best performing architecture. While the examined fusion approaches have performance comparable to state-of-the-art, our proposed convolutional network trained only with depth images outperforms all our fusion methods.

4.3 Our approach

In this section, we describe our general approach as well as our contributions on the problem of hand pose estimation. We first give a formulation of the problem, and we describe a simple procedure for segmenting the hand from depth images given an annotated dataset of hand poses. Hand segmentation and data preprocessing are very crucial steps for hand pose estimation and their absence can significantly decrease the performance of a regressor.

Next, we describe our main methodology for designing and evaluating several different convolutional network architectures, for the problem of hand pose estimation from a depth image. At this point we consider only the problem of finding the optimal architecture for the problem of hand pose estimation, using a single convolutional network trained only with depth images.

Depth maps are very useful features for hand pose estimation as the 3D information they provide is strongly correlated with the problem. Yet, depth images are noisy with quantization errors, that result in missing parts around the boundaries, which can decrease the estimation accuracy. On the other hand RGB images provide an accurate description of the object with color and texture information. We build upon this observation, and we are pushing forward combining RGB and depth information in an end-to-end learning fashion using convolutional networks. To this end, we propose three different approaches for fusing RGB and depth information in convolutional networks.

In the first approach, the channels of RGB and depth images are fused, and the network learns to combine RGB and depth features at a very early stage. The second fusion technique we employ, is to fuse the predictions of two independently trained convolutional networks, each one trained with RGB and depth images respectively. Finally, we consider the case of fusing feature maps of double-stream architectures, were we employ different fusion functions.

We describe our benchmark dataset as well as the evaluation metrics we used for our experimental analysis. We provide details related to the training of the networks. Next, we make a self-comparison between all the architectures we considered for hand pose estimation from single depth images and show our best performing convolutional network were we discuss our findings. Next, we compare our fusion approaches between each other but also with our proposed convolutional network which is trained only with depth images. We investigate whether fusing RGB-D data further improves the performance of the convolutional network. Finally, we compare our best performing approach with the state-of-the-art.

4.3.1 Problem formulation and data preprocessing

The problem of 3D hand pose estimation consists of estimating the 3D kinematic parameters of the hand which can be expressed either in joint angles or in joint positions, given an image. We consider the case of estimating the 3D joint positions of a hand $\boldsymbol{J} = \{\boldsymbol{j}_i\}_{i=1}^{J}$, where $\boldsymbol{j}_i = (x_i, y_i, z_i)$ and J is the number of joints. We assume that we have an annotated dataset where for each image in our dataset we have $\boldsymbol{J}$. In this case, a specific hand model is not needed, as the model is determined from the dataset, and consists of the joints that are considered for each different dataset. Thus, the number of joints J varies among datasets. While most of the modern discriminative approaches estimate the 3D hand pose from a single depth image, we employ both RGB and depth images.

3D hand pose estimation ”in the wild”, i.e. from cluttered images where both the whole human body and the background are present, is a very complex task. To this end, most methods first employ a hand detection step, to simplify the estimation process. Hand detection in depth images can be achieved either by performing pixel-wise classification with random forests [13] or by assuming that the hand is the closest object to the camera and performing depth segmentation [42, 47].

In our case, the dataset we employ provides the position of the center of the hand, and we incorporate this information as the hand location, which eliminates the need for hand detection. We follow a similar preprocessing procedure as in previous work [14,16], assuming that the hand is already detected. A fixed-size cube is extracted from the raw depth image, centered at the center of mass of the hand. The cube is modeled by converting a patch from real world 3D coordinates to image coordinates, and using the depth values at that image patch as the third dimension of the cube. The conversion from real word 3D coordinates to image coordinates is accomplished by assuming the pinhole camera model, where the formulas are given by:

$$u = \frac{\phi_x x}{z} + \delta_x, \tag{4.3}$$

$$v = \frac{\phi_y y}{z} + \delta_y, \tag{4.4}$$

$$d = z, \tag{4.5}$$

where $[u, v, d]^\top$ is a vector in image coordinates, where given a depth image I the depth at position (u, v) is given by $I(u, v) = d$. Vector $[x, y, z]^\top$ corresponds to 3D real world coordinates, ϕ_x, ϕ_y are the focal length parameters in x and y direction respectively and δ_x, δ_y are the principal point coordinates. The cube is resized to a 128×128 image where the depth values are normalized to $[-1, 1]$ by using:

$$d_{norm} = \frac{d - I(cm_x, cm_y)}{cs/2}, \tag{4.6}$$

where d is the depth of the unnormalized image and d_{norm} is the normalized depth, $I(cm_x, cm_y)$ is the depth at the the center of mass of the hand with coordinates (cm_x, cm_y) in depth image I and cs is the size of the cube. Pixels in the resized and normalized image for which their depth is not available, which is very common to structured-light sensors, or pixels that correspond to the background, i.e points in the 3D space that lie behind the back face of the cube are assigned to a depth of 1.

The above process, translates the hand at the origin $(0, 0, 0)$ and normalize it to a hand that fits in a unit size cube. This process is very crucial for the performance of a convolutional network, as it provides depth and translation invariance in the sense that the predictions do not rely on the 3D position of the hand. In other words, the ConvNet learns depth and translation invariant features, such that the pose can be predicted correctly independently of the position of the hand.

It is essential that the 3D joint positions in the annotations are also normalized using the process described above for depth normalization. That is, for a given joint $\boldsymbol{j}_i = (x_i, y_i, z_i)$ we apply:

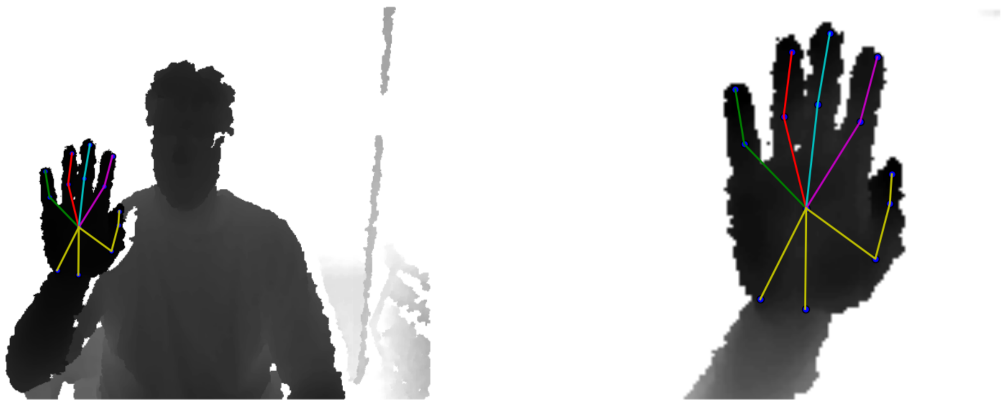
$$x'_i = \frac{x_i - cm_x}{cs/2}, \tag{4.7}$$

$$y'_i = \frac{y_i - cm_y}{cs/2}, \tag{4.8}$$

$$z'_i = \frac{z_i - I(cm_x, cm_y)}{cs/2}. \tag{4.9}$$

The 3D joint locations now become $\boldsymbol{j}'_i = (x'_i, y'_i, z'_i)$ and this process is applied for all joints. At test time, first the regressor estimates normalized joint locations and then the inverse transformations from (4.7), (4.8) and (4.9) are applied to provide the estimations in the original image locations. We train our models using as targets the 3D normalized joint locations then we apply the inverse transformations and finally we apply (4.3), (4.4) and (4.5) to obtain the joint locations in image coordinates. In figure 4.1 you can see a raw depth image with its corresponding hand pose as well as the same image after hand segmentation and depth normalization with the process described above. In the normalized image the joint locations are normalized as well.

For the RGB images segmentation we follow a very simple procedure. First we extract an 128×128 patch with a procedure similar to the one described above for the depth image. Then, given the segmented depth image, we construct a binary image with ones at the hand location and zeros everywhere else. We multiply this mask with the RGB patch and we obtain the segmented RGB image. The RGB images are normalized to follow a distribution with zero mean and standard deviation of one as:



(a) Unsegemented and unnormalized depth image.

(b) Depth image after segmentation of the hand.

Figure 4.1: Annotated depth images of hand pose. (a) Raw depth image with the annotated pose. (b) The image in (a) after hand segmentation and depth normalization. The image is from the NYU dataset [13].

$$R' = \frac{R - \mu_R}{\sigma_R}, \tag{4.10}$$

$$G' = \frac{G - \mu_G}{\sigma_G}, \tag{4.11}$$

$$B' = \frac{B - \mu_B}{\sigma_B}, \tag{4.12}$$

$$\tag{4.13}$$

where R, G, B are the initial pixel values for each color channel, R', G', B' are the corresponding normalized values, μ_R, μ_G, μ_B is the mean pixel intensity for each color channel and finally σ_R, σ_G, σ_B are the corresponding standard deviations. The mean as well the standard deviation are computed from the pixels of training images and they are used for the normalization of both training and testing images. A graphical depiction of a normalized depth image and its corresponding normalized RGB image can be seen in figure 4.2.

4.3.2 Designing convolutional networks for hand pose estimation

Now, we will describe the basic design principles we followed, to construct our ConvNet architectures for hand pose estimation. We will state each structural element that is part of the architectures, and we will outline all different ConvNet configurations

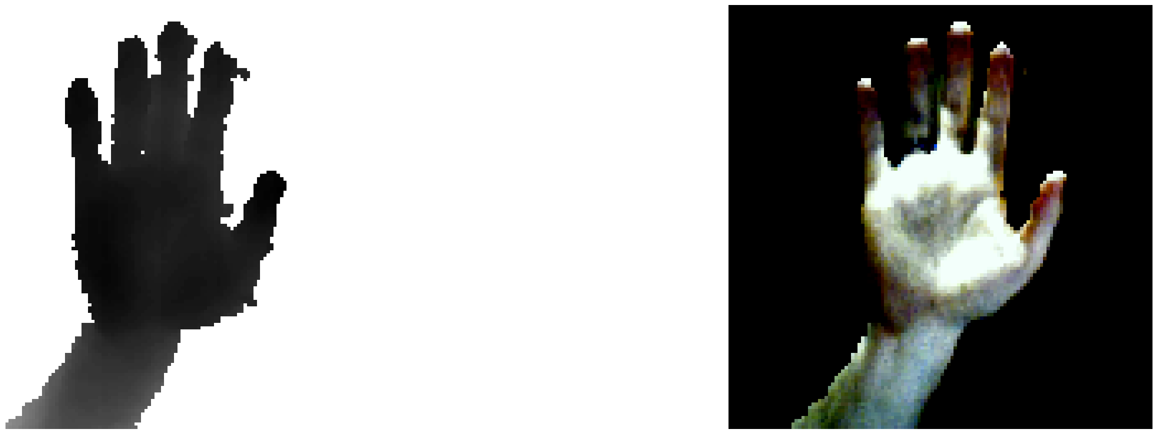

(a) Depth image (b) RGB image

Figure 4.2: Depth segmented and normalized image and its corresponding segmented and normalized RGB image. In this figure it is clear that while depth images are noisy, RGB images have a well-defined structure with several details that are absent from depth images. In the depth image you can see that the pointer, the middle and the ring fingers appear to be joined together while in the RGB image they are clearly separated. Furthermore, in the RGB image joint positions are visible while in the RGB image they are not.

we evaluated. Finally, we will show the final convolutional network we preferred for estimating 3D joint positions, which was selected based on our experiments. Here, we investigate the problem of designing convolutional networks for hand pose estimation from a single depth image. Subsequently, we will describe fusion techniques for RGB and depth information, we will use our best performing architecture that was trained with depth images also for the other networks that utilize RGB images.

Our architectures are inspired by the general design philosophy introduced in [36]. This is an extremely homogeneous architecture which deploys in all convolutional layers 3×3 kernels with stride 1 and zero-padding 1, and max-pooling layers with 2×2 receptive fields. We've discussed before about the advantages of such an architecture and in this work we will confirm from our experiments that is a beneficial architecture for hand pose estimation too.

All of our architectures apply ReLU nonlinearities for all convolutional and fully-connected (fc) hidden layers. Since the preprocessing we described in Section 4.3.1 normalizes the 3D joints positions in $[-1, 1]$, we employ hyperbolic tangent activation functions for the output units which estimate the 3D joints position. Although it is common to use a linear activation function for the output units in a regression setting, our early experiments shown improvements in the performance of our networks by using tanh activation functions. The networks training is performed by minimizing the mean squared error between the ground-truth 3D joints positions and the outputs of the network; hence the loss per image is:

$$L(\boldsymbol{f}(I;\boldsymbol{\theta}),\boldsymbol{y}) = \frac{1}{2J}\|\boldsymbol{f}(I;\boldsymbol{\theta}) - \boldsymbol{y}\|_2^2 = \frac{1}{2J}\sum_{i=1}^{3J}(f^{(i)}(I;\boldsymbol{\theta}) - y^{(i)})^2, \qquad (4.14)$$

where $\boldsymbol{f}(I;\boldsymbol{\theta})$ is a $3J$-dimensional vector of outputs from the ConvNet and J is the number of joints, $\boldsymbol{y}$ is the ground-truth vector which is formed by reshaping the ground-truth pose matrix we introduced in Section 4.3.1 denoted with $\boldsymbol{J}$, from a $3 \times J$ matrix to a $3J$-dimensional vector. Since different joints depend on each other, we use the same loss across all joints to enforce kinematic constraints, as the error for a joint depend on the error for other joints.

Similarly to [36], most of our ConvNet configurations stack 3×3 convolutional kernels with stride of 1 and zero-padding 1 such that convolutions preserve the spatial extent of feature maps. We've seen that stacking very small filters on top of each other is very effective as deeper layers have implicitly large receptive fields through the swallower layers and that using a stack of convolutional layers with small filters is preferable over a convolutional layer with a large receptive field. Nevertheless, we also experiment with an architecture that utilizes larger filters with varying size across different convolutional layers, with stride 1 and zero-padding accordingly to the size of the filter such that it preservers the spatial resolution.

All the ConvNet configurations we evaluated are presented in table 4.1 and table 4.2. In each table we consider different cases of configurations. Table 4.1 examines the case where all the nets have the same number of max-pooling layers that is 3, while table 4.2 considers architectures with bigger and varying number of max-pooling layers, specifically 4 and 5. While all these architectures are inspired by [36], they differ in the layout of the networks to be adjusted in our problem. We investigate three different design patterns of ConvNet configurations. In all design patterns we consider varying depth of the networks to investigate the importance of the depth on the performance.

Table 4.1: This table contains the first case of convolutional networks we considered. In this case all convolutional networks have the same number of pooling layers that is 3. We denote convolutional layers with ”conv$i$” which denotes the i-th convolutional layer. We denote pooling layers with ”max-pooling”. In convolutional layers, in parentheses we indicate the number of filters with the first number and the size of filters with the rest two numbers. Fully-connected layers are denoted with ”fc-4096” where 4096 is the number of hidden units. After the fc layers the output layer follows, that is 3J units, where J is the number of joints. Finally follows the loss layer.

Convolutional network architectures 1					
Net1	Net2	Net3	Net4	Net5	Net6
4 conv	5 conv	5 conv	5 conv	7 conv	9 conv
Input(depth image 128×128)					
conv1 (32x3x3)	**conv1** (32x3x3)	**conv1** (32x7x7)	**conv1** (64x3x3)	**conv1** (32x3x3) **conv2** (32x3x3)	**conv1** (32x3x3) **conv2** (32x3x3)
max-pooling					
conv2 (64x3x3)	**conv2** (64x3x3)	**conv2** (64x5x5)	**conv2** (128x3x3)	**conv3** (64x3x3) **conv4** (64x3x3)	**conv3** (64x3x3) **conv4** (64x3x3)
max-pooling					
conv3 (128x3x3) **conv4** (128x3x3)	**conv3** (128x3x3) **conv4** (128x3x3) **conv5** (128x3x3)	**conv3** (128x3x3) **conv4** (128x3x3) **conv5** (128x3x3)	**conv3** (256x3x3) **conv4** (256x3x3) **conv5** (256x3x3)	**conv5** (128x3x3) **conv6** (128x3x3) **conv7** (128x3x3)	**conv5** (128x3x3) **conv6** (128x3x3) **conv8** (128x3x3)
max-pooling					
					conv8 (128x3x3) **conv9** (128x3x3)
fc-4096					
fc-4096					
output-3J					
loss					

Table 4.2: This table contains the second case of convolutional networks we considered. In this case we consider more pooling layers than in the first case. Specifically, Net7 and Net9 have 4 pooling layers while Net8 has 5 pooling layers. We use green color in the lines that refer to the pooling layers to discriminate which networks are affected by the pooling operation. Specifically, all networks are affected by the first 4 pooling layers while only Net8 is affected by the 5-th pooling layer and that is why it is colored with green.

Convolutional network architectures 2		
Net7	Net8	Net9
6 conv / 4 pool	8 conv / 5 pool	8 conv / 4 pool
Input(depth image 128×128)	Input(depth image 128×128)	Input(depth image 128×128)
conv1 (32x3x3)	**conv1** (32x3x3)	**conv1** (32x3x3)
🟩 max-pooling	🟩 max-pooling	🟩 max-pooling
conv2 (64x3x3)	**conv2** (64x3x3)	**conv2** (64x3x3)
🟩 max-pooling	🟩 max-pooling	🟩 max-pooling
conv3 (128x3x3) **conv4** (128x3x3)	**conv3** (128x3x3) **conv4** (128x3x3)	**conv3** (128x3x3) **conv4** (128x3x3) **conv5** (128x3x3)
🟩 max-pooling	🟩 max-pooling	🟩 max-pooling
conv5 (128x3x3) **conv6** (128x3x3)	**conv5** (128x3x3) **conv6** (128x3x3)	**conv6** (128x3x3) **conv7** (128x3x3) **conv8** (128x3x3)
🟩 max-pooling	🟩 max-pooling	🟩 max-pooling
	conv7 (128x3x3) **conv8** (128x3x3)	
	🟩 max-pooling	

As it can be seen in the tables we name the configurations with Net1-Net9, and thus from now on we will refer to the nets with their names. In the tables, each net is a column in the table. Lines that are across all columns mean that this element is used by all configurations. All nets take an 128×128 depth image as input. In table 4.1 all configurations have 3 pooling layers and this is why the lines that describe the max-pooling operation are shared across all nets. On the other hand in table 4.2, Net8 has more pooling layers than Net7 and Net9 and that is why colors are used to better visualize which nets incorporate each pooling layer. After convolutional and max-pooling layers, all configurations incorporate two fc layers with 4096 hidden units each. In early experiments we evaluated architectures with smaller number of hidden units in fc layers which they have shown inferior performance. For regularization, we used dropout for each fc layer. We will talk more about selecting the dropout probability as well as its effect in performance in Section 4.4.3. Finally, each net has an output of size $3J$ with tanh activation functions to estimate the 3D joint positions.

In the first design pattern, each one of the first two convolutional layers are followed by max-pooling while the rest of the network stacks two convolutional layers on top of each other and then follows a max-pooling operation. Networks that fall in this category are Net1 in table 4.1 and Net7, Net8 in table 4.2. Our second design pattern, first incorporates two convolutional layers each one followed by max-pooling but then stacks three convolutional layers on top of each other before each max-pooling operation. Networks under this category are Net2, Net3, Net4 and Net9. The last design pattern we investigate is to stack both two and three convolutional layers at different depths of the network. Net5 and Net6 follow this design pattern. Net3 examines the case of bigger kernels with varying sizes across the network where conv1 and conv2 layers incorporate 7×7 and 5×5 kernels respectively and the rest of the network uses kernels of size 3×3. Lastly, with Net4 we evaluate the case of using bigger number of kernels where all convolutional layers have double number of kernels comparing to all the other nets.

Our experiments will show that the depth as well as the design pattern that is used for constructing the network plays a crucial role in the performance of the model. We will give details on the performance of each configuration as well as a discussion on why one architecture performs better over an other in Section 4.4.5. Our best performing configuration is Net6 which we call **Depth-Net** and is graphically depicted in figure 4.3. It has 9 convolutional layers. It utilizes a stack of two convolutional layers before the first two max-pooling layers, subsequently a stack of three convolutional layers before the third max-pooling layer and lastly a stack of two convolutional layers which they are not followed by max-pooling. We refer to stacks of convolutional layers as convolutional groups. The first convolutional group has 32 kernels on each layer, the second 64 and the third as well as the fourth 128. All convolutional kernels have size 3×3 with zero-padding 1 and stride 1.

To our knowledge, there is no prior work that evaluated such a deep network

Depth-Net

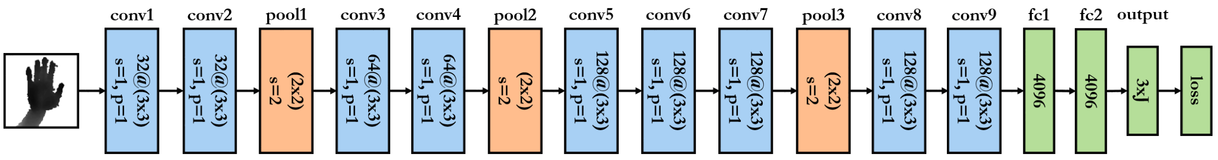


Figure 4.3: Depth-Net: Our best performing convolutional network. It is the deepest network among all different configurations we evaluated. In convolutional layers, the number before @ refers to the number of filters while the size of the filters is indicated in parentheses. The stride is denoted by s while zero-padding by p. In pooling layers, the size of the receptive field is indicated in parentheses and s refers to the stride. Different colors are used to discriminate between convolutional, max-pooling and fc layers.

for hand pose estimation. Our experimental results will show that our proposed architecture, Depth-Net, outperforms the state-of-the-art. In fact, we will show in the experimental analysis that we obtained state-of-the-art performance by directly estimating 3D joint positions from single depth images using Depth-Net, while other methods had inferior performance even by refining the joint locations [14] or by first estimating heat-maps of joint locations and then incorporating an inverse kinematics algorithm to infer the hand pose [13]. The key difference of Depth-Net with these approaches is that they used very shallow networks with limited generalization performance which prevents them from an accurate estimation of the hand pose even by employing an additional procedure for the estimation apart from the ConvNet.

4.3.3 RGB and depth fusion techniques with convolutional networks

Until now, we described a procedure for inferring 3D hand poses from a single depth image. As mentioned before, RGB and depth images have both some benefits and drawbacks. Depth images provide depth information which is strongly correlated with the task of 3D hand pose estimation and this benefits significantly the performance of regressors as it provides a more direct mapping from images to poses and the regressor requires less capacity to solve the task. Nevertheless, depth images are noisy with quantization errors and the description of the object that they provide is imprecise and much coarser than RGB images. RGB images can alleviate this problem as they provide much more precise description of the object. We have seen in figure 4.2 that in RGB hand images the 2D location of joints is much more distinct than in depth images, yet RGB images lack of depth information which make the problem of 3D pose estimation more complex.

We can say that the merits and drawbacks of RGB and depth images are complementary with each other. We want to investigate whether the combination of both sources of information can reveal the benefits of each and boost the performance of a regressor. To this end, we propose three different techniques, of fusing RGB and depth information. We call the first technique input fusion, where RGB and depth images are aggregated at the input level of a convolutional network, and the network learns to create correspondences between RGB and depth images at a very early stage. The second and third fusion techniques we propose are based on the idea of two-stream convolutional networks which was introduced in [58, 59]. In these works the authors suggested the use of two streams of convolutional networks for the problem of action recognition. A stream refers to a convolutional network. In the first stream they employed RGB images while in the second stream the corresponding optical flow frames where they showed significant improvement in the recognition of human actions. For our second and third fusion tehniques we employ double-stream architectures where the first stream is fed with RGB images while the second stream is fed with depth images. Our second fusion technique, which we call score level fusion, consists of training two separate convolutional networks and at the output layer fuse their predictions. Finally our last fusion technique which we call double-stream architecture fusion (or feature level fusion) is inspired by [59]. We consider the case of training a double-stream architecture and fuse the feature maps of the RGB and depth streams at an intermediate layer using several different feature map fusion functions. We employ fusion functions proposed in [59] and investigate their performance in hand pose estimation.

Input fusion

The idea of input fusion is very simple. First, RGB and depth images are aggregated into 4 input channels (3 channels from RGB and 1 from depth) to form RGB-D images. Then, a convolutional network is employed which has the same structure as Depth-Net, with the only difference that is alternated to take as input RGB-D images.

Now the network has 4 input-channels, which affects the first convolutional layer, in the sense that each one of its kernels has dimensionality $4 \times 3 \times 3$ instead of $1 \times 3 \times 3$ as in Depth-Net. A convolutional layer performs convolutions with different kernels for each input channel and then sum the results to obtain a feature map. This can be seen as a weighted sum of its input channels at several spatial locations.

Consequently, fusing RGB-D data which are fed in a convolutional network, has the effect that the first convolutional layer learns weighted combinations of RGB and depth channels. The weights of the kernels are learned in a way, such that at each spatial location the input channel that contributes more towards accurate estimation takes a higher weight. Hence, the weights serve as importance factors at all spatial locations across RGB and depth images. Lets say that we have an RGB and the corresponding depth image of a human hand and imagine that we are observing a joint location. It may happen that at that location the information for the third dimension of the position of the joint can be inferred from the depth image, while the 2D position of the joint is difficult to be estimated due to the coarse structure of the depth image. At the same time, the RGB image may have more precisely the 2D position of the joint yet lacks of depth information. In that case, the kernels of the first convolutional layer of the network are learned in a way such that weighted combinations of RGB and depth can form feature maps with blended information for the 3D joint position where the RGB channels contribute to the 2D spatial location of the joint and the depth channel contributes to the depth of the joint.

If this is the case, somebody may wonders why we do not train a convolutional network with RGB images to infer the 2D spatial location of joints and another convolutional network with depth images to infer the third coordinate of the joint 3D position. The answer is that it is better to leverage information for the 3D joints position from both RGB and depth images and let the training algorithm learn the best weighted combination from both information sources towards more accurate 3D hand pose estimation.

We call this architecture **RGBD-Net**, and it is depicted in figure 4.4. As you can see the architecture is identical to Depth-Net with the only difference that it takes fused RGB-D images in the input layer.

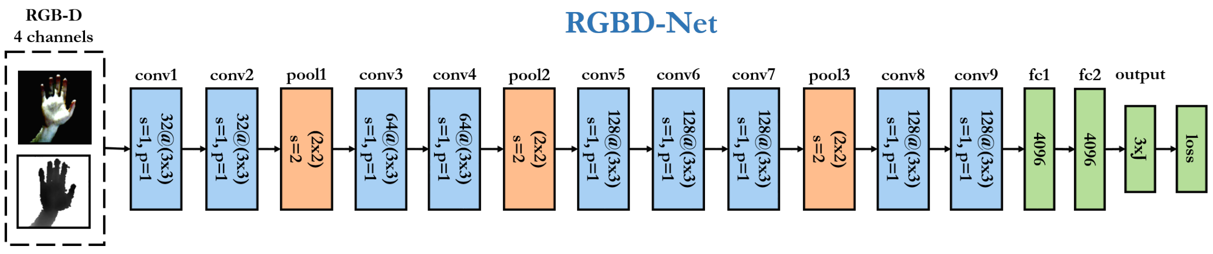


Figure 4.4: RGBD-Net: It takes fused RGB-D images at the input layer and infers the 3D hand pose at the output layer. The first convolutional layer learns weighted combinations of both RGB and depth images and exploits useful information from both domains towards more accurate 3D hand pose estimation.

Score level fusion

Score level fusion refers to the combination of the predictions of different models. We consider two convolutional networks as the models that their predictions are to be fused, the Depth-Net and we introduce a new network identical to the Depth-Net with the difference that it takes as inputs RGB images. We call this network **RGB-Net**. Depth-Net and RGB-Net are trained independently with depth and RGB images respectively. Then, weights are determined for the predictions of each network and the final 3D hand pose is estimated as a weighted average of the predictions of each net.

A graphical illustration of this approach can be seen in figure 4.5. We call this architecture **FusePred-Net**. The edges that connect the output of each network with the element that performs the score fusion, namely "Fuse Predictions", are weighted with $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ which refer to the weights of the predictions of the Depth-Net and RGB-Net respectively. The sum symbol denotes the weighted sum operation.

In the general case $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ are vectors with the same dimensionality as the number of predicted outputs which in our case is $3J$. That is, a weighted combination is computed independently between each one of the corresponding outputs of the two nets. We consider the simple case where the weight vectors have the constant value of 0.5 across all their entries, that is we compute the arithmetic mean of each predicted output of the two nets. However, more sophisticated methods for the determination of the weights can be incorporated such as an additional learning step that learns the optimal fusion weights, or another simple method is to compute the weights for each output of each regressor according to the error of the regressor in that output.

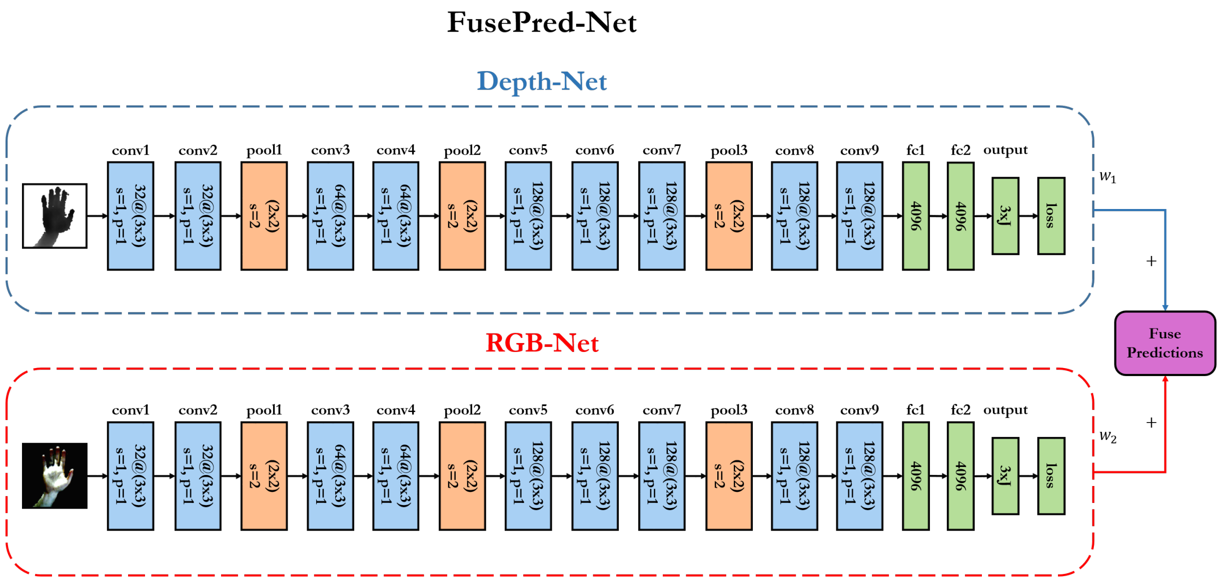


Figure 4.5: Score level fusion with RGB-Net and Depth-Net. Both networks are identical with the difference that they are trained on different domains, the RGB-Net with RGB images while the Depth-Net with depth images. After the predictions of both networks are obtained, the final 3D hand pose is estimated as a weighted sum of the predictions of both nets. The weights w_1 and w_2 are determined in advance.

These techniques have the additional beneficial effect, that different weights are used across different outputs and each network contributes to each output with a factor proportional to its ability to predict that output correctly.

Double-stream architecture fusion

The main drawback with score level fusion is that it is not able to learn pixel-wise correspondences between RGB and depth features since the fusion is performed at the output layers of the two streams. Moreover, input fusion learns pixel-wise correspondences only for the input RGB and depth images, We wish to be able to learn pixel-wise correspondences between color and depth features at any layer of our convolutional networks. To this end, we propose a double-stream architecture fusion for hand pose estimation. This method was first proposed in [59] for activity recognition using a spatial and a temporal stream. Here we investigate its performance in hand pose estimation. We adopt the idea of fusing double-stream architectures as well as some of the proposed fusion techniques.

We employ two streams similarly to score level fusion, the RGB stream and the depth stream. The goal is to fuse the two streams at any layer such that feature map responses of the RGB and depth streams at the same pixel location are put in correspondence. The motivation for feature map fusion is similar to input fusion, that is we want to learn combinations of color and depth features towards better pose estimation by blending the merits of both domains. The difference is that now we may fuse the networks at any particular layer, as input fusion is not necessarily the optimal level to fuse the networks and at a deeper layer of the network we may leverage richer fused feature representations. Additionally, we want to investigate different fusion functions as weighted combinations of the inputs is not necessarily the best choice of fusing feature maps. Depending on the layer of the ConvNet in which fusion is performed, someone can implement early fusion, late fusion or multilayer fusion.

A fusion function $f : \boldsymbol{x}^a, \boldsymbol{x}^b \rightarrow \boldsymbol{y}$, fuses two feature maps $\boldsymbol{x}^a \in \mathbb{R}^{H \times W \times D}$ and $\boldsymbol{x}^b \in \mathbb{R}^{H' \times W' \times D'}$ to produce an output blended map $\boldsymbol{y} \in \mathbb{R}^{H'' \times W'' \times D''}$, where H, W, D are the height, width and number of channels of the respective feature maps. Fusion functions can be applied straightforward when the two feature maps have the same spatial resolution. Hence, for simplicity we assume that $H = H' = H''$, $W = W' = W''$ and $D = D' = D''$. Among several different choices of fusion functions that can be considered, similarly to [59] we examine the cases of *max fusion*, *sum fusion*, *concatenation fusion* and *convolutional fusion*.

Max fusion which is denoted as $\boldsymbol{y} = f^{max}(\boldsymbol{x}^a, \boldsymbol{x}^b)$, computes the max between features maps $\boldsymbol{x}^a$ and $\boldsymbol{x}^b$ at the same spatial locations i, j and feature map channels k:

$$y^{max}_{i,j,k} = \max(x^a_{i,j,k}, x^b_{i,j,k}), \tag{4.15}$$

where $1 \leq i \leq H$, $1 \leq j \leq W$, $1 \leq k \leq D$ and $\boldsymbol{x}^a$, $\boldsymbol{x}^b$, $\boldsymbol{y} \in \mathbb{R}^{H \times W \times D}$. The ordering of the feature maps in a convolutional layer is arbitrary; hence, the correspondence between feature maps a and b is arbitrary. Nevertheless, the training procedure can learn the convolutional filters of the two streams that produced a and b respectively, in a way such that it can leverage advantageous correspondence between feature maps a and b.

Sum fusion, $\boldsymbol{y} = f^{sum}(\boldsymbol{x}^a, \boldsymbol{x}^b)$ computes the sum of two feature maps a and b at the same pixels i, j and feature map channels k as:

$$y^{sum}_{i,j,k} = x^a_{i,j,k} + x^b_{i,j,k}, \tag{4.16}$$

where the notation is the same as (4.15). Again correspondence between feature maps can be accomplished via learning the corresponding kernels.

Concatenation fusion, $\boldsymbol{y} = f^{cat}(\boldsymbol{x}^a, \boldsymbol{x}^b)$ is equivalent to input fusion which we described in Section 4.3.3, yet applied on feature maps instead of input images and can be injected at any particular layer. The feature maps are aggregated at the same spatial locations i, j and the resulting fused layer has a double number of feature maps. Concatenation function is given by:

$$y^{cat}_{i,j,2k} = x^a_{i,j,k},\ y^{cat}_{i,j,2k-1} = x^b_{i,j,k}, \tag{4.17}$$

where $\boldsymbol{y} \in \mathbb{R}^{H \times W \times 2D}$. Similarly to input fusion, concatenation fusion lets the subsequent convolutional layers to learn the correspondence between feature maps as weighted combinations of their responses.

Lastly, we consider the case of convolutional fusion denoted as $\boldsymbol{y} = f^{conv}(\boldsymbol{x}^a, \boldsymbol{x}^b)$. It first stacks the two feature maps at the same spatial locations i, j across the feature channels k, as in (4.17), producing $2D$ concatenated feature maps, and subsequently a convolutional layer is incorporated that utilizes D kernels of size 1×1 to reduce the feature maps from $2D$ to D. Hence, the convolutional layer employs D filters of size $1 \times 1 \times 2D$. Convolutional fusion is given by:

$$\boldsymbol{y}^{conv} = \text{conv}\left(\boldsymbol{y}^{cat}, \boldsymbol{f}\right), \tag{4.18}$$

where $\text{conv}(\cdot,\cdot)$ is the convolutional layer that convolves the $2D$ aggregated feature maps $\boldsymbol{y}^{cat}$ with a filter bank $\boldsymbol{f} \in \mathbb{R}^{1 \times 1 \times 2D \times D}$. The difference with input fusion is that in the case of input fusion the first convolutional layer computes for each spatial location i, j weighted combinations of the input channels in a local neighborhood 3×3 since it utilizes 3×3 filters. On the other hand convolutional fusion computes for each spatial location i, j weighted combinations of the input channels only in location i, j across the input channels and not in a neighborhood around i, j. Thus convolutional fusion learns directly pixel-wise correspondences while input fusion learns pixel-wise correspondences but averaged in a local neighborhood.

Feature map fusion can be inserted anywhere in the convolutional network after convolutional, fully-connected or pooling layers. And that is its main advantage in comparison with input fusion and score level fusion. At any particular layer we can investigate whether the fusion of color and depth features can lead to increasing

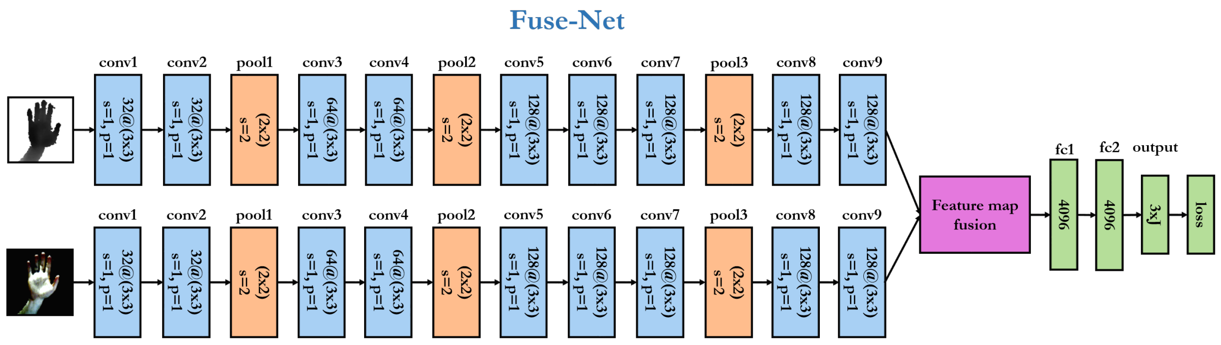


Figure 4.6: FuseNet: Our architecture for double-stream architecture fusing. Two streams are trained in parallel, the depth stream and the RGB stream with depth and RGB images respectively, and at any intermediate layer the feature maps may be fused. Depth and RGB streams have identical structure with Depth-Net and RGB-Net respectively with the difference that after the fusion the remaining part of each respective net is truncated. Here, for simplicity we demonstrate the fusion only in the last layer but it can be inserted anywhere in the network. After the fusion, the architecture continues as a single ConvNet.

estimation performance. In figure 4.6 we demonstrate our double-stream fusing architecture for hand pose estimation which we call **Fuse-Net**. Two separate streams, the depth stream and the RGB stream are put in parallel and trained simultaneously with depth and RGB images respectively. At any particular layer feature map fusion can be inserted with any of the fusion functions we presented above. For simplicity of demonstration we show the case were the fusion takes part after the last convolutional layer. Each stream is identical to Depth-Net and RGB-Net respectively with the difference that after the inserted fusion each respective net is truncated. After fusion, the network continues as a single ConvNet. The feature maps $\boldsymbol{x}^a$ and $\boldsymbol{x}^a$ we introduced in the fusion functions (4.15),(4.16),(4.17) and (4.18) are now a depth feature map from the Depth-Net and an RGB feature map from the RGB-Net.

4.4 Evaluation

In this section we evaluate our approach for hand pose estimation. We first introduce the benchmark dataset which we used for training the ConvNets as well as for evaluation. We then describe the evaluation metrics we used as well as details concerning the training procedure. We first evaluate all our different ConvNet architectures for hand pose estimation from a single depth image and subsequently we evaluate the different fusion techniques we proposed to find the most beneficial for hand pose estimation. Finally, we compare our methods with the state of the art on hand pose estimation.

4.4.1 Benchmark dataset

We evaluated our methods on NYU Hand pose dataset [13]. It contains 72757 training-set frames and 8252 test-set frames of RGB-D data, captured with PrimeSense, a structured-light RGB-D sensor. The training set contains samples from a single subject, while the test set contains samples from two subjects. While for each frame the dataset provides RGB-D data for three different views, a frontal view and two side views, we used only the frames that correspond to the frontal view, similarly to prior works [14, 15, 16].

Ground truth is annotated by fitting a skinned 3D hand model, using an offline hybrid optimization algorithm which combines particle swarm optimization and the Nelder-Mead algorithm. NYU Hand pose dataset provides very accurate ground truth annotations. As mentioned in [55], it exhibits very large pose variation, which makes it one of the most challenging datasets in the literature. Although the ground truth contains $J = 36$ annotated joints, we use a subset of $J = 14$ joints to follow the evaluation protocol of prior work [13, 14, 15, 16].

Although there are also other popular benchmark datasets in the literature, such as [46, 47], we did not consider another dataset since the NYU Hand pose dataset is the only one that provides RGB-D data. The Dexter dataset [57] also provides RGB-D data but is very limited in pose variation, and hence evaluating in NYU is more challenging. All the other datasets provide only depth images, and thus they are inappropriate for our fusion approaches.

4.4.2 Evaluation metrics

We used two evaluation metrics which are widely used in prior works ([14, 15, 16, 46, 47]). The first is the **per-joint error** averaged over the test set, that is the average Euclidean distance between the ground truth joint location and the predicted joint location over the frames of the test set, for each joint independently.

The second evaluation metric is the **success-rate**, that is the fraction of test set frames whose max-joint-error is below a threshold. In other words, it measures the fraction of test set frames for which each predicted joint is below a maximum Euclidean distance between the ground truth and the predicted joint locations. This is a very challenging evaluation metric since with a single displaced joint prediction the whole pose can be regarded as false positive. It can be considered as analog to the accuracy metric in classification, and the variation of the value of the threshold, results in graphs similar to ROC curves on which we base our evaluation when we consider this metric.

4.4.3 Experimental setup and training

We performed our experiments in a three step pipeline. In the first step, we performed several experiments to evaluate all different ConvNet architectures we considered, for training a ConvNet with depth images (Net1-Net9), in order to select the best performing architecture. In the second step, we employed the best performing ConvNet architecture from the previous step, and we performed hyperparameter optimization with cross validation in order to find the best hyperparameters to train our models. Finally, in the last step, we used the best performing architecture from the first step, and the best values for the hyperparameters from the second, with which we trained and subsequently evaluated our models and fusion techniques.

As it is fairly well-known, convolutional networks have several sensitive hyperparameters that affect the optimization as well as the generalization of the model. Slightly different values for these hyperparameters can lead to underfitting and overfitting issues. To this end, we performed hyperparameter optimization to select good settings for the hyperparameters. For cross validation we split the training set in training and validation set, where we train our models with the trainining set and select the best hyperparameters based on the performance on the validation set. The validation set size was set to 25% of the training set size while the validation examples were sampled uniformly form the training set.

For generating several different combinations of hyperparameters values for evaluation there are two popular methods, grid search and random search. In grid search, sets of values are defined for each hyperparameter and the cartesian product between all sets is computed, that is all possible combinations of hyperparameters values between the values of the sets. Random search [60], was introduced for hyperparameter optimization in deep learning models. The motivation is that when the number of hyperparameters becomes big, which is the case for neural nets, grid search becomes inefficient as the search space is bigger; hence, more configurations should be considered for each hyperparameter, and the cartesian product will produce a large number of possible combinations. Training ConvNets is computationally expensive and training such a big number of models would demand a lot of resources.

To this end, [60] proposes to sample independently each hyperparameter from a different distribution. A desired number of experiments is defined and for each training-evaluation experiment different values of hyperparameters are sampled from the hyperparameter distributions. The authors show that this method searches effectively the hyperparameter space and less experiments are needed in order to find good values.

We train our models using mini-batch stochastic gradient descent with momentum. We performed preliminary experiments with simple SGD without momentum but since it is much slower it was leading to underfitting of our models. We use a batch size of 128 training examples per iteration, while we train our models for 100 epochs. The hyperparameters that we validate are the learning rate, the momentum term and dropout probability. For the validation we use random search. For the learning rate we define a uniform distribution in the log-scale, that is:

$$\mathrm{lr} \sim 10^{U(\log_{10} a, \log_{10} b)}, \tag{4.19}$$

where lr is the learning rate and $U(a, b)$ is a uniform distribution in the range (a, b). We sample momentum from a uniform distribution in log-scale too. For the dropout probability p we define a uniform distribution in the linear scale. We performed cross validation in an iterative fashion were we start with a big range for the distributions of the hyperparameters and subsequently we shrink the ranges based on the evaluation of the previous cross-validation step. We start with a range of $(10^{-3}, 10^{-1})$ for the learning rate, $(0.8, 1)$ for the momentum and $(0, 0.1)$ for the dropout probability. We iterate this process three times and each time we sample 50 different hyperparameter combinations. For the initial ranges we performed preliminary experiments.

For the first step of our experimental pipeline, we use manually tuned hyperparameters values for training all our ConvNet architectures. We use the value of 0.01 for the learning rate, 0.9 for the momentum and 0.03 for the dropout probability.

We use the best performing architecture from this step that is Depth-Net to perform hyperparameter optimization with cross validation. The best hyperparameter settings we obtained was 0.009 for the learning rate, 0.98 for the momentum and 0.03 for the dropout probability. It is surprising that such a small dropout probability is sufficient, since typically in classification problems dropout is set around 0.5. In our problem, this hyperparameter is very sensitive and slightly larger values than the optimal ones can very easily lead to underfitting. Nevertheless, using the right probability, dropout is a very effective regularizer for hand pose estimation and probably for regression generally. Although our model is large with the use of dropout it does not overfit and provide good generalization performance. We conclude that dropout is a very strong regularizer for regression problems, and very small probability values should be used in comparison with classification problems.

We use early stopping such that the optimal number of training epochs is selected and training stops when the validation error saturates. For learning rate decay strategy, we decay the learning rate with a factor of 0.5 every time the validation error saturates. On each epoch the training examples are shuffled where preliminary experiments showed that this increases the performance.

For the double-stream architecture fusion we show our experiments by fusing their feature maps at the fourth convolutional layer. Of course the fusion can be inserted anywhere in the network, but we did not consider other cases due to limited computational resources. Our intuition of fusing at the fourth convolutional layer is that the fusion takes place approximately in the middle of the network; hence, before the fusion the convolutional layers learn meaningful feature representations to by fused, and after the fusion the subsequent layers extract further useful knowledge from the fused feature maps.

4.4.4 Implementation details

We implemented our approach in Python using Lasagne [61] which is a framework that provides abstractions for Theano library [62]. The experiments ran on Opuntia Cluster, were we utilized two HP Proliant SL 250 compute blades, each one equipped with an NVIDIA Tesla K40 GPU with 24GB of memory and an Intel Xeon E5-2680v2 2.8 GHz CPU with 64GB of memory.

Table 4.3: Self-comparison of our ConvNet architectures which are trained with depth images and are described in table 4.1 and table 4.2. Each row represents a different architecture. In the first column is the name of each net. In the second and third row, we state the training and validation error respectively which are measured with the mean square error (MSE) between the nets' predictions and the ground truth across the training and validation set respectively. We choose the best performing architecture based on its validation error. The winning ConvNet is Net6 which is our deepest architecture.

ConvNet	Training error	Validation Error
Net1	0.00366	0.00447
Net2	0.00359	0.00419
Net3	0.00353	0.00411
Net4	0.00341	0.00364
Net5	0.00235	0.00325
Net6	**0.00183**	**0.00242**
Net7	0.00384	0.00434
Net8	0.00415	0.00462
Net9	0.00393	0.00439

4.4.5 Self-comparison of ConvNet configurations

We make a self-comparison of our ConvNet architectures trained with depth images. We provide the performance of each net on the training and validation set by measuring the mean square error (MSE) between the predicted joint locations and the ground truth across the training and validation set respectively. We present our experimental results in table 4.3. In the first column is the name of each ConvNet, and in the second and third column is the training and validation error respectively.

Net2 performs better than Net1 where Net2 stacks three convolutional layers on top of each other while Net1 two. This implies that is beneficial two stack more convolutional layers on top of each other since as we explained before the deeper convolutional layers have implicitly larger receptive fields and apart from that, more convolutions followed by nonlinearities provide more expressive feature representations.

Now we keep Net2 for our following two comparisons as it outperforms Net1.

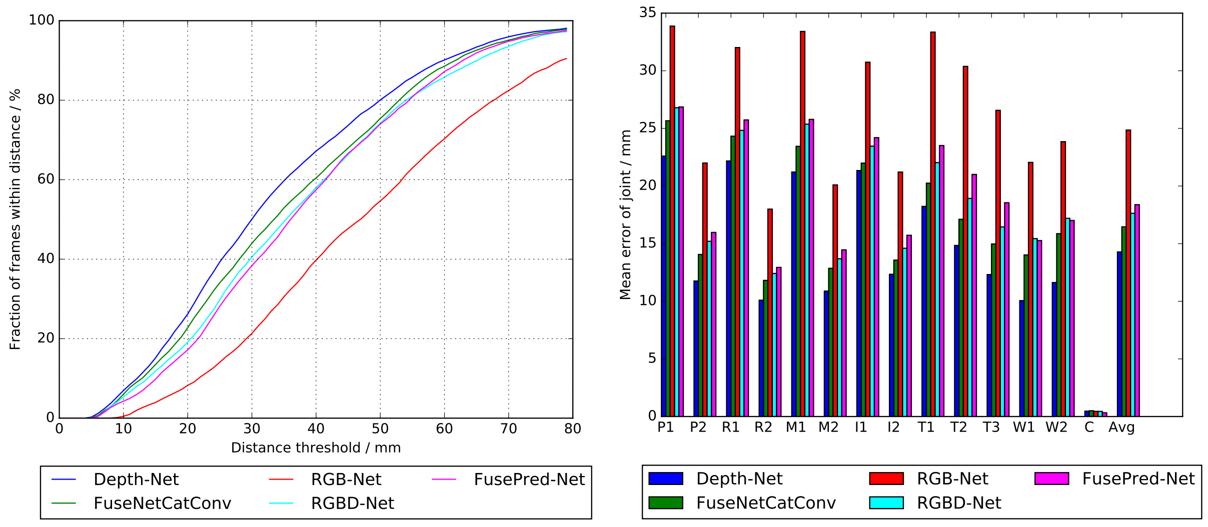
We compare Net2 with Net3 and Net4 with which they have the same number of convolutional and pooling layers with the same ordering, but with the difference that Net3 has bigger filters with varying size across the network while Net4 has double size of filters than all the other nets. We can see that the training and validation errors of Net2 and Net3 is quite close which implies that using bigger filters with varying size does not have significant impact in performance. That is why in all our rest architectures we employ 3×3 convolutional kernels as they show almost identical performance with bigger filters. Net4 has more significant improvement in performance, in terms of validation error, as if you compare Net2 and Net4 their training error is quite close while their validation error has more significant difference in favor of Net4. This finding suggests that using more filters does not affect drastically the training error but improves the generalization of the model, as more feature maps in a layer means more feature map combinations for the computation of each feature map of the subsequent layer (since convolutional layers convolve each kernel with each input feature map channel and sum the results), and hence more general features of high abstraction. Yet, using the double number of filters affects highly the training time and the model becomes computationally inefficient, concerning our limited computational resources. Hence, in all the rest configurations we keep the same number of filters as in Net2 (form 32 to 128).

Next, we wanted to test the impact in performance of the depth of the network. To this end, first we considered the case of keeping the architectural design of Net1 and Net2, namely stacking two and three conv layers respectively, but by adding more convolutional groups of stacked convolutional layers followed by max-pooling. For this purpose we evaluated Net7-Net9. Net7 and Net8 are extensions of Net1 which add one and two more stacks of two convolutional layers followed by max pooling respectively. Net9 is extension of Net2 which adds one more stack of three convolutional layers followed by max-pooling. We observe that while Net7 has very small improvement performance comparing to Net1, Net8 performance degrades. Similarly, Net9 validation error increases comparing to Net2. Thus, for both Net1 and Net2, even with an increasing depth validation error increases when adding more pooling layers. Net1 and Net2 has three pooling layers while Net8 and Net9 has five and four respectively. We conclude that the number of max-pooling layers is very important since redundant subsampling can lead to information loss. To this end, we propose Net5 and Net6 which keep the same number of max-pooling as Net2 or Net1 but increase the depth of the network by adding multiple convolutional layers on top of each other without adding more max-pooling layers. In the first two convolutional groups Net5 stacks two convolutional layers while Net2 one. Lastly, Net6 extends Net5 by adding two more convolutional layers stacked on top of each other but without the addition of max-pooling after.

As you can see from table 4.3, the best performing architecture is Net6 which is the Depth-Net we introduced in Section 4.3.2. In Section 4.4.7, we show that Depth-Net has state-of-the-art performance in our benchmark dataset and outperforms previous convolutional network based methods. We conclude that the depth of the network is of high importance but should be combined with controlled number of max-pooling layers by stacking multiple convolutional layers on top of each other before a max-pooling operation.

4.4.6 Is fusion beneficial to the accuracy of the convolutional networks?

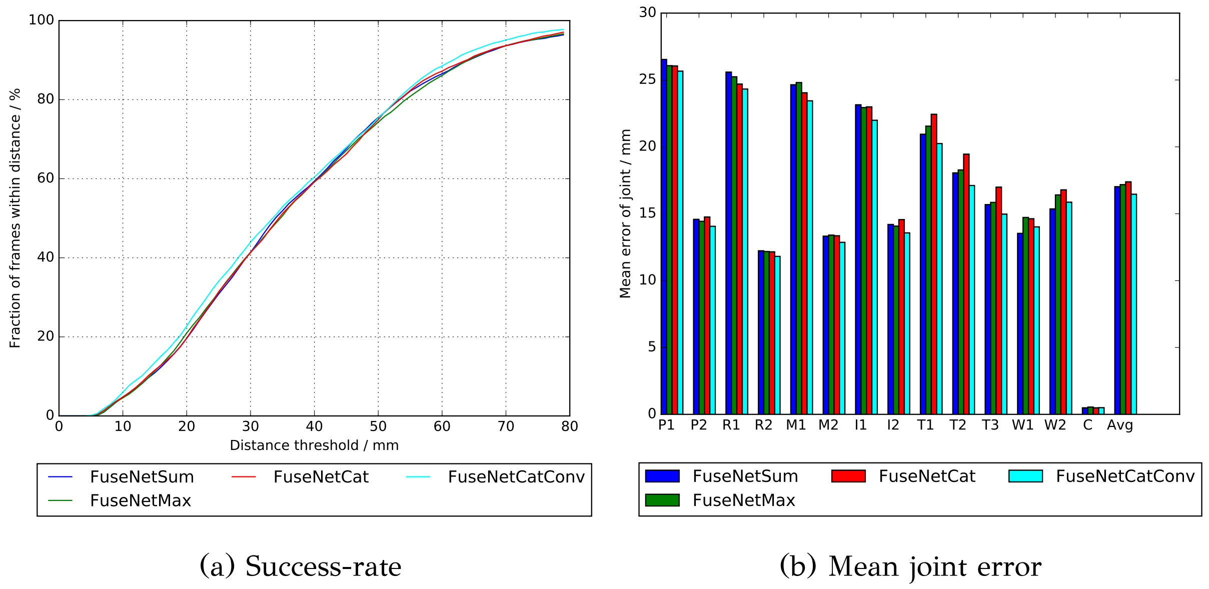
Now, we compare the performance of our fusing approaches with each other but also with Depth-Net, which is our best performing architecture trained with depth images. Our intention is to investigate which fusion method performs better as well as whether fusing RGB with depth information enhances the accuracy of the predictions with respect to training and predicting from single depth images. For the evaluation we show graphs of the evaluation indices we described in Section 4.4.2. We refer to each approach with the name of the respective architecture, namely RGB-Net which is trained only with RGB images, RGBD-Net which refers to the input fusion approach, FusePred-Net that is the score level fusion and Fuse-Net which is the double-stream architecture fusion. All these architectures are trained using the same net configuration and hyperparameters that works best for Depth-Net which we presented in Section 4.4.5 and Section 4.4.3 respectively. The comparative results are presented in figure 4.7 and figure 4.8. Figure 4.7 compares the performance of our fusion approaches between each other as well as with Depth-Net. Figure 4.8 compares the performance of double-stream architecture fusion for each fusion function. The best performing fusion function is used for the comparisons in figure 4.7. Figure 4.7a and figure 4.8a depict the success-rate, where the horizontal axis represents the distance threshold (mm) and the vertical axis represents the fraction of frames (%) for which the squared error for each joint is below the distance threshold. Figure 4.7b and figure 4.8b illustrate the mean joint error where the horizontal axis represents the different joints, and the vertical axis represents the mean error per joint across the test set. The rightmost bars in figure 4.7b and figure 4.8b show the mean error per joint averaged over all joints. Abbreviations: pinky (P), ring (R), middle (M), index (I), thumb (T), wrist (W), palm center (C). For all fingers, the indices 1 and 2 refer to the fingertip joints, for the thumb, the index 3 refers to the lower joint, and for the wrist, the indices 1 and 2 refer to the left and right wrist position respectively.



(a) Success-rate

(b) Mean joint error

Figure 4.7: Comparison of fusing approaches with Depth-Net. (a) In the success-rate graph, the horizontal axis represents the distance threshold (mm) and the vertical axis represents the fraction of frames (%) where the maximum joint error is below the distance threshold. (b) In the mean joint error graph, the horizontal axis represents the various joints and the vertical axis indicates the the mean error per joint (%). For the double-stream architecture fusion, the performance of the convolutional fusion function is shown since it outperforms all the other fusion functions (Fig. 4.8). Abbreviations: pinky (P), ring (R), middle (M), index (I), thumb (T), wrist (W), palm center (C). For all fingers, the indices 1 and 2 refer to the fingertip joints, for the thumb, the index 3 refers to th lower joint, and for the wrist, the indices 1 and 2 refer to the left and right wrist position respectively.



(a) Success-rate

(b) Mean joint error

Figure 4.8: Comparison of different fusing functions used in the double-stream architecture. (a) Success-rate and (b) the mean joint error. While all the fusion functions perform comparably, convolutional fusion performs best among them.

We first look at figure 4.8, as that we first compare the different fusion functions in double-stream fusion, and subsequently we use the best performing fusion function to compare double-stream fusion with the rest of our fusion methods. In general, it may be observed that all fusion functions perform quite comparably. If we observe carefully at figure 4.8a we can see that the success-rate of convolutional fusion is higher with respect to the rest of the fusion functions. In figure 4.8b, it is more clear that convolutional fusion has lower mean joint error averaged over all joints than the other fusion function as well as lower error on each joint separately. We conclude that convolutional fusion can create better pixel-wise correspondences between RGB and depth feature maps and as a result leverage more expressive fused feature representations towards more accurate hand pose estimation. The main reason is that convolutional fusion involves the extra step of convolving the concatenated feature maps with 1×1 kernels, that is the value at each position (i, j) of the fused feature map is a weighted combination of the values of the concatenated feature maps at the same spatial location (i, j). Hence, convolutional fusion, first fuses the feature maps by concatenation and subsequently learns inherently pixel-wise correspondences between depth and RGB feature maps by learning the kernel weights. Sum fusion, max fusion, and concatenation fusion only fuse the feature maps and let subsequent convolutional layers to learn the pixel-wise correspondences. It turns out that it is better to involve a step that learns pixel-wise correspondences inside the fusion function than letting the subsequent convolutional layers to learn the correspondences. Consequently, in the comparison of our fusing techniques that follows for double-stream architecture fusion we show the performance using the convolutional fusion function.

If we look at figure 4.7, surprisingly Depth-Net outperforms all our fusion approaches. We discuss the results for each fusion approach. Since score level fusion computes an average of the predictions of Depth-Net and RGB-Net, we plot the performance of RGB-Net in both evaluations to understand its effect on the predictions fusion. Figure 4.7a shows clearly that RGB-Net has significantly lower success-rate comparing to Depth-Net. To have a better understanding of the performance of RGB-Net in comparison to Depth-Net on each joint separately we look at figure 4.7b. We observe that the RGB-Net performs worse for each joint in comparison to Depth-Net. Nevertheless, if we look at the rightmost bar at figure 4.7b, the mean joint error of RGB-Net averaged over all joints is around 25mm which is a relatively low error and suggests that the predictions of RGB-Net could be leveraged. This is maybe possible by using different weights for each prediction of each respective net instead of using a simple average. A possible solution could be to set the prediction weights of each net for each joint to be proportional to the error of that net on that joint.

For the deteriorated performance of input fusion, we conjecture that the input of the ConvNet is a very early stage to fuse the data and as a result the network is not able to extract useful features from the fusion. That is because, RGBD-Net fuses raw depth and RGB images at the first convolutional layer by computing weighted averages of the respective channel of each image. Hence, the feature maps of the first convolutional layer are weighted combinations of RGB and depth images, which may result in images that contain the artifacts of both domains since the fusion takes place in a very early stage. This suggests that first feature representations should be extracted from both RGB and depth images, that will preserve the useful information from each domain and discard information that is not related with the task, and then the features should be fused to combine more meaningful representations. That is why we propose a double-stream architecture fusion, since first each net compute feature maps from the convolutional layers and subsequently fusion is inserted to create meaningful pixel-wise correspondence between feature maps which contain mostly task-related information opposite to raw images which contain also noisy artifacts. We observe that the performance of score level fusion and input fusion is very similar, with input fusion to perform slightly better than score level fusion in both success-rate and mean error per joint.

Double-stream architecture fusion outperforms both input fusion and score level fusion. Our experiments confirm that it is better to fuse the feature maps of two networks and let the subsequent learning define correspondences between feature maps than to fuse the inputs or the predictions of two nets. Double-stream architecture fusion learns more meaningful feature representations and exploits better, combined RGB and depth information, and as a result, provide more accurate pose estimations than input fusion and score level fusion.

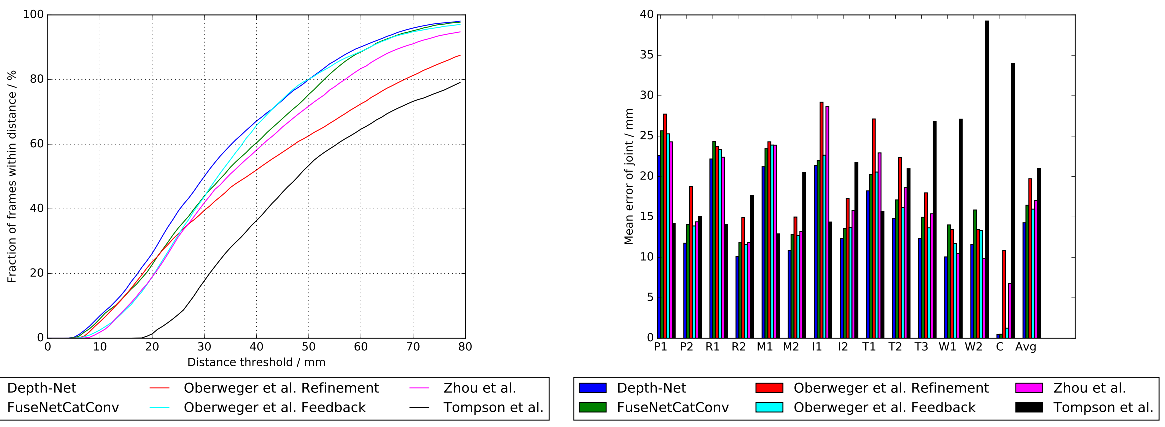
We observe, that Depth-Net has the best performance among all our approaches and outperforms all fusion methods. Despite that double-stream fusion provides us more accurate predictions than input fusion and score level fusion, its performance is still limited compared with Depth-Net which is trained only with depth images. From our experiments we conclude that RGB-D fusion does not leverage further useful information towards more accurate pose prediction. Training a large network only with depth images performs better than incorporating also RGB images. Nevertheless, the increased performance of double-stream fusion compared with the other fusion approaches reveals that it can learn better feature combinations. We performed experiments by fusing only at the fourth convolutional layer. Maybe double-stream architecture fusion at deeper layers of the network can learn better pixel-wise correspondences and provide more accurate predictions than using just a convolutional network trained with depth images.

Lastly, we discuss some general observations about the performance of all methods on each joint by looking at figure 4.7b. Firstly, we observe that all ConvNets perform very well on the palm center where the error is near zero. Even the RGB-Net has very low error rate on the palm center. This suggests, that regression convolutional networks can serve as very good hand detectors given datasets with annotations of the hand center, which is a very useful observation and can be further leveraged in computer vision methods. Lastly we observe that all methods have higher errors in the fingertips and lower errors in the middle joints and the thumb in the lowest joint too. That is because there are several configurations of the human hand where the joints are visible while the fingertips are occluded. An example is a closed hand. Another reason is that depth images have missing values on the hand boundaries and fingertips lie on the hand boundaries, and hence the missing values make the fingertip detection even more difficult. We conclude that fingertip detection is a very challenging problem and more challenging than joint detection.

4.4.7 Comparison with the state of the art

In this section, we compare our proposed deep architecture Depth-Net as well as the double-stream architecture fusion with the state-of-the-art. The baselines we are using for our comparison are Tompson et al. [13], Oberweger et al. [14], Oberweger et al. [15] and Zhou et al. [16] which are all convolutional network based approaches. For each method we use their originally published predictions on the test set. Oberweger et al. [14] provide the predictions for both the network with the pose prior layer as well as the predictions based on the refinement stage. We use the predictions of the refinement stage since they are more accurate. Tompson et al. [13] provide the estimated 2D heat-map locations. We follow the protocol of [13, 14, 15] and augment the 2D locations with the depth at that 2D location from the depth images, so that we obtain comparable 3D pose predictions. If the augmented depth for a joint lies outside the hand cube we assign to the joint the ground truth depth similarly to [15] to alleviate large errors caused by occluded joints.

Figure 4.9a shows the comparisons based on the success-rate while figure 4.9b presents the mean joint error. It may be observed that Depth-Net outperforms all baseline methods in both evaluation metrics. The key difference between all these approaches and our proposed architecture is the depth of the network. Tompson et al. [13] use a convolutional network with 2 convolutional layers to infer 2D joint positions. Oberweger et al. [14] incorporate a convolutional network with three convolutional layers to infer 3D joint positions by introducing the pose prior layer and subsequently refine the joint locations in an iterative fashion. Zhou et al. [16] use the same network as in [14] but the authors also introduce a hand model layer. Oberweger et al. [15] utilize three convolutional networks to train a feedback loop, the first with one convolutional layer and the other two with four convolutional layers. The average error over all joints in figure 4.9b shows that all approaches perform well, yet our proposed architecture outperforms these approaches both in terms of average error over all joints and in mean error per joint. Despite the efforts of these approaches to improve the accuracy of the predictions of the ConvNet by involving



(a) Success-rate

(b) Mean joint error

Figure 4.9: Comparison of our proposed deep architecture Depth-Net and the double-stream architecture fusion with the state-of-the-art methods of Tompson et al. [13], Oberweger et al. [14], Oberweger et al. [15] and Zhou et al. [16]. (a) Success rate, and (b) mean joint error.

extra steps (e.g. prior+refinement [14], hand model layer [16]), their limited performance is due to the shallow networks they incorporate. 3D hand pose estimation is a problem of high complexity, and the limited capacity of such shallow architectures is not sufficient to learn good mappings from input images to 3D hand poses. Hence, they cannot learn very good feature representations and have limited generalization performance which affects the subsequent steps they involve. We obtain state-of-the-art performance with a single network that does not involve any extra step and directly estimates 3D hand poses from depth images. Our proposed architecture (Depth-Net) with nine convolutional layers - significantly deeper than the baseline we compare - has higher complexity and can learn better feature representations, and hence provide better generalization performance on the problem of 3D hand pose estimation. Double-stream architecture fusion outperforms most of the other approaches and achieves approximately similar accuracy with [15]. The method of Oberweger et al. [15] is the competitor with the closest performance to Depth-Net. In this work, the authors train a feedback loop using three different ConvNets with a limited depth relatively to ours. Yet our approach with a single ConvNet of increased depth outperforms the method in [15] and provides state-of-the-art performance.

We conclude that due to the high complexity of 3D hand pose estimation, high capacity models are needed for accurate pose estimations. From our comparative results we witness that large convolutional networks which directly estimate the 3D hand pose, provide better estimation accuracy than shallow models that involve extra refinement steps [14, 15] or impose hand pose priors [14, 16]. Consequently, it is more advantageous to use a large network with sufficient representation capacity that learn good mappings from the input image directly to the 3D pose space, over using shallow networks and try to improve their performance with subsequent steps. We obtain state-of-the-art performance with a single, large convolutional network trained with depth images and designed with important architecture considerations in mind. Double-stream fusion performs comparably with the previous state-of-the-art method of Oberweger et al. [15], yet we do not observe further improvements in performance in comparison with Depth-Net which suggests that fusion of RGB and depth information does not leverage more expressive feature representations for more accurate pose estimation. Experiments with fusion in a different layer may result in increased performance, and we will investigate this issue in a future work.

4.4.8 Qualitative results

In this section, we will show some qualitative results of the estimated poses of Depth-Net. We show results only from Depth-Net, since it is our best performing model. We plot the estimated poses together with the ground truth poses for comparison along with their corresponding hand images. We show the results for both the 2D and 3D cases. For the 2D case, the 2D hand depth images are plotted with the 2D poses (from the 3D estimated pose we plot only X and Y components), while for the 3D case the 3D point clouds that correspond to the 2D hand depth images are plotted with the 3D poses. We created the point cloud using equations (4.3), (4.4), (4.5) and converting each hand pixel to a 3D point.

We show the results for both subjects of the test set. In figure 4.10, we show the results for the first subject, while in figure 4.11 we show the results for the second subject. The training set contains hand images only from the first subject. We plot the results of both subjects to examine how well the model generalizes to the second subject. In both figures 4.10 and 4.11, the left column shows results for the 2D case while the right column for the 3D case. The ground truth poses are colored with blue while the estimated poses with red. While we show the original 2D images, we rotate the 3D point clouds properly such that the 3D poses are visible. The colors in the point clouds denote different depth values where the colorbar shows the corresponding depth for each color.

We can see that in general, the predictions for the first subject in figure 4.10 are very accurate. Since the model is trained on this subject it is natural to learn its hand topology, viewpoint change and gestures well. The results for the second subject in figure 4.11 show that the model is less accurate on this subject. Nevertheless, the predictions of the model on this subject are still quite close to the ground truth. Specifically, in images 4.11a-4.11f we can see that the predictions are quite good, while in images 4.11g-4.11j the predictions drift more from the ground truth.

The model predicts well the new unknown subject, but its generalization performance is still limited to some level. This happens because in the specific dataset the training set contains only one subject. We conclude that for the problem of hand pose estimation, the training set should contain multiple subjects such that the model observes different hand shapes, bigger range of gestures and viewpoints from each subject. This makes the model able to generalize better to new subjects with varying hand shapes and gestures.

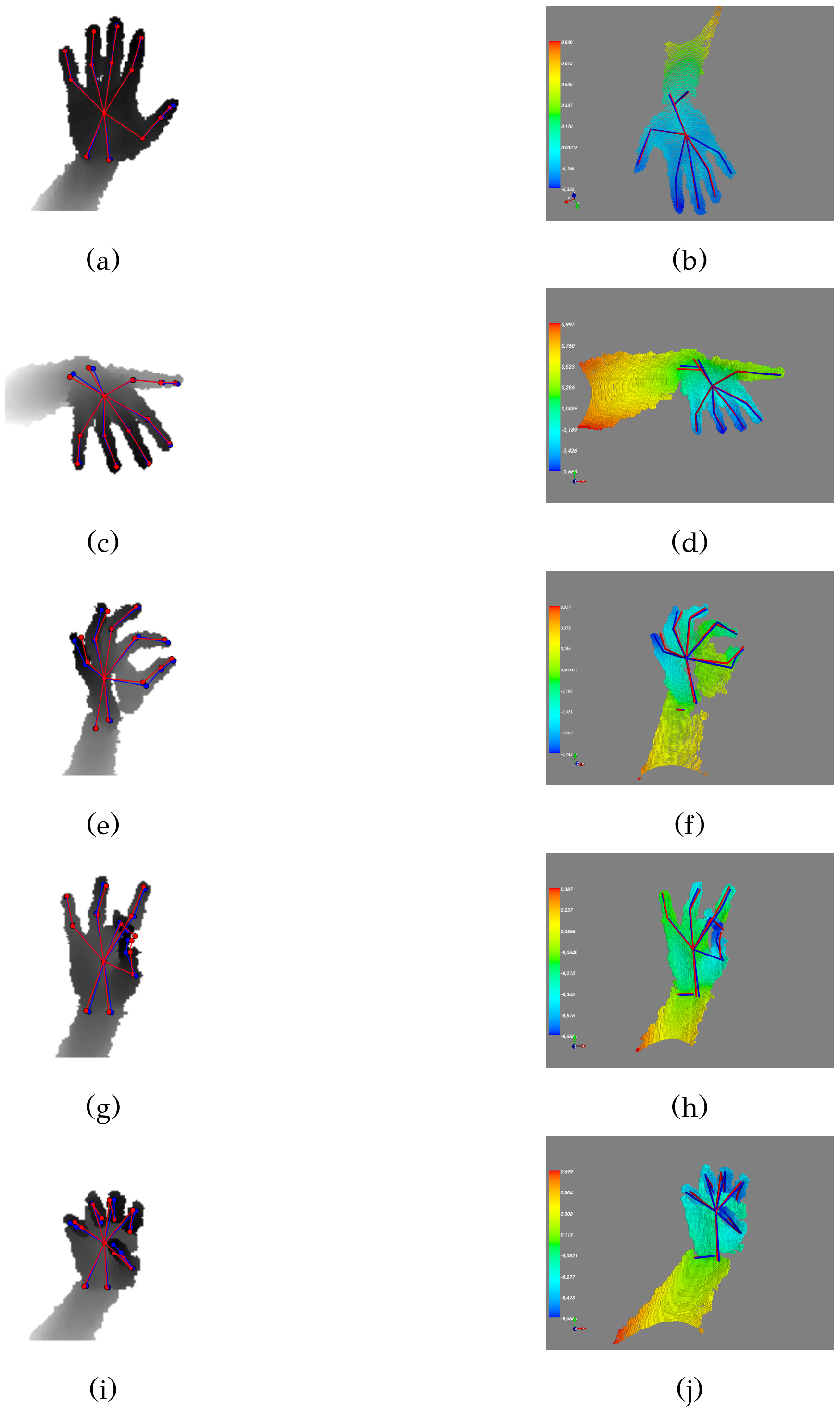


Figure 4.10: We show some qualitative results by plotting both the groundtruth pose and the estimated pose of Depth-Net along with the corresponding hand image for both the 2D and 3D case. Left: 2D depth images with the 2D poses. Right: 3D point clouds with the 3D poses. In each case, the groundtruth poses are colored with blue while the estimated poses with red. Here, we show the results for the first subject of the test set. The training set contains images only from this subject. Since the model is trained on this subject, the predictions are very accurate.

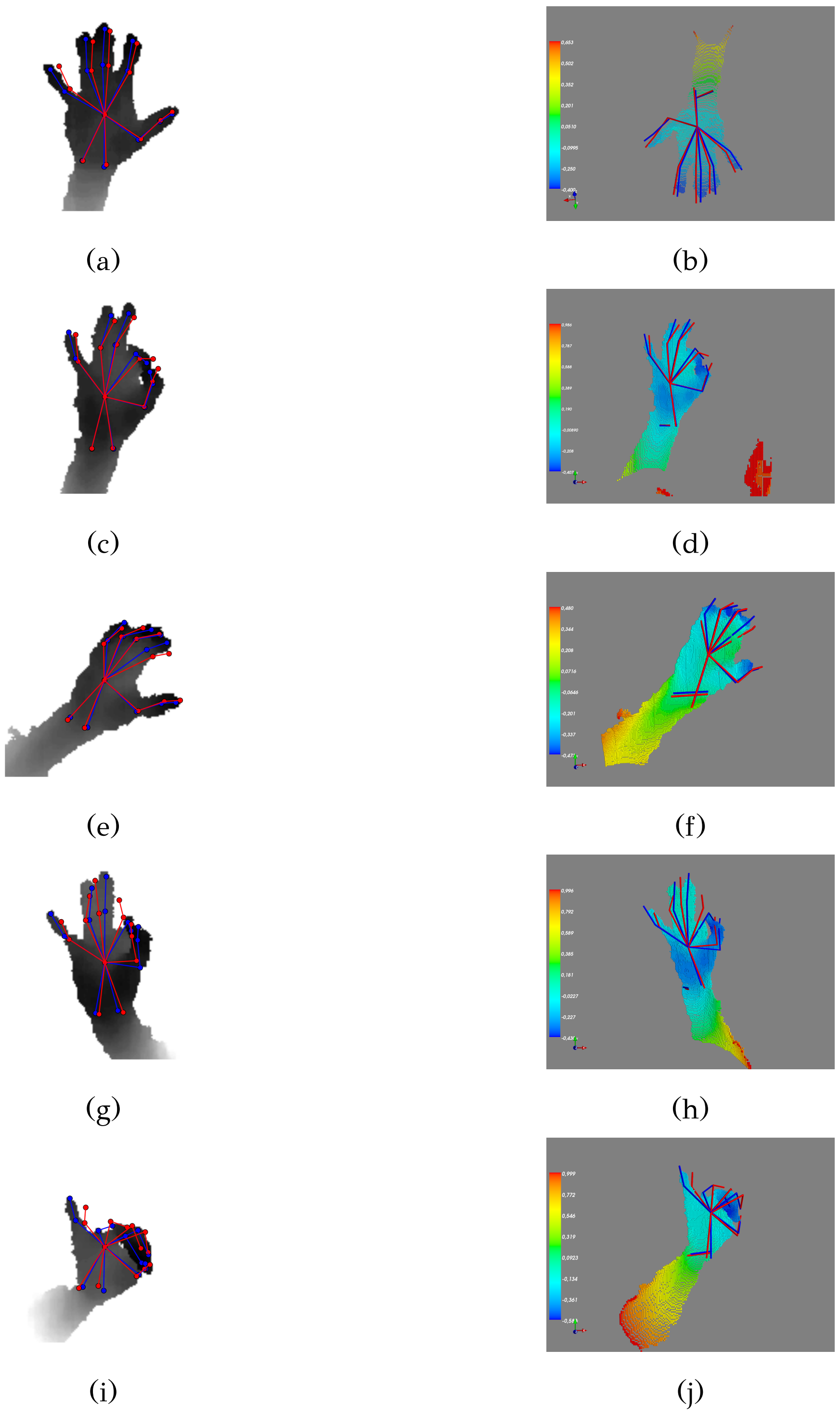


Figure 4.11: Here we show some qualitative results for the second subject of the test set who is not contained in the training set. The predictions are less accurate for this subject which suggests that the model is subject to some level of overfitting. Nevertheless, the predictions are still satisfying for this subject. Left: 2D depth images with the 2D poses. Right: 3D point clouds with the 3D poses. In each case, the groundtruth poses are colored with blue while the estimated poses with red.

Chapter 5

Conclusion

We studied the problem of 3D hand pose estimation with convolutional networks using RGB-D data. Our contribution is twofold. First, we designed and evaluated thoroughly several convolutional network architectures trained with depth images in order to find which performs better for the problem of hand pose estimation. We investigated the importance of the depth as well as the importance of the general layout of the network, such as the number of pooling layers, the size and the number of the convolutional kernels. We paid special attention to the training procedure, were we performed hyperparameter optimization with cross validation in order to find the best setting for several of our hyperparameters, since they are very sensitive and have significant impact to training. We regularized our models using dropout, were our experiments show that dropout serves as a very good regularizer for regression convolutional networks. Interestingly, much smaller dropout probabilities should be used in comparison with classification tasks.

Our best performing architecture is our deepest convolutional network with nine convolutional layers. To the best of our knowledge, no prior work employed such a large convolutional network for hand pose estimation. This is our main proposed architecture which we called Depth-Net. Our experimental analysis showed that Depth-Net outperforms the state-of-the-art. We conclude that the depth of the network is of great importance towards more accurate hand pose estimation, since 3D hand pose estimation is a problem with high complexity and requires high capacity models that can learn good mappings from depth images to 3D hand poses. Apart from the depth, the number of max-pooling layers is crucial since redundant max-pooling can lead to information loss, and as a result, decreased performance. The solution is to stack multiple convolutional layers on top of each other before a max-pooling layer instead using a max-pooling layer after each convolutional layer. Finally, we found that it is beneficial to use a homogeneous architecture of 3×3 convolutional kernels across the whole network with a stride of 1 and a pooling of 1. This is a VGG [36] based architecture proposed for image recognition with state-of-the-art performance. Since then, it has been applied to several problems with state-of-the-art performance, such as in activity recognition [59]. We confirmed its state-of-the-art performance in the problem of 3D hand pose estimation with depth images.

Our second contribution, is the study of the benefits of combining RGB and depth information with convolutional networks for 3D hand pose estimation, where we proposed three novel approaches that fuse RGB-D information. Input fusion, aggregates RGB and depth images and trains a convolutional network with four input channels. In this approach correspondences between RGB and depth features are learned in the first convolutional layer of the network. Score level fusion trains independently two convolutional networks with RGB and depth images respectively and fuses their predictions. Finally, double-stream architecture fusion trains two convolutional networks in parallel and fuses their feature maps at any intermediate layer using feature map fusion functions where we employed different fusion functions proposed in state-of-the-art activity recognition methods [59]. Double-stream fusion, fuses the feature map responses at each spatial location and lets subsequent learning to leverage meaningful correspondences between fused feature maps. Our experimental results showed that double-stream architecture fusion outperforms, input fusion and score level fusion since in these methods the information is fused in a very early and a very late stage of the network respectively, and as a result, the fusion is more coarse with limited performance. Thus, double-stream fusion leverages more expressive feature representations from the combination of RGB and depth information. Double-stream fusion performs comparably to the state of the art. Nevertheless, our proposed architecture trained only with depth images outperforms double-stream fusion and provides us state-of-the-art performance in the problem of 3D hand pose estimation. We conclude from our experiments that RGB-D fusion cannot leverage further useful information for more accurate hand pose estimation. Nevertheless, our knowledge of the problem is limited since we did not performed experiments by fusing double-stream architectures in multiple layers of the networks and with more sophisticated fusion functions.

In a future work, we intend to investigate further the problem of double-stream architecture fusion for 3D hand pose estimation. We plan to study the effect of fusing feature maps in different layers of the respective networks. Furthermore, we would like to research on new fusion functions that can exploit better, useful information on RGB data. Another future goal is to incorporate unsupervised learning in the training of the convolutional network in order to impose a prior on the hand poses, learned through hand images.

Bibliography

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] http://dsdeepdive.blogspot.com/2016/03/optimizations-of-gradient-descent.html.

[3] http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[4] https://prateekvjoshi.com/2015/10/20/dissecting-bias-vs-variance-tradeoff-in-machine-le

[5] http://parse.ele.tue.nl/education/cluster2.

[6] L. Fei-fei, R. Fergus, and P. Perona, "One-shot learning of object categories," *IEEE Transactions On Pattern Analysis And Machine Intelligence*, vol. 28, no. 4, pp. 594–611, 2006.

[7] https://www.slideshare.net/zukun/icml2012-learning-hierarchies-of-invariant-features.

[8] https://devblogs.nvidia.com/parallelforall/deep-learning-nutshell-core-concepts/.

[9] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, pp. 2278–2324, Nov. 1998.

[10] http://eblearn.sourceforge.net/beginner_tutorial2_train.html.

[11] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25 (NIPS 2012)*, (Lake Tahoe, NV, USA), pp. 1097–1105, Dec. 2012.

[12] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper With Convolutions," in *Proceedings*

of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR), (Boston, MA, USA), pp. 1–9, June 2015.

[13] J. Tompson, M. Stein, Y. Lecun, and K. Perlin, "Real-Time Continuous Pose Recovery of Human Hands Using Convolutional Networks," *ACM Transactions on Graphics*, vol. 33, pp. 169:1–169:10, Sept. 2014.

[14] M. Oberweger, P. Wohlhart, and V. Lepetit, "Hands Deep in Deep Learning for Hand Pose Estimation," in *Proceedings of the 20th Computer Vision Winter Workshop (CVWW)*, (Schloss Seggau, Styria, Austria), pp. 21–30, Feb. 2015.

[15] M. Oberweger, P. Wohlhart, and V. Lepetit, "Training a Feedback Loop for Hand Pose Estimation," in *Proceedings of the 15th IEEE International Conference on Computer Vision (ICCV)*, (Santiago, Chile), pp. 3316–3324, Dec. 2015.

[16] X. Zhou, Q. Wan, W. Zhang, X. Xue, and Y. Wei, "Model-based Deep Hand Pose Estimation," in *Proceedings of the 25th International Joint Conference on Artificial Intelligence (IJCAI)*, (New York, NY, USA), pp. 2421–2427, July 2016.

[17] F. Rosenblatt, "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain," *Psychological Review*, vol. 65, pp. 386–408, 1958.

[18] K. Jarrett, K. Kavukcuoglu, M. Ranzato, and Y. LeCun, "What is the best multi-stage architecture for object recognition?," in *Proceedings of the 12th IEEE International Conference on Computer Vision (ICCV)*, (Kyoto, Japan), pp. 2146–2153, Sept. 2009.

[19] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier Nonlinearities Improve Neural Network Acoustic Models," in *Proceedings of the 30th International Conference on Machine Learning (ICML) Workshop on Deep Learning for Audio, Speech, and Language Processing*, (Atlanta, GA, USA), June 2013.

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," in *Proceedings of the 15th IEEE International Conference on Computer Vision (ICCV)*, (Santiago, Chile), pp. 1026–1034, December 2015.

[21] B. Polyak, “Some methods of speeding up the convergence of iteration methods,” *USSR Computational Mathematics and Mathematical Physics*, vol. 4, no. 5, pp. 1–17, 1964.

[22] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, “On the importance of initialization and momentum in deep learning,” in *Proceedings of the 30th International Conference on Machine Learning (ICML)*, (Atlanta, GA, USA), pp. 1139–1147, June 2013.

[23] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*. Springer Publishing Company, Incorporated, 1 ed., 2014.

[24] Y. Nesterov, “A method of solving a convex programming problem with convergence rate O(1/sqr(k)),” *Soviet Mathematics Doklady*, vol. 27, no. 2, pp. 372–376, 1983.

[25] J. Duchi, E. Hazan, and Y. Singer, “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization,” *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, July 2011.

[26] M. D. Zeiler, “ADADELTA: An Adaptive Learning Rate Method,” *CoRR*, vol. abs/1212.5701, 2012.

[27] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” *CoRR*, vol. abs/1412.6980, 2014.

[28] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 9, (Sardinia, Italy), pp. 249–256, May 2010.

[29] J. Martens, “Deep learning via Hessian-free optimization,” in *Proceedings of the 27th International Conference on Machine Learning (ICML)*, (Haifa, Israel), pp. 735–742, June 2010.

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.

[31] Y. LeCun, *Generalization and network design strategies*. Elsevier, 1989.

[32] F. F. Li, A. Karpathy, and J. Johnson, "CS231n: Convolutional Neural Networks for Visual Recognition." `http://cs231n.stanford.edu/`. Stanford University.

[33] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," Master's thesis, 2009.

[34] A. Coates, H. Lee, and A. Ng, "An analysis of single-layer networks in unsupervised feature learning," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, vol. 15 of *JMLR Workshop and Conference Proceedings*, (Ft. Lauderdale, FL, USA), pp. 215–223, Apr. 2011.

[35] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[36] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *CoRR*, vol. abs/1409.1556, 2014.

[37] J. Shotton, T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore, "Real-time Human Pose Recognition in Parts from Single Depth Images," *Communications of the ACM*, vol. 56, pp. 116–124, Jan. 2013.

[38] L. Ballan, A. Taneja, J. Gall, L. V. Gool, and M. Pollefeys, "Motion Capture of Hands in Action using Discriminative Salient Points," in *Proceedings of the 12th European Conference on Computer Vision (ECCV)*, (Firenze, Italy), pp. 640–653, October 2012.

[39] I. Oikonomidis, M. I. A. Lourakis, and A. A. Argyros, "Evolutionary Quasi-Random Search for Hand Articulations Tracking," in *Proceedings of the 27th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Columbus, Ohio, USA), pp. 3422–3429, June 2014.

[40] N. K. Iason Oikonomidis and A. Argyros, "Efficient model-based 3D tracking of hand articulations using Kinect," in *Proceedings of 22nd the British Machine Vision Conference (BMVC)*, (University of Dundee), pp. 101.1–101.11, Aug. 2011.

[41] M. de La Gorce, D. J. Fleet, and N. Paragios, “Model-Based 3D Hand Pose Estimation from Monocular Video,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, pp. 1793–1805, Sept. 2011.

[42] C. Qian, X. Sun, Y. Wei, X. Tang, and J. Sun, “Realtime and Robust Hand Tracking from Depth,” in *Proceedings of the 27th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Columbus, OH, USA), pp. 1106–1113, June 2014.

[43] L. Ge, H. Liang, J. Yuan, and D. Thalmann, “Robust 3D Hand Pose Estimation in Single Depth Images: From Single-View CNN to Multi-View CNNs,” in *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NV, USA), pp. 3593–3601, June 2016.

[44] C. Keskin, F. Kiraç, Y. E. Kara, and L. Akarun, “Real time hand pose estimation using depth sensors,” in *Proceedings of the 13th IEEE International Confernce on Computer Vision (ICCV) Workshops*, (Barcelona, Spain), pp. 1228–1234, Nov. 2011.

[45] R. Girshick, J. Shotton, P. Kohli, A. Criminisi, and A. Fitzgibbon, “Efficient Regression of General-activity Human Poses from Depth Images,” in *Proceedings of the 13th IEEE International Conference on Computer Vision (ICCV)*, (Barcelona, Spain), pp. 415–422, Nov. 2011.

[46] X. Sun, Y. Wei, S. Liang, X. Tang, and J. Sun, “Cascaded Hand Pose Regression,” in *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Boston, MA, USA), pp. 824–832, June 2015.

[47] D. Tang, H. J. Chang, A. Tejani, and T.-K. Kim, “Latent Regression Forest: Structured Estimation of 3D Articulated Hand Posture,” in *Proceedings of the 27th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Columbus, OH, USA), pp. 3786–3793, June 2014.

[48] D. Tang, T.-H. Yu, and T.-K. Kim, “Real-Time Articulated Hand Pose Estimation Using Semi-supervised Transductive Regression Forests,” in *Proceedings of the 14th IEEE International Conference on Computer Vision (ICCV)*, (Sydney, Australia), pp. 3224–3231, Dec. 2013.

[49] A. Erol, G. Bebis, M. Nicolescu, R. D. Boyle, and X. Twombly, “Vision-based hand pose estimation: A review,” *Computer Vision and Image Understanding*,

vol. 108, no. 1–2, pp. 52–73, 2007. Special Issue on Vision for Human-Computer Interaction.

[50] Z. Zhang, “Microsoft Kinect Sensor and Its Effect,” *IEEE MultiMedia*, vol. 19, pp. 4–10, Apr. 2012.

[51] C. Keskin, F. Kıraç, Y. E. Kara, and L. Akarun, “Hand Pose Estimation and Hand Shape Classification Using Multi-layered Randomized Decision Forests,” in *Proceedings of the 12th European Conference on Computer Vision (ECCV)*, (Firenze, Italy), pp. 852–863, Oct. 2012.

[52] C. Xu and L. Cheng, “Efficient Hand Pose Estimation from a Single Depth Image,” in *Proceedings of the 14th IEEE International Conference on Computer Vision (ICCV)*, (Sydney, Australia), pp. 3456–3462, December 2013.

[53] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox, “Learning to Generate Chairs With Convolutional Neural Networks,” in *Proceedings of the 28th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Boston, MA, USA), pp. 1538–1546, June 2015.

[54] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks,” in *Proceedings of the 13th European Conference on Computer Vision (ECCV)*, (Zurich, Switzerland,), pp. 818–833, Sept. 2014.

[55] J. S. Supančič, III, G. Rogez, Y. Yang, J. Shotton, and D. Ramanan, “Depth-Based Hand Pose Estimation: Data, Methods, and Challenges,” in *Proceedings of the 15th IEEE International Conference on Computer Vision (ICCV)*, (Santiago, Chile), pp. 1868–1876, Dec. 2015.

[56] I. Oikonomidis, “Tracking the Articulated Motion of Two Strongly Interacting Hands,” in *Proceedings of the 25th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Providence, RI, USA), pp. 1862–1869, June 2012.

[57] S. Sridhar, A. Oulasvirta, and C. Theobalt, “Interactive Markerless Articulated Hand Motion Tracking Using RGB and Depth Data,” in *Proceedings of the 14th IEEE International Conference on Computer Vision (ICCV)*, (Sydney, Australia), pp. 2456–2463, Dec. 2013.

[58] K. Simonyan and A. Zisserman, "Two-Stream Convolutional Networks for Action Recognition in Videos," in *Advances in Neural Information Processing Systems 27 (NIPS 2014)*, (Montreal, Canada), pp. 568–576, Dec. 2014.

[59] C. Feichtenhofer, A. Pinz, and A. Zisserman, "Convolutional Two-Stream Network Fusion for Video Action Recognition," in *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (Las Vegas, NV, USA), pp. 1933–1941, June 2016.

[60] J. Bergstra and Y. Bengio, "Random Search for Hyper-Parameter Optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, Feb. 2012.

[61] S. Dieleman, J. Schlüter, C. Raffel, E. Olson, S. K. Sønderby, D. Nouri, D. Maturana, M. Thoma, E. Battenberg, J. Kelly, J. D. Fauw, M. Heilman, D. M. de Almeida, B. McFee, H. Weideman, G. Takács, P. de Rivaz, J. Crall, G. Sanders, K. Rasul, C. Liu, G. French, and J. Degrave, "Lasagne: First release.," Aug. 2015.

[62] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky, Y. Bengio, A. Bergeron, J. Bergstra, V. Bisson, J. Bleecher Snyder, N. Bouchard, N. Boulanger-Lewandowski, X. Bouthillier, A. de Brébisson, O. Breuleux, P.-L. Carrier, K. Cho, J. Chorowski, P. Christiano, T. Cooijmans, M.-A. Côté, M. Côté, A. Courville, Y. N. Dauphin, O. Delalleau, J. Demouth, G. Desjardins, S. Dieleman, L. Dinh, M. Ducoffe, V. Dumoulin, S. Ebrahimi Kahou, D. Erhan, Z. Fan, O. Firat, M. Germain, X. Glorot, I. Goodfellow, M. Graham, C. Gulcehre, P. Hamel, I. Harlouchet, J.-P. Heng, B. Hidasi, S. Honari, A. Jain, S. Jean, K. Jia, M. Korobov, V. Kulkarni, A. Lamb, P. Lamblin, E. Larsen, C. Laurent, S. Lee, S. Lefrancois, S. Lemieux, N. Léonard, Z. Lin, J. A. Livezey, C. Lorenz, J. Lowin, Q. Ma, P.-A. Manzagol, O. Mastropietro, R. T. McGibbon, R. Memisevic, B. van Merriënboer, V. Michalski, M. Mirza, A. Orlandi, C. Pal, R. Pascanu, M. Pezeshki, C. Raffel, D. Renshaw, M. Rocklin, A. Romero, M. Roth, P. Sadowski, J. Salvatier, F. Savard, J. Schlüter, J. Schulman, G. Schwartz, I. V. Serban, D. Serdyuk, S. Shabanian, E. Simon, S. Spieckermann, S. R. Subramanyam, J. Sygnowski, J. Tanguay, G. van Tulder, J. Turian, S. Urban, P. Vincent, F. Visin, H. de Vries, D. Warde-Farley, D. J. Webb, M. Willson, K. Xu, L. Xue, L. Yao, S. Zhang, and Y. Zhang, "Theano:

A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016.

[63] https://en.wikipedia.org/wiki/Convolution.

[64] http://mathworld.wolfram.com/Convolution.html.

[65] H. Larochelle, "Neural networks class." https://www.youtube.com/playlist?list=PL6Xpj9I5qXYEcOhn7TqghAJ6NAPrNmUBH. Université de Sherbrooke.

[66] L. Bottou, "Online Algorithms and Stochastic Approximations," in *Online Learning and Neural Networks* (D. Saad, ed.), Cambridge, UK: Cambridge University Press, 1998. revised, oct 2012.

[67] L. Bottou, "Stochastic Gradient Descent Tricks," in *Neural Networks: Tricks of the Trade (Second Edition)* (G. Montavon, G. B. Orr, and K.-R. Müller, eds.), Lecture Notes in Computer Science, pp. 421–436, Springer, 2012.

[68] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade (Second Edition)* (G. Montavon, G. B. Orr, and K.-R. Müller, eds.), Lecture Notes in Computer Science, pp. 437–478, Springer, 2012.

[69] S. Ruder, "An overview of gradient descent optimization algorithms." http://sebastianruder.com/optimizing-gradient-descent/.

[70] G. Hinton, "Neural Networks for Machine Learning (Coursera Video Lectures)." https://www.coursera.org/learn/neural-networks. University of Toronto.

[71] A. Gibiansky, "Convolutional Neural Networks." http://andrew.gibiansky.com/blog/machine-learning/convolutional-neural-networks/.

[72] J. Kafunah, "Backpropagation in Convolutional Neural Networks." http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

Short Biography

Evangelos Kazakos received his B.Sc. and the M.Sc. in Computer Science from the Department of Computer Science and Engineering, University of Ioannina, Greece in 2014 and 2017 respectively, where he has been a member of the Information Processing and Analysis Research Group. During the summer of 2016, he worked as an intern at the Computational Biomedicine Lab, Univeristy of Houston, USA. His research interests include computer vision and machine learning with a special focus on deep learning and convolutional networks.