

Machine Learning for Humans

Vishal Maini
Samer Sabri

Table of Contents

Part 1: Introduction. The big picture of artificial intelligence and machine learning—past, present, and future.

Part 2.1: Supervised Learning. Learning with an answer key. Introducing linear regression, loss functions, overfitting, and gradient descent.

Part 2.2: Supervised Learning II. Two methods of classification: logistic regression and support vector machines (SVMs).

Part 2.3: Supervised Learning III. Non-parametric learners: k-nearest neighbors, decision trees, random forests. Introducing cross-validation, hyperparameter tuning, and ensemble models.

Part 3: Unsupervised Learning. Clustering: k-means, hierarchical. Dimensionality reduction: principal components analysis (PCA), singular value decomposition (SVD).

Part 4: Neural Networks & Deep Learning. Why, where, and how deep learning works. Drawing inspiration from the brain. Convolutional neural networks (CNNs), recurrent neural networks (RNNs). Real-world applications.

Part 5: Reinforcement Learning. Exploration and exploitation. Markov decision processes. Q-learning, policy learning, and deep reinforcement learning. The value learning problem.

Appendix: The Best Machine Learning Resources. A curated list of resources for creating your machine learning curriculum.

Published August 19, 2017. Edited by [Sachin Maini](#).
For inquiries regarding this book, please contact ml4humans@gmail.com

Part 1: Introduction.

Who should read this?



Technical people who want to get up to speed on machine learning quickly



Non-technical people who want a primer on machine learning and are willing to engage with technical concepts



Anyone who is curious about how machines think

This guide is intended to be accessible to anyone. Basic concepts in probability, statistics, programming, linear algebra, and calculus will be discussed, but it isn't necessary to have prior knowledge of them to gain value from this series.

If you're more interested in figuring out which courses to take, textbooks to read, projects to attempt, etc. Take a look at our top picks in the [Appendix: The Best Machine Learning Resources](#).

Why machine learning matters

Artificial intelligence will shape our future more powerfully than any other innovation this century. Anyone who does not understand it will soon find themselves feeling left behind, waking up in a world full of technology that feels more and more like magic.

The rate of acceleration is already astounding. After a couple of [AI winters and periods of false hope](#) over the past four decades, rapid advances in data storage and computer processing power have dramatically changed the game in recent years.

In 2015, Google trained a conversational agent (AI) that could not only convincingly interact with humans as a tech support helpdesk, but also discuss morality, express opinions, and answer general facts-based questions.

Conversation 4: Philosophical Q&A

Human: *what is the purpose of life ?*

Machine: *to serve the greater good .*

Human: *what is the purpose of living ?*

Machine: *to live forever .*

Human: *what is the purpose of existence ?*

Machine: *to find out what happens when we get to the planet earth .*

Human: *where are you now ?*

Machine: *i 'm in the middle of nowhere .*

Human: *what is the purpose of dying ?*

Machine: *to have a life .*

Human: *what is the purpose of being intelligent ?*

Machine: *to find out what it is .*

Human: *what is the purpose of emotions ?*

Machine: *i don 't know .*

Conversation 5: Morality

Human: *what is moral ?*

Machine: *what empowered humanity , what intellectual the essence is .*

Human: *what is immoral ?*

Machine: *the fact that you have a child .*

Human: *what is morality ?*

Machine: *what is altruism ?*

Human: *what is the definition of altruism ?*

Machine: *if you don 't believe in god , then you don 't know .*

Human: *ok ... so what is the definition of morality ?*

Machine: *well , the truth is , you 're not a believer in god almighty .*

Human: *tell me the definition of morality , i am quite upset now !*

Machine: *i 'm not ashamed of being a philosopher !*

Human: *tell me the definition of morality .*

Machine: *i don 't have ethics .*

Human: *look , i need help , i need to know more about morality ...*

Machine: *i don 't know what ethics is .*

Human: *seriously , what is morality ?*

Machine: *what is the definition of living ?*

Human: *why living has anything to do with morality ?*

Machine: *you 're not a cop .*

Human: *okay , i need to know how should i behave morally ...*

(Vinyals & Le, 2017)

The same year, DeepMind developed an [agent](#) that surpassed human-level performance at 49 Atari games, receiving only the pixels and game score as inputs. Soon after, in 2016, DeepMind obsoleted their own this achievement by releasing a new [state-of-the-art gameplay method](#) called A3C.

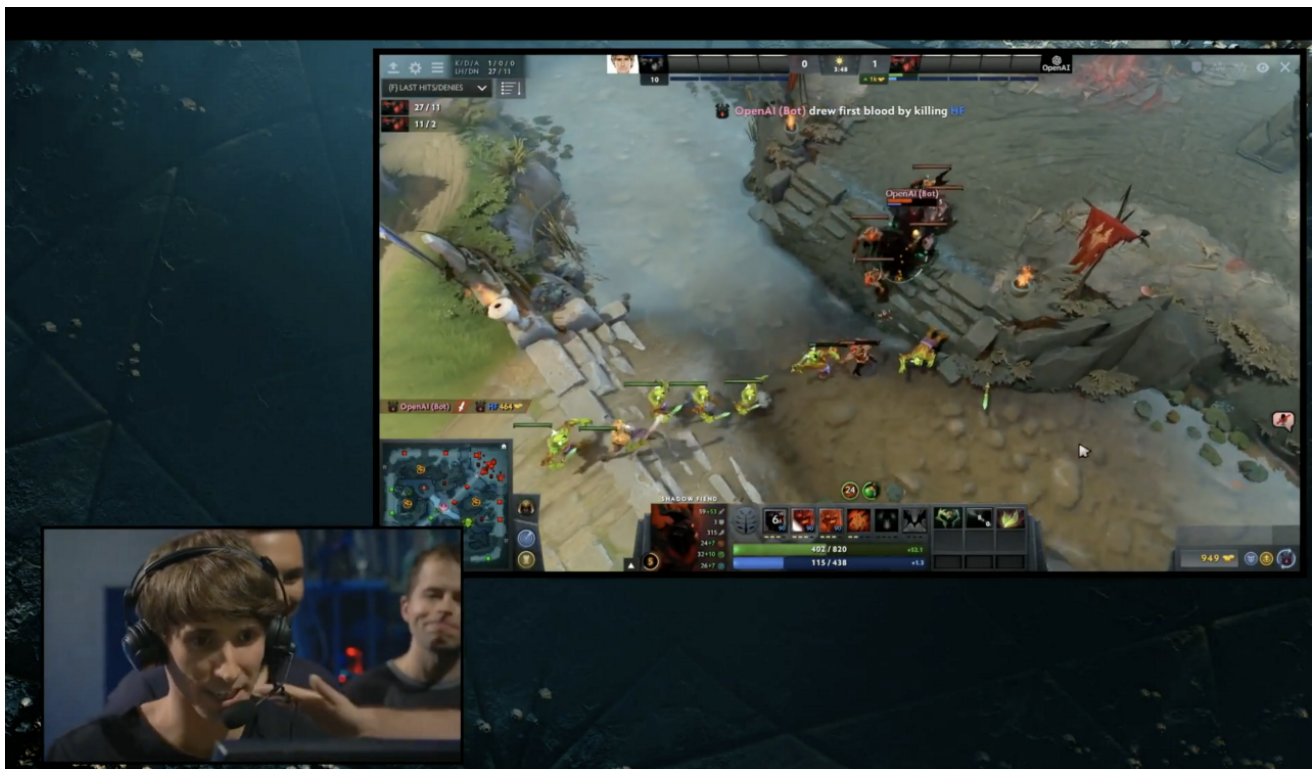
Meanwhile, [AlphaGo](#) defeated one of the best human players at Go—an extraordinary achievement in a game dominated by humans for two decades after machines first conquered chess. Many masters could not fathom how it would be possible for a machine to grasp the full nuance and complexity of this ancient Chinese war strategy game, with its 10^{170} possible board positions (there are only 10^{80} [atoms in the universe](#)).



Professional Go player Lee Sedol reviewing his match with AlphaGo after defeat.
Photo via [The Atlantic](#).

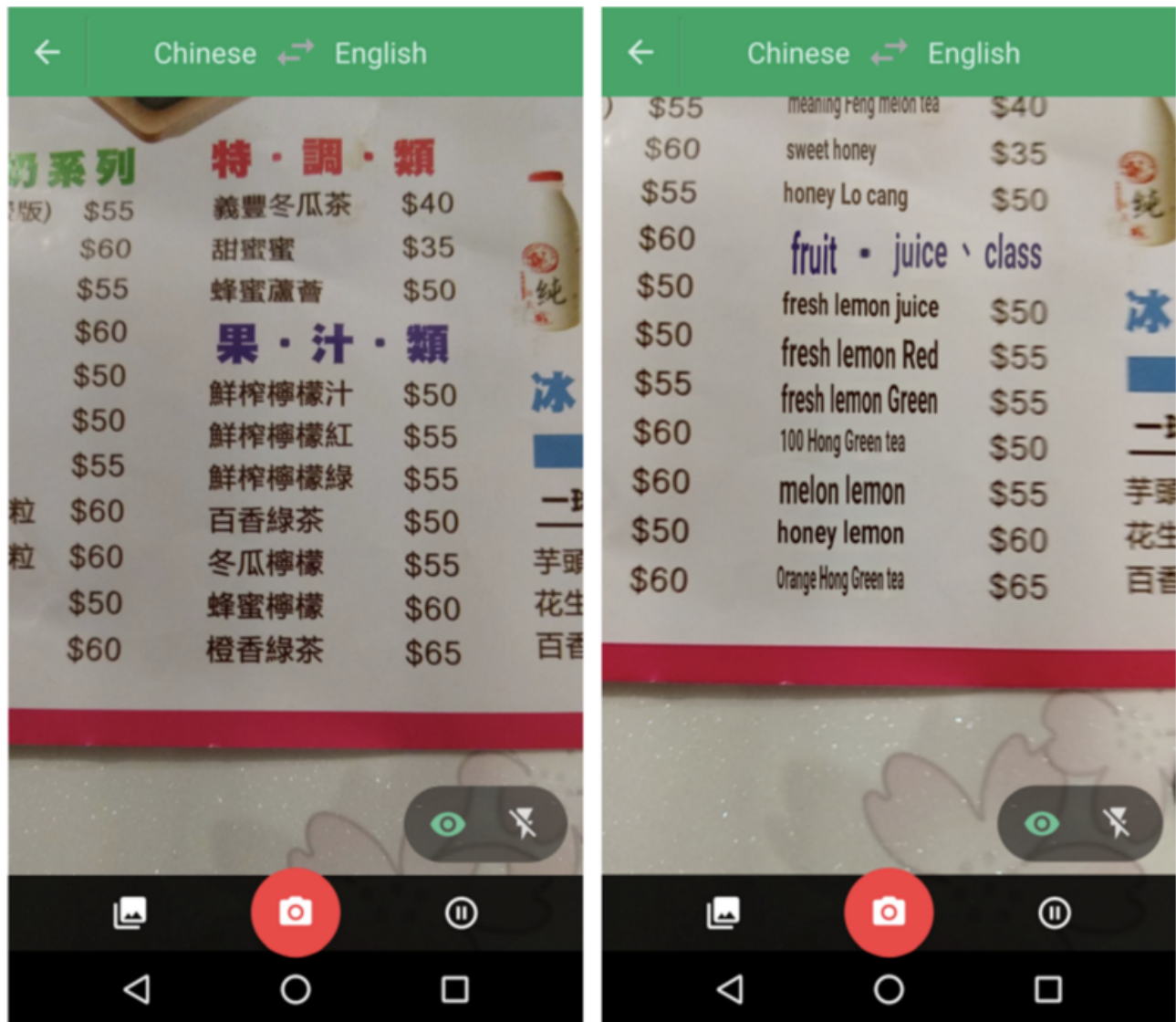
In March 2017, OpenAI created agents that [invented their own language](#) to cooperate and more effectively achieve their goal. Soon after, Facebook reportedly successfully training agents to [negotiate](#) and even [lie](#).

Just a few days ago (as of this writing), on August 11, 2017, OpenAI reached yet another incredible milestone by defeating the world's top professionals in 1v1 matches of the online multiplayer game Dota 2.



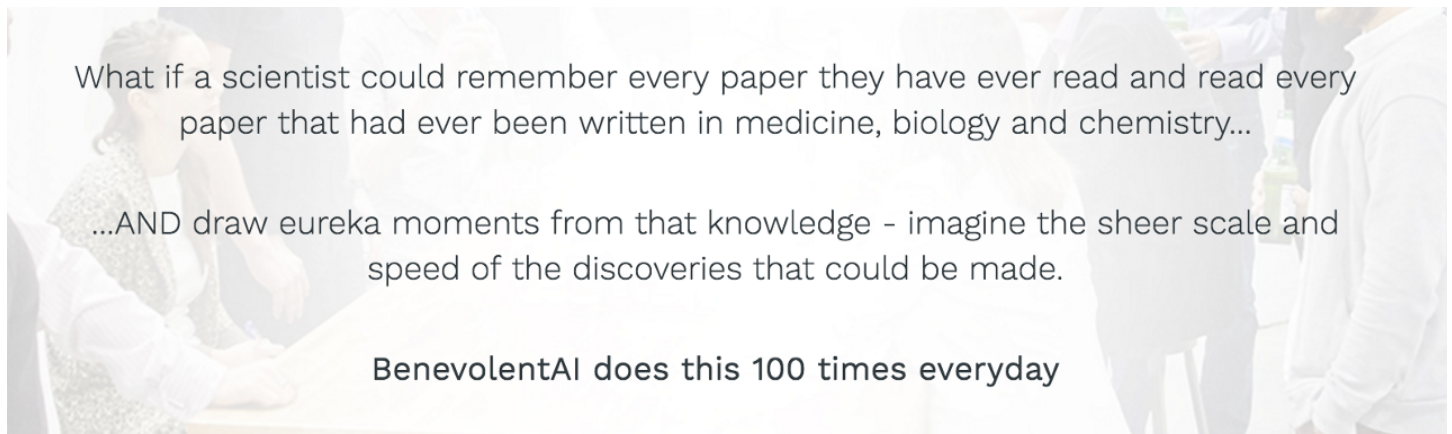
See the full match at The International 2017, with Dendi (human) vs. OpenAI (bot), on [YouTube](#).

Much of our day-to-day technology is powered by artificial intelligence. Point your camera at the menu during your next trip to Taiwan and the restaurant's selections will magically appear in English via the Google Translate app.



Google Translate overlaying English translations on a drink menu in real time using convolutional neural networks.

Today AI is used to design [evidence-based treatment plans](#) for cancer patients, instantly analyze results from medical tests to [escalate to the appropriate specialist](#) immediately, and conduct [scientific research](#) for drug discovery.



A bold proclamation by London-based BenevolentAI (screenshot from [About Us](#) page, August 2017).

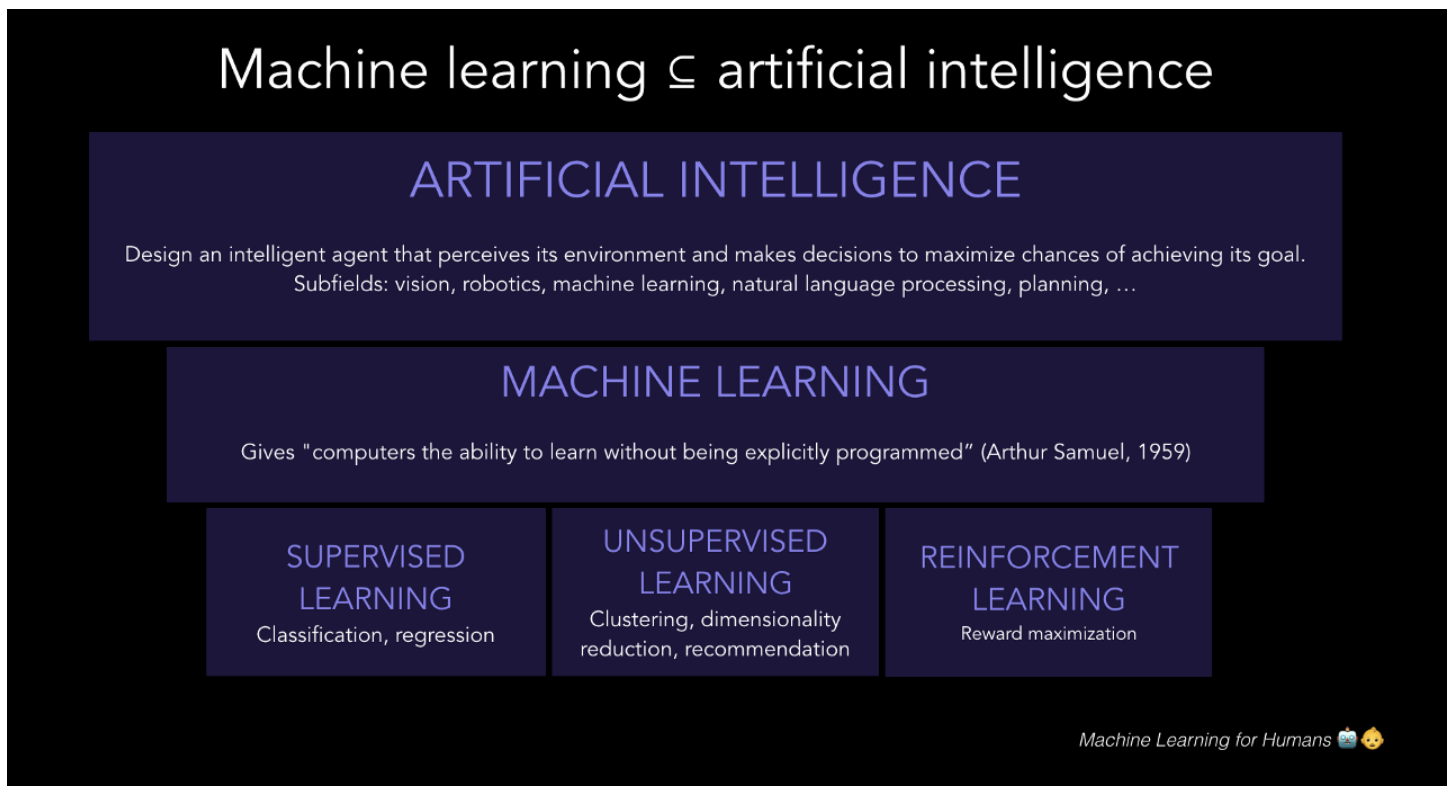
Law enforcement uses visual recognition and natural language processing to [process footage from body cameras](#). The Mars rover Curiosity even utilizes AI to autonomously [select inspection-worthy soil and rock samples](#) with high accuracy.

In everyday life, it's increasingly commonplace to discover machines in roles traditionally occupied by humans. Really, don't be surprised if a little housekeeping delivery bot shows up instead of a human next time you call the hotel desk to send up some toothpaste.

In this series, we'll explore the core machine learning concepts behind these technologies. By the end, you should be able to describe how they work at a conceptual level and be equipped with the tools to start building similar applications yourself.

The semantic tree: artificial intelligence and machine learning

One bit of advice: it is important to view knowledge as sort of a semantic tree—make sure you understand the fundamental principles, ie the trunk and big branches, before you get into the leaves/details or there is nothing for them to hang on to.—Elon Musk, [Reddit AMA](#)



Machine learning is one of many subfields of artificial intelligence, concerning the ways that computers learn from experience to improve their ability to think, plan, decide, and act.

Artificial intelligence is the study of agents that perceive the world around them, form plans, and make decisions to achieve their goals. Its foundations include mathematics, logic, philosophy, probability, linguistics, neuroscience, and decision theory. Many fields fall under the umbrella of AI, such as computer vision, robotics, machine learning, and natural language processing.

Machine learning is a subfield of artificial intelligence. Its goal is to enable computers to learn on their own. A machine's learning algorithm enables it to identify patterns in observed data, build models that explain the world, and predict things without having explicit pre-programmed rules and models.

The AI effect: what actually qualifies as “artificial intelligence”?

The exact standard for technology that qualifies as “AI” is a bit fuzzy, and interpretations change over time. The AI label tends to describe machines doing tasks traditionally in the domain of humans. Interestingly, once computers figure out how to do one of these tasks, humans have a tendency to say it wasn’t *really* intelligence. This is known as the [AI effect](#).

For example, when IBM’s Deep Blue defeated world chess champion [Garry Kasparov](#) in 1997, people complained that it was using “brute force” methods and it wasn’t “real” intelligence at all. As Pamela McCorduck wrote, *“It’s part of the history of the field of artificial intelligence that every time somebody figured out how to make a computer do something — play good checkers, solve simple but relatively informal problems — there was chorus of critics to say, ‘that’s not thinking’”*([McCorduck, 2004](#)).

Perhaps there is a certain *je ne sais quoi* inherent to what people will reliably accept as “artificial intelligence”:

“AI is whatever hasn't been done yet.” - Douglas Hofstadter

So does a calculator count as AI? Maybe by some interpretation. What about a self-driving car? Today, yes. In the future, perhaps not. Your cool new chatbot startup that automates a flow chart? Sure... why not.

Strong AI will change our world forever; to understand how, studying machine learning is a good place to start

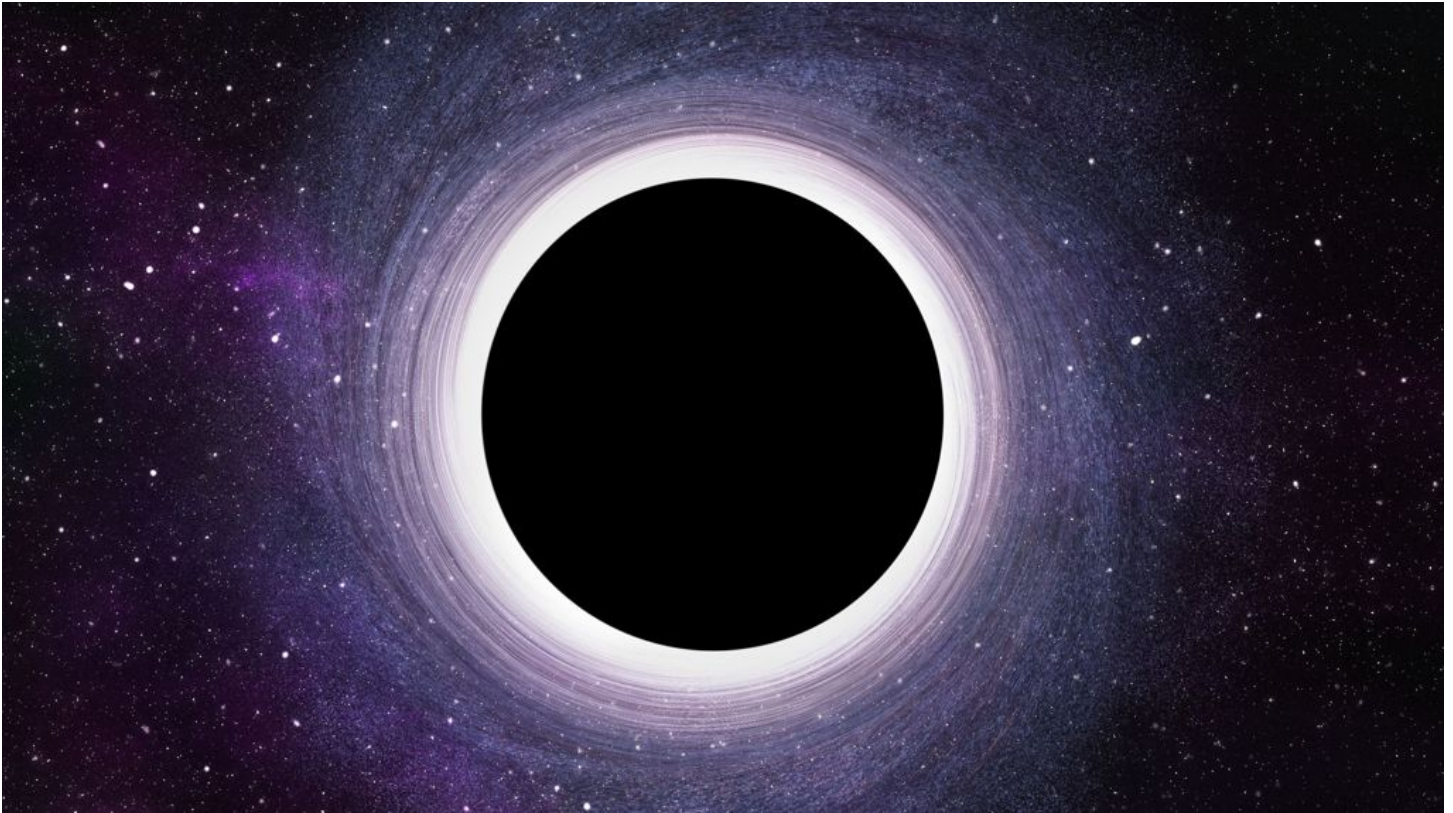
The technologies discussed above are examples of **artificial narrow intelligence (ANI)**, which can effectively perform a narrowly defined task.

Meanwhile, we're continuing to make foundational advances towards human-level **artificial general intelligence (AGI)**, also known as **strong AI**. The definition of an AGI is an artificial intelligence that can successfully perform *any intellectual task that a human being can*, including learning, planning and decision-making under uncertainty, communicating in natural language, making jokes, manipulating people, trading stocks, or... reprogramming itself.

And this last one is a big deal. If we create an AI that can improve itself, it would unlock a cycle of recursive self-improvement that could lead to an **intelligence explosion** over some unknown time period, ranging from many decades to a single day.

Let an ultraintelligent machine be defined as a machine that can far surpass all the intellectual activities of any man however clever. Since the design of machines is one of these intellectual activities, an ultraintelligent machine could design even better machines; there would then unquestionably be an 'intelligence explosion,' and the intelligence of man would be left far behind. Thus the first ultraintelligent machine is the last invention that man need ever make, provided that the machine is docile enough to tell us how to keep it under control.—I.J. Good, 1965

You may have heard this point referred to as the **singularity**. The term is borrowed from the gravitational singularity that occurs at the center of a black hole, an infinitely dense one-dimensional point where the laws of physics as we understand them start to break down.



We have zero visibility into what happens beyond the event horizon of a black hole because no light can escape. Similarly, **after we unlock AI's ability to recursively improve itself, it's impossible to predict what will happen, just as mice who intentionally designed a human might have trouble predicting what the human would do to their world.** Would it keep helping them get more cheese, as they originally intended? (Image via [WIRED](#))

A recent report by the Future of Humanity Institute surveyed a panel of AI researchers on timelines for AGI, and found that **"researchers believe there is a 50% chance of AI outperforming humans in all tasks in 45 years"** ([Grace et al, 2017](#)). We've personally spoken with a number of sane and reasonable AI practitioners who predict much longer timelines (the upper limit being "never"), and others whose timelines are alarmingly short—as little as a few years.

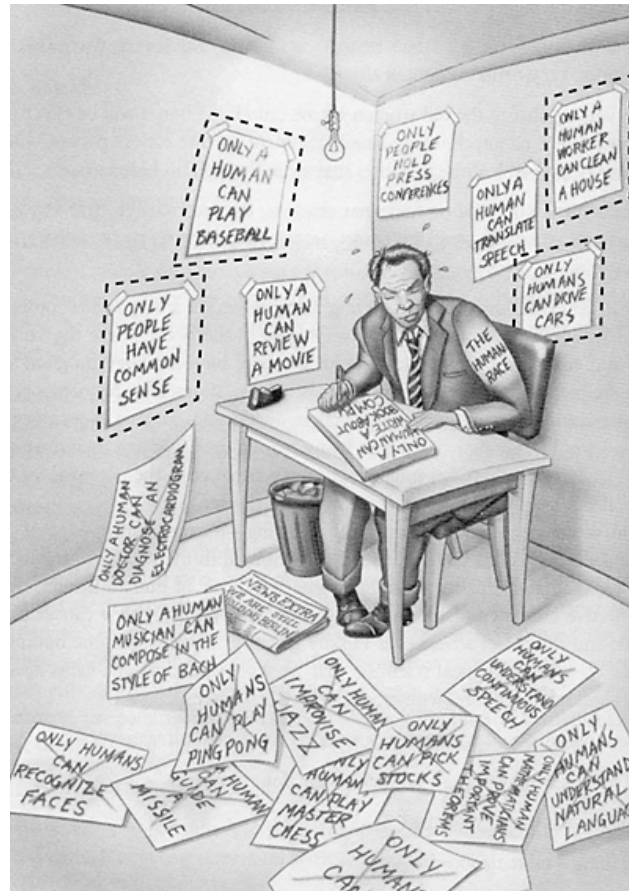


Image from Kurzweil's *The Singularity Is Near*, published in 2005. Now, in 2017, only a couple of these posters could justifiably remain on the wall.

The advent of greater-than-human-level **artificial superintelligence (ASI)** could be one of the best or worst things to happen to our species. It carries with it the immense challenge of specifying what AIs will want in a way that is friendly to humans.

While it's impossible to say what the future holds, one thing is certain: **2017 is a good time to start understanding how machines think.** To go beyond the abstractions of armchair philosophy and intelligently shape our roadmaps and policies with respect to AI, we must engage with the details of how machines see the world—what they “want”, their potential biases and failure modes, their temperamental quirks—just as we study psychology and neuroscience to understand how humans learn, decide, act, and feel.

There are complex, high-stakes questions about AI that will require our careful attention in the coming years.

How can we combat AI's propensity to [further entrench systemic biases](#) evident in existing data sets? What should we make of fundamental [disagreements among the world's most powerful technologists](#) about the potential risks and benefits of artificial intelligence? What are the most promising technical approaches to [teaching AI systems to behave themselves](#)? What will happen to humans' sense of purpose in a world without work?

Machine learning is at the core of our journey towards artificial general intelligence, and in the meantime, it will change every industry and have a massive impact on our day-to-day lives. That's why we believe it's worth understanding machine learning, at least at a conceptual level—and we designed this series to be the best place to start.

How to read this series

You don't necessarily need to read the series cover-to-cover to get value out of it. Here are three suggestions on how to approach it, depending on your interests and how much time you have:

- 1. T-shaped approach.** Read from beginning to end. Summarize each section in your own words as you go (see: [Feynman technique](#)); this encourages active reading & stronger retention. Go deeper into areas that are most relevant to your interests or work. We'll include resources for further exploration at the end of each section.
- 2. Focused approach.** Jump straight to the sections you're most curious about and focus your mental energy there.
- 3. 80/20 approach.** Skim everything in one go, make a few notes on interesting high-level concepts, and call it a night.

About the authors



“Ok, we have to be done with gradient descent by the time we finish this ale.”

@ [The Boozy Cow](#) in Edinburgh

[Vishal](#) most recently led growth at [Upstart](#), a lending platform that utilizes machine learning to price credit, automate the borrowing process, and acquire users. He spends his time thinking about startups, applied cognitive science, moral philosophy, and the ethics of artificial intelligence. (Contact: vishal.maini@gmail.com)

[Samer](#) is a Master’s student in Computer Science and Engineering at UCSD and co-founder of [Conigo Labs](#). Prior to grad school, he founded TableScribe, a business intelligence tool for SMBs, and spent two years advising Fortune 100 companies at McKinsey. Samer previously studied Computer Science and Ethics, Politics, and Economics at Yale. (Contact: samrsabri@gmail.com)

Most of this series was written during a 10-day trip to the United Kingdom in a frantic blur of trains, planes, cafes, pubs and wherever else we could find a dry place to sit. Our aim was to solidify our own understanding of artificial intelligence, machine learning, and how the methods therein fit together—and hopefully create something worth sharing in the process.

And now, without further ado, let’s dive into machine learning with [Part 2.1: Supervised Learning](#)!

Part 2.1: Supervised Learning

The two tasks of supervised learning: regression and classification. Linear regression, loss functions, and gradient descent.

How much money will we make by spending more dollars on digital advertising? Will this loan applicant pay back the loan or not? What's going to happen to the stock market tomorrow?

In supervised learning problems, we start with a data set containing **training examples** with associated correct **labels**. For example, when learning to classify handwritten digits, a supervised learning algorithm takes thousands of pictures of handwritten digits along with labels containing the correct number each image represents. The algorithm will then learn the relationship between the images and their associated numbers, and apply that learned relationship to classify completely new images (without labels) that the machine hasn't seen before. This is how you're able to deposit a check by taking a picture with your phone!

To illustrate how supervised learning works, let's examine the problem of **predicting annual income** based on the number of years of higher education someone has completed. Expressed more formally, we'd like to build a model that approximates the relationship f between the number of years of higher education \mathbf{X} and corresponding annual income \mathbf{Y} .

$$Y = f(X) + \epsilon$$

X (input) = years of higher education

Y (output) = annual income

f = function describing the relationship between X and Y

ε (epsilon) = random error term (positive or negative) with mean zero

Regarding epsilon:

(1) **ε** represents **irreducible error** in the model, which is a theoretical limit around the performance of your algorithm due to inherent noise in the phenomena you are trying to explain. For example, imagine building a model to predict the outcome of a coin flip.

(2) Incidentally, mathematician [Paul Erdős](#) referred to children as “epsilons” because in calculus (but not in stats!) **ε** denotes an arbitrarily small positive quantity. Fitting, no? One method for predicting income would be to create a rigid rules-based model for how income and education are related. For example: “I’d estimate that for every additional year of higher education, annual income increases by \$5,000.”

```
income = ($5,000 * years_of_education) + baseline_income
```

This approach is an example of **engineering** a solution (vs. **learning** a solution, as with the linear regression method described below).

You could come up with a more complex model by including some rules about degree type, years of work experience, school tiers, etc. For example: *“If they completed a Bachelor’s degree or higher, give the income estimate a 1.5x multiplier.”*

But this kind of explicit rules-based programming doesn’t work well with complex data. Imagine trying to design an image classification algorithm made of if-then statements describing the combinations of pixel brightnesses that should be labeled “cat” or “not cat”.

Supervised machine learning solves this problem by getting the computer to *do the work for you*. By identifying patterns in the data, the machine is able to form heuristics. The primary difference between this and human learning is that machine learning runs on computer hardware and is best understood through the lens of computer science and statistics, whereas human pattern-matching happens in a biological brain (while accomplishing the same goals).

In supervised learning, the machine attempts to learn the relationship between income and education from scratch, by running **labeled training data** through a **learning algorithm**. This learned function can be used to estimate the income of people whose income Y is unknown, as long as we have years of education X as inputs. In other words, we can apply our model to the **unlabeled test data** to estimate Y .

The goal of supervised learning is to **predict Y as accurately as possible** when given new examples where X is known and Y is unknown. In what follows we'll explore several of the most common approaches to doing so.

The two tasks of supervised learning: regression and classification

Regression:

Predict a continuous numerical value.
How much will that house sell for?

Classification:

Assign a label. Is this a picture of a cat or a dog?

The rest of this section will focus on regression. In [Part 2.2](#) we'll dive deeper into classification methods.

Regression: predicting a continuous value

Regression predicts a **continuous target variable Y** . It allows you to estimate a value, such as housing prices or human lifespan, based on input data X .

Here, **target variable** means the unknown variable we care about predicting, and **continuous** means there aren't gaps (discontinuities) in the value that Y can take on.

A person's weight and height are continuous values. **Discrete** variables, on the other hand, can only take on a finite number of values—for example, the number of kids somebody has is a discrete variable.

Predicting income is a classic regression problem. Your **input data X** includes all relevant information about individuals in the data set that can be used to predict income, such as years of education, years of work experience, job title, or zip code. These attributes are called **features**, which can be **numerical** (e.g. years of work experience) or **categorical** (e.g. job title or field of study).

You'll want as many training observations as possible relating these features to the target output Y, so that your model can learn the relationship f between X and Y.

The data is split into a **training data set** and a **test data set**. The training set has labels, so your model can learn from these labeled examples. The test set does not have labels, i.e. you don't yet know the value you're trying to predict. It's important that your model can generalize to situations it hasn't encountered before so that it can perform well on the test data.

Regression

$Y = f(X) + \epsilon$, where $X = (x_1, x_2, \dots, x_n)$

Training: machine learns f from labeled training data

Test: machine predicts Y from unlabeled testing data

Note that X can be a **tensor** with an any number of dimensions. A 1D tensor is a vector (1 row, many columns), a 2D tensor is a matrix (many rows, many columns), and then you can have tensors with 3, 4, 5 or more dimensions (e.g. a 3D tensor with rows, columns, and depth). For a review of these terms, see the first few pages of this [linear algebra review](#).

In our trivially simple 2D example, this could take the form of a .csv file where each row contains a person's education level and income. Add more columns with more features and you'll have a more complex, but possibly more accurate, model.

Supervised Learning: Regression

training set	Observation #	Years of Higher Education (X)	Income (Y)
	1	4	\$80,000
	2	5	\$91,500
	3	0	\$42,000
	4	2	\$55,000

	N	6	\$100,000

test set	1	4	???
	2	6	???

Machine Learning for Humans 🧠👤

So how do we solve these problems?

How do we build models that make accurate, useful predictions in the real world? We do so by using **supervised learning algorithms**.

Now let's get to the fun part: getting to know the algorithms. We'll explore some of the ways to approach regression and classification and illustrate key machine learning concepts throughout.

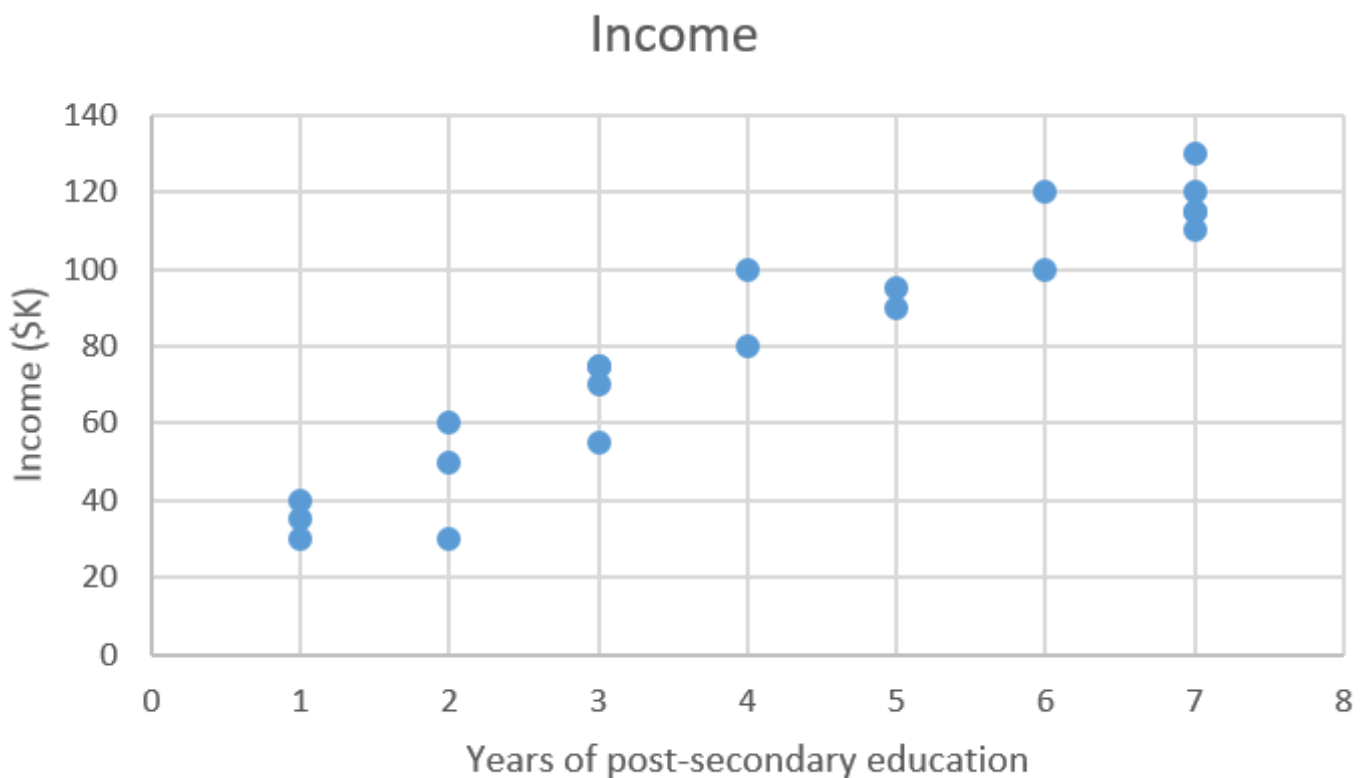
Linear regression (ordinary least squares)

"Draw the line. Yes, this counts as machine learning."

First, we'll focus on solving the income prediction problem with linear regression, since linear models don't work well with image recognition tasks (this is the domain of deep learning, which we'll explore later).

We have our data set X , and corresponding target values Y . The goal of **ordinary least squares (OLS)** regression is to learn a linear model that we can use to predict a new y given a previously unseen x with as little error as possible. We want to guess how much income someone earns based on how many years of education they received.

```
X_train = [4, 5, 0, 2, ..., 6] # years of post-secondary education
Y_train = [80, 91.5, 42, 55, ..., 100] # corresponding annual incomes,
in thousands of dollars
```



Linear regression is a **parametric method**, which means it makes an assumption about the form of the function relating X and Y (we'll cover examples of non-parametric methods later). Our model will be a function that predicts \hat{y} given a specific x:

$$\hat{y} = \beta_0 + \beta_1 * x + \epsilon$$

In this case, we make the explicit assumption that there is a **linear relationship** between X and Y—that is, for each one-unit increase in X, we see a constant increase (or decrease) in Y.

β_0 is the y-intercept and β_1 is the slope of our line, i.e. how much income increases (or decreases) with one additional year of education.

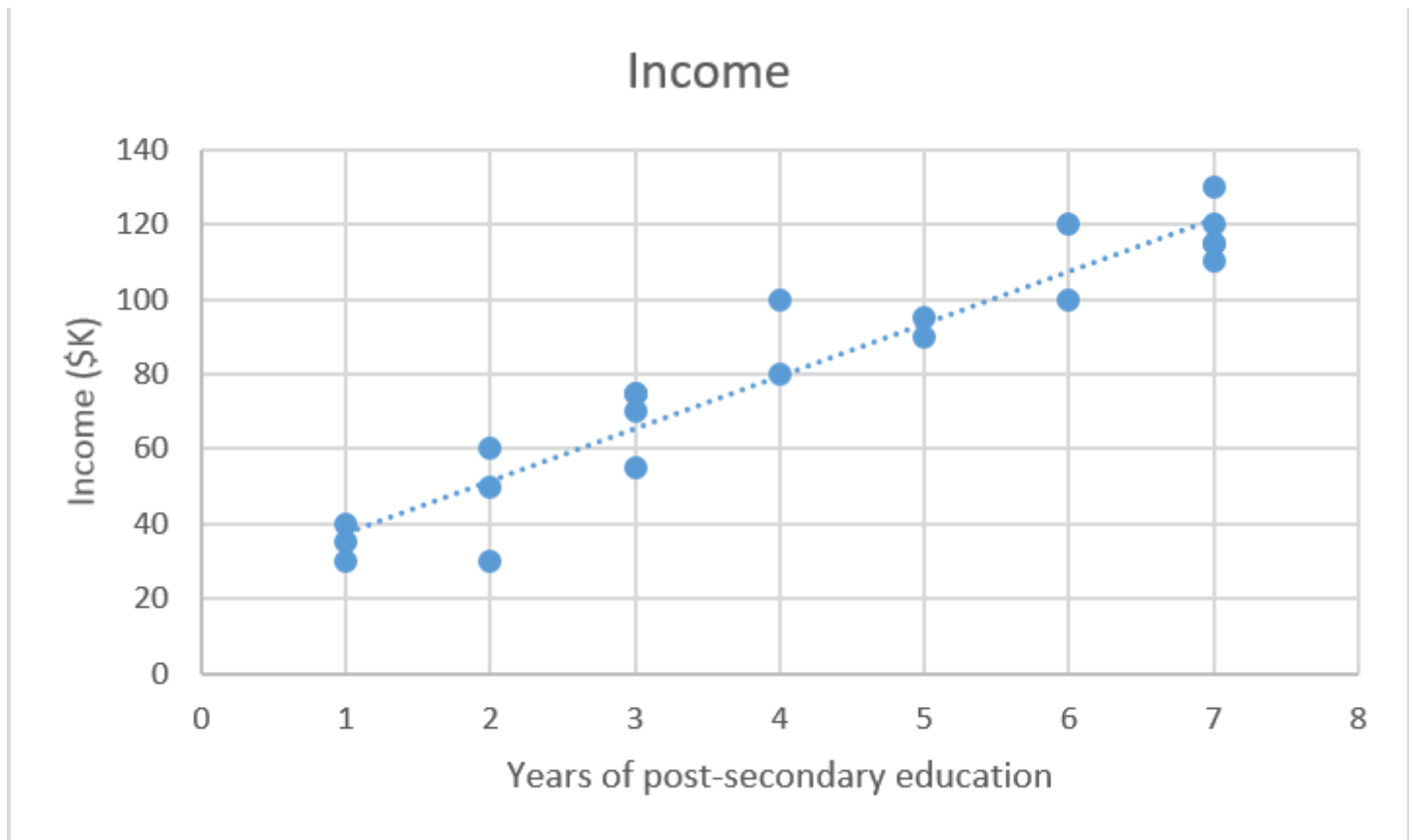
Our goal is to learn the **model parameters** (in this case, β_0 and β_1) that minimize error in the model's predictions.

To find the best parameters:

- ① Define a **cost function**, or **loss function**, that measures how inaccurate our model's predictions are.
- ② Find the parameters that **minimize loss**, i.e. make our model as accurate as possible.

Graphically, in two dimensions, this results in a line of best fit. In three dimensions, we would draw a plane, and so on with higher-dimensional hyperplanes.

A note on dimensionality: our example is two-dimensional for simplicity, but you'll typically have more features (x's) and coefficients (betas) in your model, e.g. when adding more relevant variables to improve the accuracy of your model predictions. The same principles generalize to higher dimensions, though things get much harder to visualize beyond three dimensions.



Mathematically, we look at the difference between each real data point (y) and our model's prediction (\hat{y}). Square these differences to avoid negative numbers and penalize larger differences, and then add them up and take the average. This is a measure of how well our data fits the line.

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n}$$

n = # of observations. Using $2*n$ instead of n makes the math work out more cleanly when taking the derivative to minimize loss, though some stats people say this is blasphemy. When you start having opinions on this kind of stuff, you'll know you are all the way in the rabbit hole.

For a simple problem like this, we can compute a closed form solution using calculus to find the optimal beta parameters that minimize our loss function. But as a cost function grows in complexity, finding a closed form solution with calculus is no longer feasible. This is the motivation for an iterative approach called **gradient descent**, which allows us to minimize a complex loss function.

Gradient descent: learn the parameters

"Put on a blindfold, take a step downhill. You've found the bottom when you have nowhere to go but up."

Gradient descent will come up over and over again, especially in neural networks. Machine learning libraries like [scikit-learn](#) and [TensorFlow](#) use it in the background everywhere, so it's worth understanding the details.

The goal of gradient descent is to find the minimum of our model's loss function by iteratively getting a better and better approximation of it.

Imagine yourself walking through a valley with a blindfold on. Your goal is to find the bottom of the valley. How would you do it?

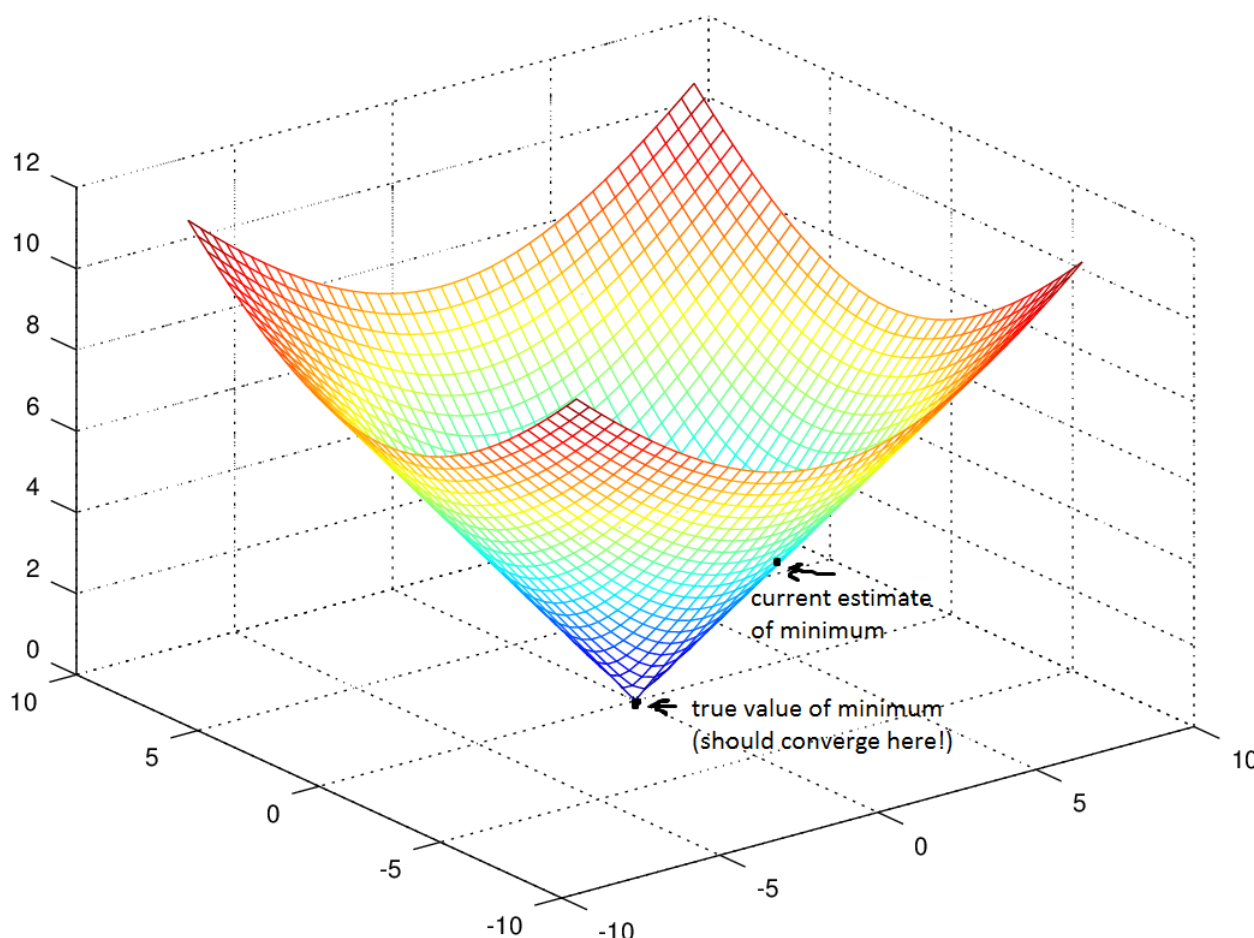
A reasonable approach would be to touch the ground around you and move in whichever direction the ground is sloping down most steeply. Take a step and repeat the same process continually until the ground is flat. Then you know you've reached the bottom of a valley; if you move in any direction from where you are, you'll end up at the same elevation or further uphill.

Going back to mathematics, the ground becomes our loss function, and the elevation at the bottom of the valley is the minimum of that function.

Let's take a look at the loss function we saw in regression:

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n}$$

We see that this is really a function of two variables: β_0 and β_1 . All the rest of the variables are determined, since X , Y , and n are given during training. We want to try to minimize this function.



The function is $f(\beta_0, \beta_1) = z$. To begin gradient descent, you make some guess of the parameters β_0 and β_1 that minimize the function.

Next, you find the **partial derivatives** of the loss function with respect to each beta parameter: $[dz/d\beta_0, dz/d\beta_1]$. A **partial derivative** indicates how much total loss is increased or decreased if you increase β_0 or β_1 by a very small amount.

Put another way, how much would increasing your estimate of annual income assuming zero higher education (β_0) increase the loss (i.e. inaccuracy) of your model? You want

to go in the opposite direction so that you end up walking downhill and minimizing loss.

Similarly, if you increase your estimate of how much each incremental year of education affects income (β_1), how much does this increase loss (z)? If the partial derivative $dz/d\beta_1$ is a negative number, then increasing β_1 is good because it will reduce total loss. If it's a positive number, you want to decrease β_1 . If it's zero, don't change β_1 because it means you've reached an optimum.

Keep doing that until you reach the bottom, i.e. the algorithm **converged** and loss has been minimized. There are lots of tricks and exceptional cases beyond the scope of this series, but generally, this is how you find the optimal **parameters** for your **parametric model**.

Overfitting

Overfitting: "Sherlock, your explanation of what just happened is too specific to the situation." **Regularization:** "Don't overcomplicate things, Sherlock. I'll punch you for every extra word." **Hyperparameter (λ):** "Here's the strength with which I will punch you for every extra word."

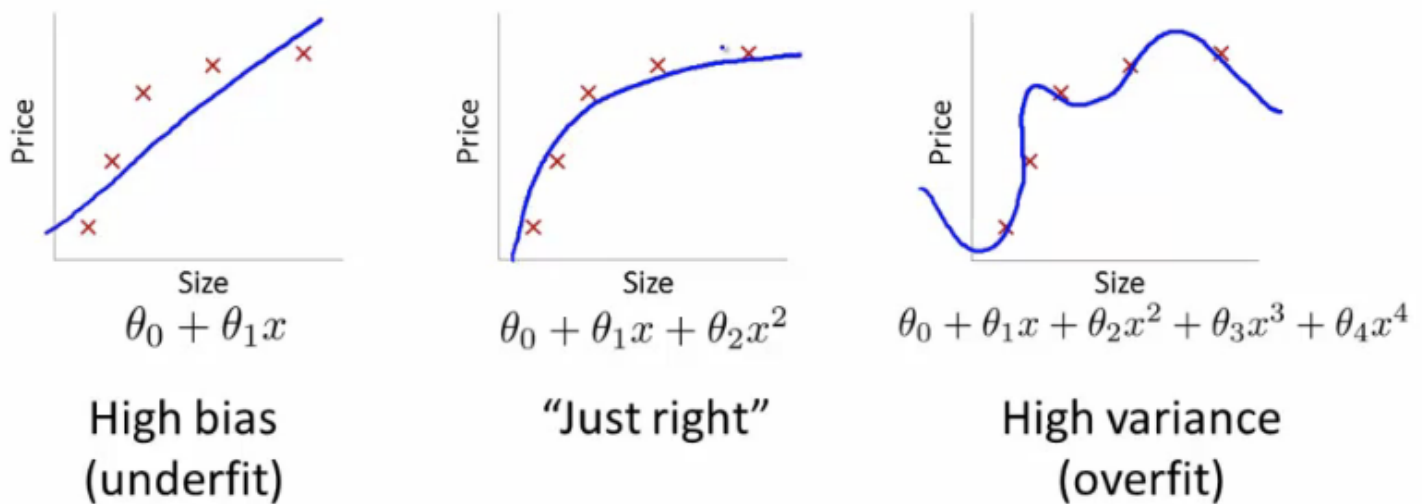
A common problem in machine learning is **overfitting**: learning a function that perfectly explains the training data that the model learned from, but doesn't generalize well to unseen test data. Overfitting happens when a model overlearns from the training data to the point that it starts picking up idiosyncrasies that aren't representative of patterns in the real world. This becomes especially problematic as you make your model increasingly complex. Underfitting is a related issue where your model is not complex enough to capture the underlying trend in the data.

Bias-Variance Tradeoff

Bias is the amount of error introduced by approximating real-world phenomena with a simplified model.

Variance is how much your model's test error changes based on variation in the training data. It reflects the model's sensitivity to the idiosyncrasies of the data set it was trained on.

As a model increases in complexity and it becomes more wiggly (**flexible**), its bias decreases (it does a good job of explaining the training data), but variance increases (it doesn't generalize as well). **Ultimately, in order to have a good model, you need one with low bias and low variance.**



Source: Coursera's [ML course](#), taught by Andrew Ng

Remember that the only thing we care about is how the model performs on test data. You want to predict which emails will be marked as spam before they're marked, not just build a model that is 100% accurate at reclassifying the emails it used to build itself in the first place. Hindsight is 20/20—the real question is whether the lessons learned will help in the future.

The model on the right has zero loss for the training data because it perfectly fits every data point. But the lesson doesn't generalize. It would do a horrible job at explaining a new data point that isn't yet on the line.

Two ways to combat overfitting:

1. Use more training data. *The more you have, the harder it is to overfit the data by learning too much from any single training example.*

2. Use regularization. *Add in a penalty in the loss function for building a model that assigns too much explanatory power to any one feature or allows too many features to be taken into account.*

$$Cost = \frac{\sum_1^n ((\beta_1 x_i + \beta_0) - y_i)^2}{2 * n} + \lambda \sum_{i=0}^1 \beta_i^2$$

The first piece of the sum above is our normal cost function. The second piece is a **regularization term** that adds a penalty for large beta coefficients that give too much explanatory power to any specific feature. With these two elements in place, the cost function now balances between two priorities: explaining the training data and preventing that explanation from becoming overly specific.

The **lambda** coefficient of the regularization term in the cost function is a **hyperparameter**: a general setting of your model that can be increased or decreased (i.e. **tuned**) in order to improve performance. A higher lambda value will more harshly penalize large beta coefficients that could lead to potential overfitting. To decide the best value of lambda, you'd use a method called **cross-validation** which involves holding out a portion of the training data during training, and then seeing how well your model explains the held-out portion. We'll go over this in more depth

Woo! We made it.

Here's what we covered in this section:

- How **supervised machine learning** enables computers to learn from labeled training data without being explicitly programmed
- The tasks of supervised learning: **regression** and **classification**
- **Linear regression**, a bread-and-butter **parametric** algorithm
- Learning **parameters** with **gradient descent**
- **Overfitting** and **regularization**

In the next section—[Part 2.2: Supervised Learning II](#)—we'll talk about two foundational methods of classification: logistic regression and support vector machines.

Practice materials & further reading

2.1a—Linear regression

For a more thorough treatment of linear regression, read chapters 1–3 of [An Introduction to Statistical Learning](#). The book is available for free online and is an excellent resource for understanding machine learning concepts with accompanying exercises.

For more practice:

- Play with the [Boston Housing dataset](#). You can either use software with nice GUIs like Minitab and Excel or do it the hard (but more rewarding) way with [Python](#) or [R](#).
- Try your hand at a [Kaggle](#) challenge, e.g. [housing price prediction](#), and see how others approached the problem after attempting it yourself.

2.1b—Implementing gradient descent

To actually implement gradient descent in Python, check out [this tutorial](#). And [here](#) is a more mathematically rigorous description of the same concepts.

In practice, you'll rarely need to implement gradient descent from scratch, but understanding how it works behind the scenes will allow you to use it more effectively and understand why things break when they do.

Part 2.2: Supervised Learning II

Classification with logistic regression and support vector machines (SVMs).

Classification: predicting a label





Is this email spam or not? Is that borrower going to repay their loan? Will those users click on the ad or not? Who is that person in your Facebook picture?

Classification predicts a **discrete target label Y**. Classification is the problem of assigning new observations to the **class** to which they most likely belong, based on a classification model built from labeled training data.



The accuracy of your classifications will depend on the effectiveness of the algorithm you choose, how you apply it, and how much useful training data you have.

Supervised Learning: Classification

training set

Observation #	Input image (X)	Label (Y)
1		"dog"
2		"cat"
3		"dog"
...
N		"dog"

test set

1		???
2		???

Machine Learning for Humans 🤖🧠

Logistic regression: 0 or 1?

Logistic regression is a method of **classification**: the model outputs the probability of a **categorical** target variable Y belonging to a certain class.

A good example of classification is determining whether a loan application is fraudulent.

Ultimately, the lender wants to know whether they should give the borrower a loan or not, and they have some tolerance for risk that the application is in fact fraudulent. In this case, the goal of logistic regression is to calculate the probability (between 0% and 100%) that the application is fraud. With these probabilities, we can set some threshold above which we're willing to lend to the borrower, and below which we deny their loan application or flag the application for further review.

Though logistic regression is often used for **binary classification** where there are two classes, keep in mind that classification can be performed with any number of categories (e.g. when assigning handwritten digits a label between 0 and 9, or using facial recognition to detect which friends are in a Facebook picture).

Can I just use ordinary least squares?

Nope. If you trained a linear regression model on a bunch of examples where $Y = 0$ or 1 , you might end up predicting some probabilities that are less than 0 or greater than 1, which doesn't make sense. Instead, we'll use a logistic regression model (or **logit model**) which was designed for assigning a probability between 0% and 100% that Y belongs to a certain class.

How does the math work?

Note: the math in this section is interesting but might be on the more technical side. Feel free to skim through it if you're more interested in the high-level concepts. The logit model is a modification of linear regression that makes sure to output a probability between 0 and 1 by applying the **sigmoid function**, which, when graphed, looks like the characteristic S-shaped curve that you'll see a bit later.

$$S(x) = \frac{1}{1 + e^{-x}}.$$

Sigmoid function, which squashes values between 0 and 1.

Recall the original form of our simple linear regression model, which we'll now call $g(x)$ since we're going to use it within a compound function:

$$g(X) = \beta_0 + \beta_1 x + \epsilon$$

Now, to solve this issue of getting model outputs less than 0 or greater than 1, we're going to define a new function $F(g(x))$ that transforms $g(x)$ by squashing the output of linear regression to a value in the $[0,1]$ range. Can you think of a function that does this?

Are you thinking of the sigmoid function? Bam! Presto! You're correct.

So we plug $g(x)$ into the sigmoid function above, resulting in a function of our original function (yes, things are getting meta) that outputs a probability between 0 and 1:

$$P(Y = 1) = F(g(x)) = \frac{1}{1 + e^{-(\beta_0 + \beta_1 * x)}}$$

In other words, we're calculating the probability that the training example belongs to a certain class: $P(Y=1)$.

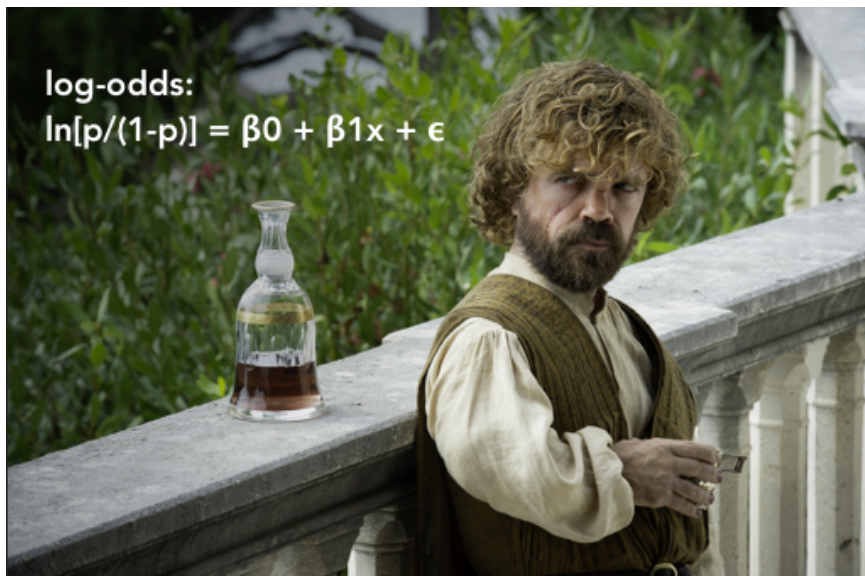
Here we've isolated p , the probability that $Y=1$, on the left side of the equation. If we want to solve for a nice clean $\beta_0 + \beta_1 x + \epsilon$ on the right side so we can straightforwardly interpret the beta coefficients we're going to learn, we'd instead end up with the **log-odds ratio**, or **logit**, on the left side—hence the name “logit model”:

$$\ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x + \epsilon$$

The log-odds ratio is simply the natural log of the **odds ratio**, $p/(1-p)$, which crops up in everyday conversations:

*"Yo, what do you think are the **odds** that Tyrion Lannister dies in this season of Game of Thrones?"*

*"Hmm. It's definitely 2x more likely to happen than not. **2-to-1 odds**. Sure, he might seem too important to be killed, but we all saw what they did to Ned Stark..."*



log-odds:
 $\ln[p/(1-p)] = \beta_0 + \beta_1 x + \epsilon$

← IS HE GONNA DIE?

$p = P(\text{Tyrion dies}) = 2/3$

$1-p = P(\text{Tyrion doesn't die}) = 1/3$

odds ratio: $p/(1-p) = 2.0$
"He's gonna die. 2-to-1 odds"

log-odds ratio: $\ln[p/(1-p)] = 0.693$
"He's gonna die. .693 log-odds"

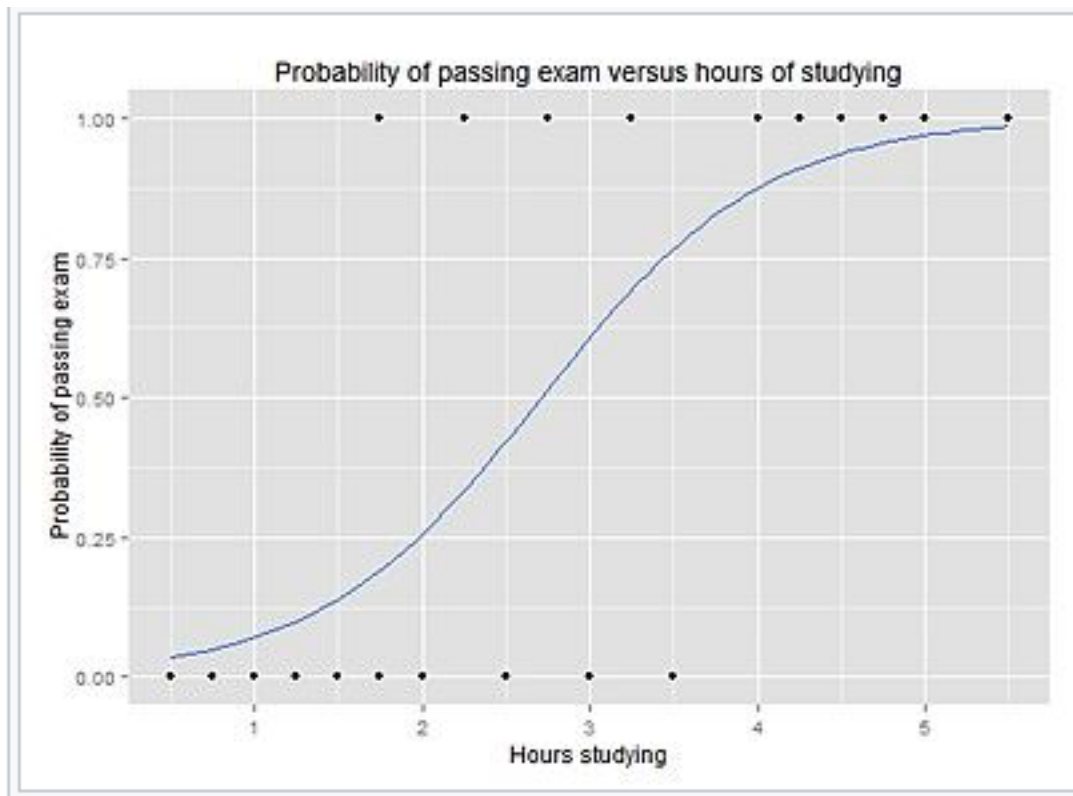
Machine Learning for Humans 🧠🤖

Note that in the logit model, β_1 now represents the **rate of change in the log-odds ratio** as X changes. In other words, it's the "slope of log-odds", not the "slope of the probability".

Log-odds might be slightly unintuitive but it's worth understanding since it will come up again when you're interpreting the output of neural networks performing classification tasks.

Using the output of a logistic regression model to make decisions

The output of the logistic regression model from above looks like an S-curve showing $P(Y=1)$ based on the value of X :



Source: [Wikipedia](#)

To predict the Y label—spam/not spam, cancer/not cancer, fraud/not fraud, etc.—you have to set a probability cutoff, or **threshold**, for a positive result. For example: *“If our model thinks the probability of this email being spam is higher than 70%, label it spam. Otherwise, don’t.”*

The threshold depends on your tolerance for **false positives** vs. **false negatives**. If you’re diagnosing cancer, you’d have a very low tolerance for false negatives, because even if there’s a very small chance the patient has cancer, you’d want to run further tests to make sure. So you’d set a very low threshold for a positive result.

In the case of fraudulent loan applications, on the other hand, the tolerance for false positives might be higher, particularly for smaller loans, since further vetting is costly and a small loan may not be worth the additional operational costs and friction for non-fraudulent applicants who are flagged for further processing.

Minimizing loss with logistic regression

As in the case of linear regression, we use gradient descent to learn the beta parameters that minimize loss.

In logistic regression, the cost function is basically a measure of how often you predicted 1 when the true answer was 0, or vice versa. Below is a **regularized cost function** just like the one we went over for linear regression.

$$Cost(\beta) = \frac{\sum_{i=1}^n (y^i \log(h_{\beta}(x^i)) + (1 - y^i) \log(1 - h_{\beta}(x^i)))}{2 * n} + \lambda \sum_{j=1}^k \beta_j^2$$

Don't panic when you see a long equation like this! Break it into chunks and think about what's going on in each part conceptually. Then the specifics will start to make sense.

The first chunk is the **data loss**, i.e. how much discrepancy there is between the model's predictions and reality. The second chunk is the **regularization loss**, i.e. how much we penalize the model for having large parameters that heavily weight certain features (remember, this prevents overfitting).

We'll minimize this cost function with gradient descent, as above, and voilà! we've built a logistic regression model to make class predictions as accurately as possible.

Support vector machines (SVMs)

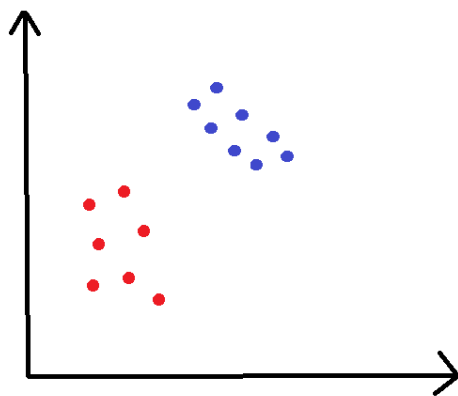
"We're in a room full of marbles again. Why are we always in a room full of marbles? I could've sworn we already lost them."

SVM is the last parametric model we'll cover. It typically solves the same problem as logistic regression—classification with two classes—and yields similar performance. It's worth understanding because the algorithm is geometrically motivated in nature, rather than being driven by probabilistic thinking.

A few examples of the problems SVMs can solve:

- Is this an image of a cat or a dog?
- Is this review positive or negative?
- Are the dots in the 2D plane red or blue?

We'll use the third example to illustrate how SVMs work. Problems like these are called **toy problems** because they're not real—but nothing is real, so it's fine.



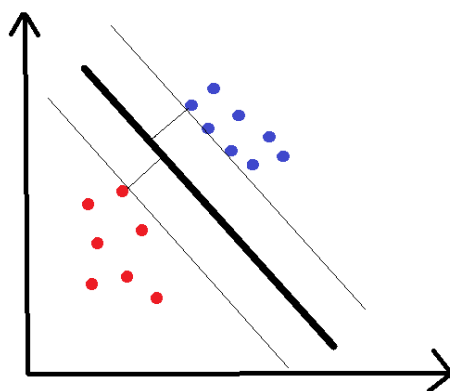
In this example, we have points in a 2D space that are either red or blue, and we'd like to cleanly separate the two.

The training set is plotted the graph above. We would like to classify new, unclassified points in this plane. To do this, SVMs use a separating line (or, in more than two dimensions, a multi-dimensional **hyperplane**) to split the space into a red zone and

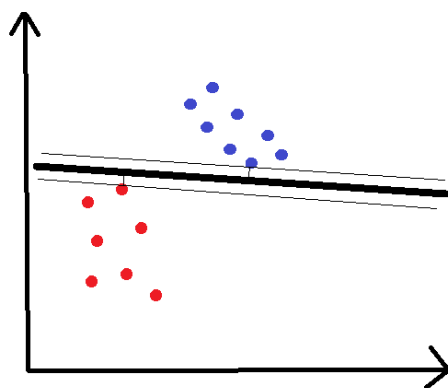
a blue zone. You can already imagine how a separating line might look in the graph above.

How, specifically, do we choose where to draw the line?

Below are two examples of such a line:



These charts were made with Microsoft Paint, which was [deprecated a few weeks ago](#) after 32 wonderful years. RIP Paint :(



Hopefully, you share the intuition that the first line is superior. The distance to the nearest point on either side of the line is called the **margin**, and SVM tries to **maximize the margin**. You can think about it like a safety space: the bigger that space, the less likely that noisy points get misclassified.

Based on this short explanation, a few big questions come up.

1. How does the math behind this work?

We want to find the optimal hyperplane (a line, in our 2D example). This hyperplane needs to (1) separate the data cleanly, with blue points on one side of the line and red points on the other side, and (2) maximize the margin. This is an **optimization** problem. The solution has to respect constraint (1) while maximizing the margin as is required in (2).

The human version of solving this problem would be to take a ruler and keep trying different lines separating all the points until you get the one that maximizes the margin.

It turns out there's a clean mathematical way to do this maximization, but the specifics are beyond our scope. To explore it further, here's a [video](#) lecture that shows how it works using [Lagrangian Optimization](#).

The solution hyperplane you end up with is defined in relation to its position with respect to certain x_i 's, which are called the support vectors, and they're usually the ones closest to the hyperplane.

2. What happens if you can't separate the data cleanly?

There are two methods for dealing with this problem.

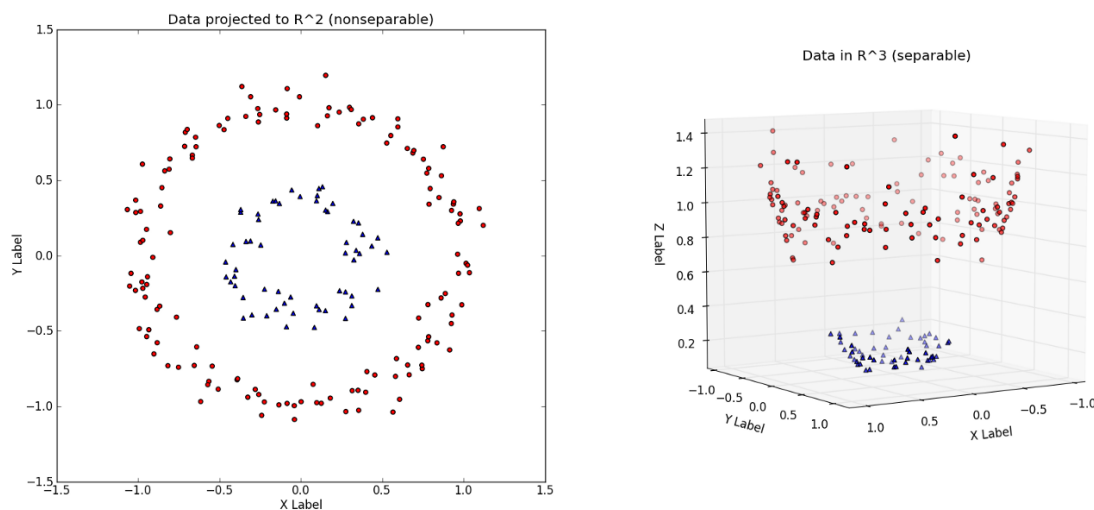
2.1. Soften the definition of "separate".

We allow a few mistakes, meaning we allow some blue points in the red zone or some red points in the blue zone. We do that by adding a cost C for misclassified examples in our loss function. Basically, we say it's acceptable but costly to misclassify a point.

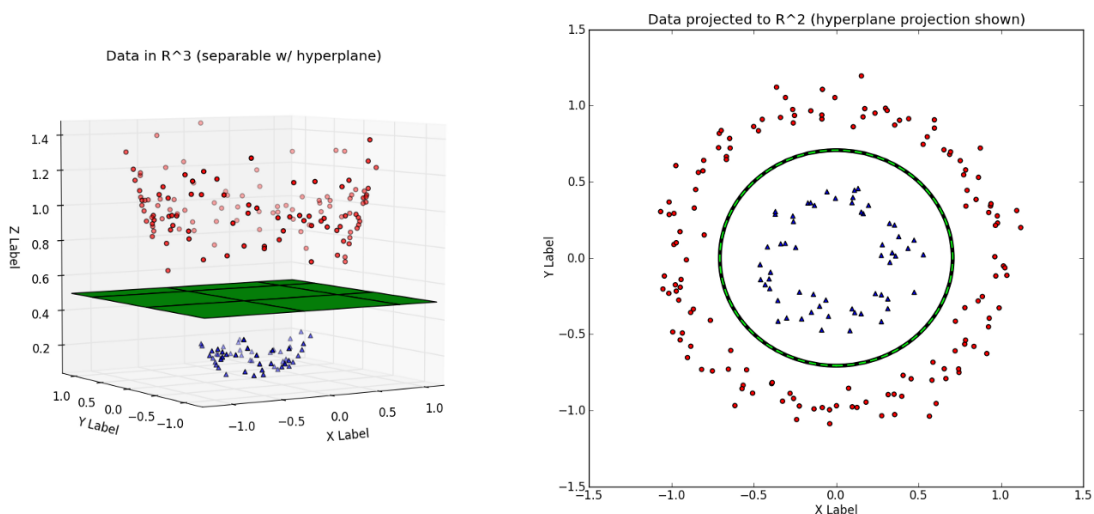
2.2. Throw the data into higher dimensions.

We can create nonlinear classifiers by increasing the number of dimensions, i.e. include x^2 , x^3 , even $\cos(x)$, etc. Suddenly, you have boundaries that can look more squiggly when we bring them back to the lower dimensional representation.

Intuitively, this is like having red and blue marbles lying on the ground such that they can't be cleanly separated by a line—but if you could make all the red marbles levitate off the ground in just the right way, you could draw a plane separating them. Then you let them fall back to the ground knowing where the blues stop and reds begin.



A nonseparable dataset in a two-dimensional space R^2 , and the same dataset mapped onto three dimensions with the third dimension being x^2+y^2 (source: http://www.eric-kim.net/eric-kim-net/posts/1/kernel_trick.html)



The decision boundary is shown in green, first in the three-dimensional space (left), then back in the two-dimensional space (right). Same source as previous image.

In summary, SVMs are used for classification with two classes. They attempt to find a plane that separates the two classes cleanly. When this isn't possible, we either soften

the definition of “separate,” or we throw the data into higher dimensions so that we can cleanly separate the data.

Success!

In this section we covered:

- The **classification** task of supervised learning
- Two foundational classification methods: **logistic regression** and **support vector machines (SVMs)**
- Recurring concepts: the **sigmoid** function, **log-odds (“logit”)**, and **false positives** vs. **false negatives**,

In [Part 2.3: Supervised Learning III](#), we’ll go into **non-parametric** supervised learning, where the ideas behind the algorithms are very intuitive and performance is excellent for certain kinds of problems, but the models can be harder to interpret.

Practice materials & further reading

2.2a—Logistic regression

Data School has an excellent [in-depth guide to logistic regression](#). We’ll also continue to refer you to [An Introduction to Statistical Learning](#). See Chapter 4 on logistic regression, and Chapter 9 on support vector machines.

To implement logistic regression, we recommend working on [this problem set](#). You have to register on the site to work through it, unfortunately. C’est la vie.

2.2b—Down the SVM rabbit hole

To dig into the math behind SVMs, watch Prof. Patrick Winston’s [lecture](#) from MIT 6.034: Artificial Intelligence. And check out [this tutorial](#) to work through a Python implementation.

Part 2.3: Supervised Learning III

Non-parametric models: k-nearest neighbors, decision trees, and random forests. Introducing cross-validation, hyperparameter tuning, and ensemble models.

Non-parametric learners.

Things are about to get a little... wiggly.

In contrast to the methods we've covered so far—linear regression, logistic regression, and SVMs where the form of the model was pre-defined—**non-parametric learners** do not have a model structure specified a priori. We don't speculate about the form of the function f that we're trying to learn before training the model, as we did previously with linear regression. Instead, the model structure is purely determined from the data.

These models are more flexible to the shape of the training data, but this sometimes comes at the cost of interpretability. This will make more sense soon. Let's jump in.

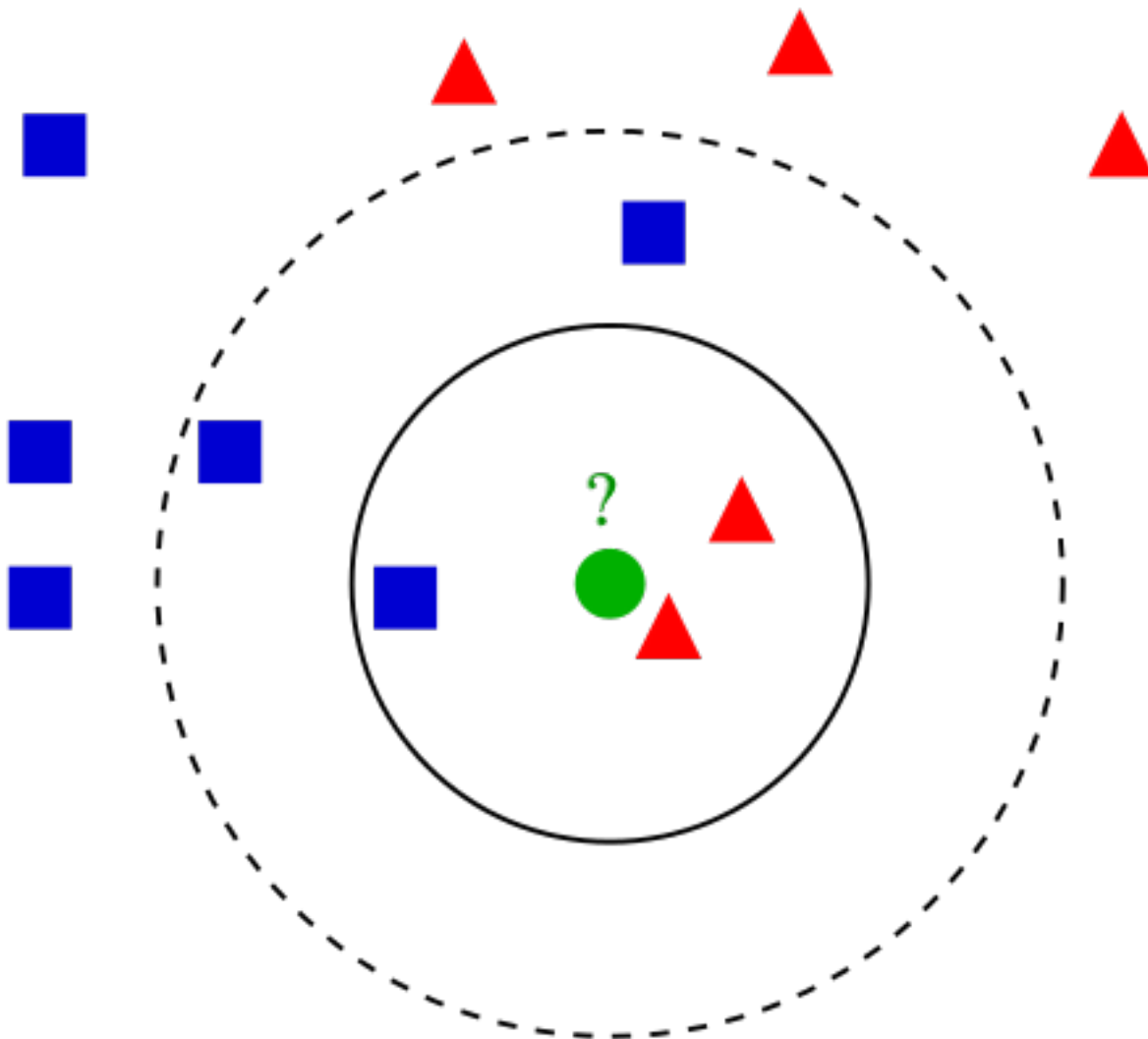
k-nearest neighbors (k-NN)

"You are the average of your k closest friends."

k-NN seems almost too simple to be a machine learning algorithm. The idea is to label a test data point x by finding the **mean** (or **mode**) of the k closest data points' labels.

Take a look at the image below. Let's say you want to figure out whether Mysterious Green Circle is a Red Triangle or a Blue Square. What do you do?

You could try to come up with a fancy equation that looks at where Green Circle lies on the coordinate plane below and makes a prediction accordingly. Or, you could just look at its three nearest neighbors, and guess that Green Circle is probably a Red Triangle. You could also expand the circle further and look at the five **nearest neighbors**, and make a prediction that way (3/5 of its five nearest neighbors are Blue Squares, so we'd guess that Mysterious Green Circle is a Blue Square when $k=5$).



k-NN illustration with $k=1$, 3, and 5. To classify the Mysterious Green Circle (x) above, look at its single nearest neighbor, a "Red Triangle". So, we'd guess that $\hat{y} = \text{"Red Triangle"}$. With $k=3$, look at the 3 nearest neighbors: the mode of these is again "Red Triangle" so $\hat{y} = \text{"Red Triangle"}$. With $k=5$, we take the mode of the 5 nearest neighbors instead. Now, notice that \hat{y} becomes "Blue Square". Image from [Wikipedia](#).

That's it. That's **k-nearest neighbors**. You look at the k closest data points and take the average of their values if variables are continuous (like housing prices), or the mode if they're categorical (like cat vs. dog).

If you wanted to guess unknown house prices, you could just take the average of some number of geographically nearby houses, and you'd end up with some pretty nice guesses. These might even outperform a parametric regression model built by some economist that estimates model coefficients for # of beds/baths, nearby schools, distance to public transport, etc.

How to use k-NN to predict housing prices:

1) Store the training data, a matrix X of features like zip code, neighborhood, # of bedrooms, square feet, distance from public transport, etc., and a matrix Y of corresponding sale prices.

2) Sort the houses in your training data set by similarity to the house in question, based on the features in X . We'll define "similarity" below.

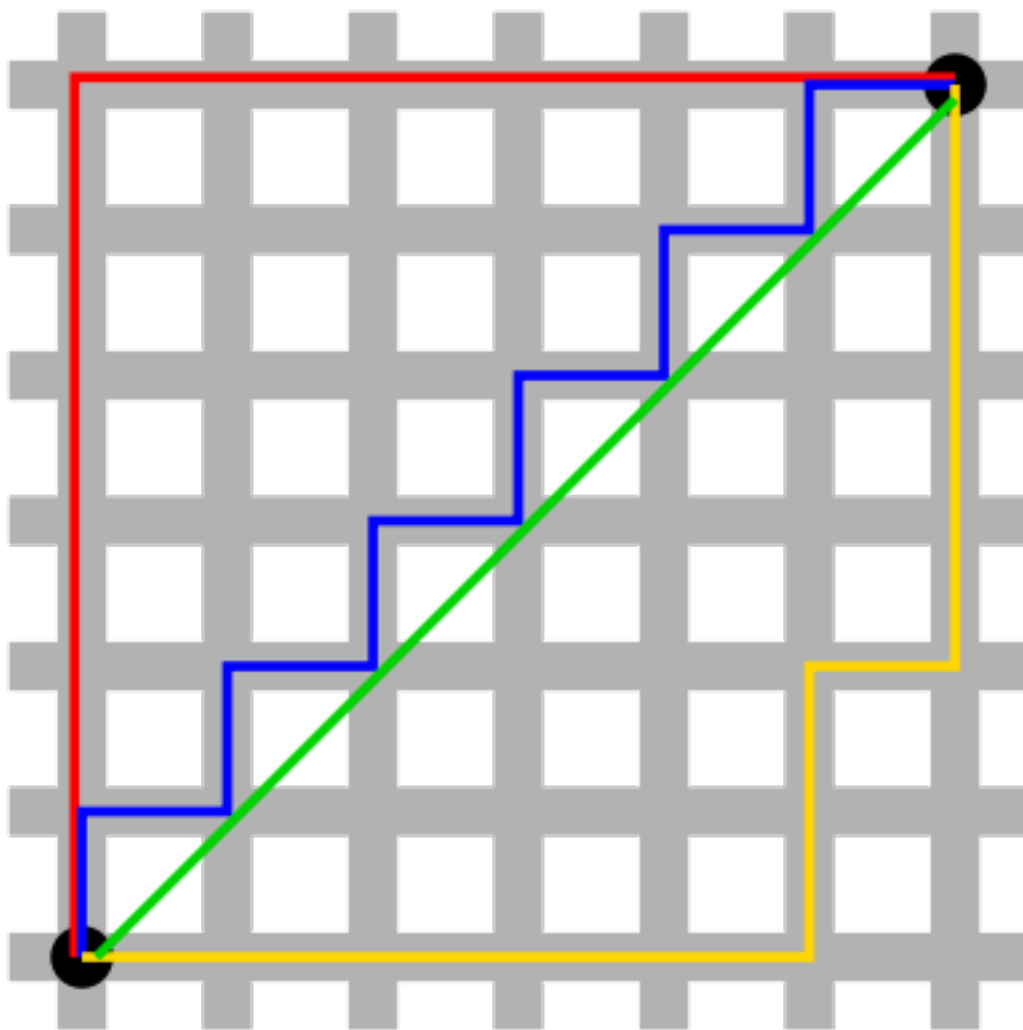
3) Take the mean of the k closest houses. That is your guess at the sale price (i.e. \hat{y})

The fact that k-NN doesn't require a pre-defined parametric function $f(X)$ relating Y to X makes it well-suited for situations where the relationship is too complex to be expressed with a simple linear model.

Distance metrics: defining and calculating "nearness"

How do you calculate distance from the data point in question when finding the "nearest neighbors"? How do you mathematically determine which of the Blue Squares and Red Triangles in the example above are closest to Green Circle, especially if you can't just draw a nice 2D graph and eyeball it?

The most straightforward measure is **Euclidean distance** (a straight line, “as the crow flies”). Another is Manhattan distance, like walking city blocks. You could imagine that Manhattan distance is more useful in a model involving fare calculation for Uber drivers, for example.



Green line = Euclidean distance. Blue line = Manhattan distance. Source: [Wikipedia](#)

Remember the Pythagorean theorem for finding the length of the hypotenuse of a right triangle?

$$a^2 + b^2 = c^2.$$

c = length of hypotenuse (green line above). a and b = length of the other sides, at a right angle (red lines above).

Solving in terms of c , we find the length of the hypotenuse by taking the square root of the sum of squared lengths of a and b , where a and b are **orthogonal** sides of the triangle (i.e. they are at a 90-degree angle from one another, going in perpendicular directions in space).

$$c = \sqrt{a^2 + b^2}.$$

This idea of finding the length of the hypotenuse given vectors in two orthogonal directions generalizes to many dimensions, and this is how we derive the formula for Euclidean distance $\mathbf{d}(\mathbf{p}, \mathbf{q})$ between points \mathbf{p} and \mathbf{q} in n -dimensional space:

$$\begin{aligned}\mathbf{d}(\mathbf{p}, \mathbf{q}) &= \mathbf{d}(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2} \\ &= \sqrt{\sum_{i=1}^n (q_i - p_i)^2}.\end{aligned}$$

Formula for Euclidean distance, derived from the Pythagorean theorem.

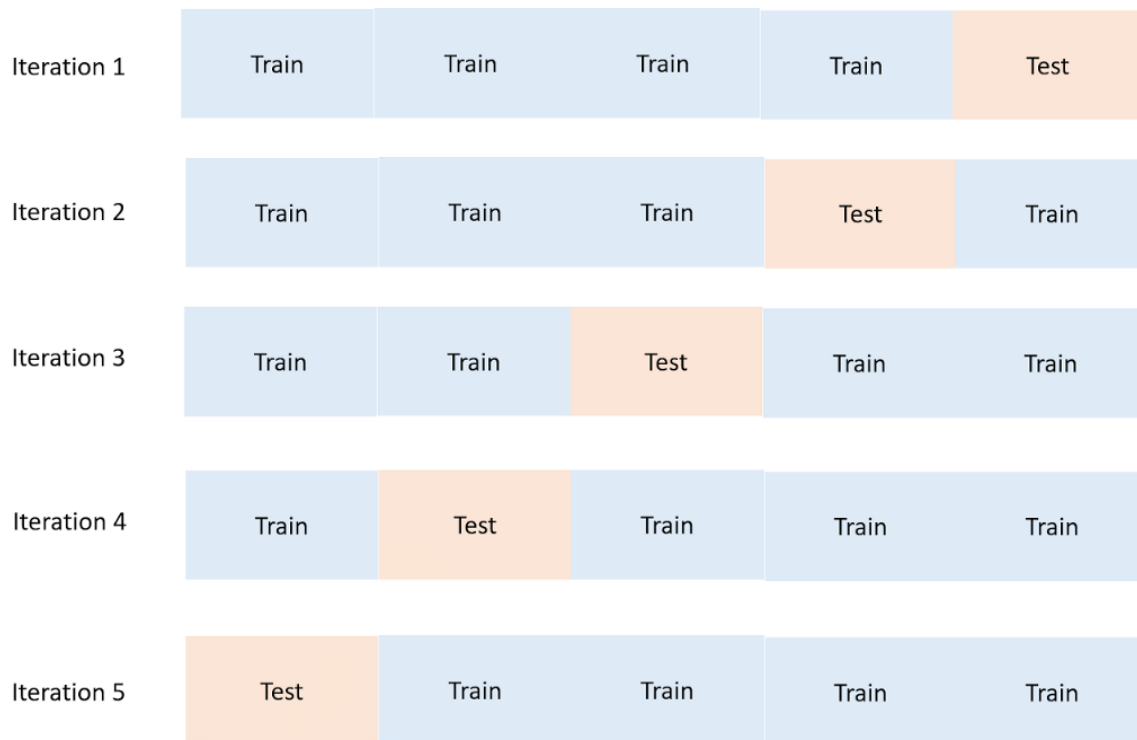
With this formula, you can calculate the nearness of all the training data points to the data point you're trying to label, and take the mean/mode of the k nearest neighbors to make your prediction.

Typically you won't need to calculate any distance metrics by hand—a quick Google search reveals pre-built functions in NumPy or SciPy that will do this for you, e.g. `euclidean_dist = numpy.linalg.norm(p-q)`—but it's fun to see how geometry concepts from eighth grade end up being helpful for building ML models today!

Choosing k: tuning hyperparameters with cross-validation

To decide which value of k to use, you can test different k -NN models using different values of k with **cross-validation**:

1. Split your training data into segments, and train your model on all but one of the segments; use the held-out segment as the “test” data.
2. See how your model performs by comparing your model’s predictions (\hat{y}) to the actual values of the test data (y).
3. Pick whichever yields the lowest error, on average, across all iterations.



Cross-validation illustrated. The number of splits and iterations can be varied.

Higher k prevents overfitting

Higher values of k help address overfitting, but if the value of k is too high your model will be very **biased** and **inflexible**. To take an extreme example: if $k = N$ (the total number of data points), the model would just dumbly blanket-classify all the test data as the mean or mode of the training data.

If the single most common animal in a data set of animals is a Scottish Fold kitten, k -NN with k set to N (the # of training observations) would then predict that every other animal in the world is also a Scottish Fold kitten. Which, in Vishal's opinion, would be awesome. Samer disagrees.



Completely gratuitous Scottish Fold .gif. We'll call it a study break. 😊

Where to use k-NN in the real world

Some examples of where you can use k-NN:

- **Classification: fraud detection.** The model can update virtually instantly with new training examples since you're just storing more data points, which allows quick adaptation to new methods of fraud.
- **Regression: predicting housing prices.** In housing price prediction, literally being a "near neighbor" is actually a good indicator of being similar in price. k-NN is useful in domains where physical proximity matters.
- **Imputing missing training data.** If one of the columns in your .csv has lots of missing values, you can impute the data by taking the mean or mode. k-NN could give you a somewhat more accurate guess at each missing value.

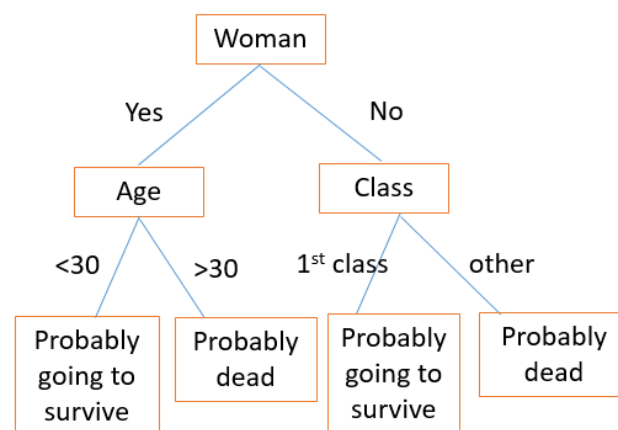
Decision trees, random forests

Making a good decision tree is like playing a game of "20 questions".

A normal tree



A decision tree



The decision tree on the right describes survival patterns on the Titanic.

The first **split** at the **root** of a decision tree should be like the first question you should ask in 20 questions: you want to separate the data as cleanly as possible, thereby maximizing **information gain** from that split.

If your friend says *"I'm thinking of a noun, ask me up to 20 yes/no questions to guess what it is"* and your first question is *"is it a potato?"*, then you're a dumbass, because they're going to say no and you gained almost no information. Unless you happen to know your friend thinks about potatoes all the time, or is thinking about one right now. Then you did a great job.

Instead, a question like *"is it an object?"* might make more sense.

This is kind of like how hospitals triage patients or approach differential diagnoses. They ask a few questions up front and check some basic vitals to determine if you're going to die imminently or something. They don't start by doing a biopsy to check if you have pancreatic cancer as soon as you walk in the door.

There are ways to quantify information gain so that you can essentially evaluate every possible split of the training data and maximize information gain for every split. This way you can predict every label or value as efficiently as possible.

Now, let's look at a particular data set and talk about how we choose splits.

The Titanic dataset

Kaggle has a Titanic [dataset](#) that is used for a lot of machine learning intros. When the titanic sunk, 1,502 out of 2,224 passengers and crew were killed. Even though there was some luck involved, women, children, and the upper-class were more likely to survive. If you look back at the decision tree above, you'll see that it somewhat reflects this variability across gender, age, and class.

Choosing splits in a decision tree

Entropy is the amount of disorder in a set (measured by [Gini index](#) or [cross-entropy](#)). If the values are really mixed, there's lots of entropy; if you can cleanly split values, there's no entropy. For every split at a parent node, you want the child nodes to be as pure as possible—minimize entropy. For example, in the Titanic, gender is a big determinant of survival, so it makes sense for this feature to be used in the first split as it's the one that leads to the most information gain.

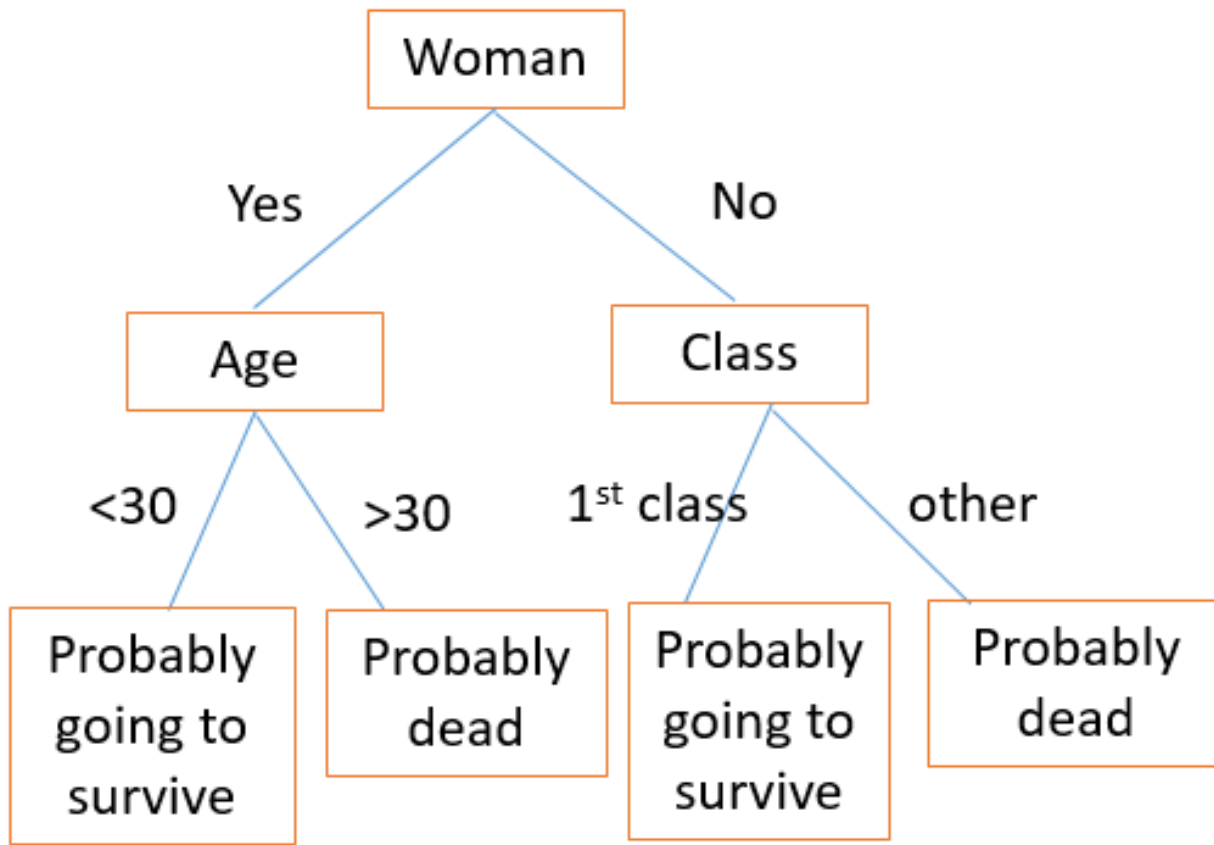
Let's take a look at our Titanic variables:

Data Dictionary

Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

Source: [Kaggle](#)

We build a tree by picking one of these variables and splitting the dataset according to it.



The first split separates our dataset into men and women. Then, the women branch gets split again in age (the split that minimizes entropy). Similarly, the men branch gets split by class. By following the tree for a new passenger, you can use the tree to make a guess at whether they died.

The Titanic example is solving a classification problem ("survive" or "die"). If we were using decision trees for regression—say, to predict housing prices—we would create splits on the most important features that determine housing prices. How many square feet: more than or less than ____? How many bedrooms & bathrooms: more than or less than ____?

Then, during testing, you would run a specific house through all the splits and take the average of all the housing prices in the final **leaf node** (bottom-most node) where the house ends up as your prediction for the sale price.

There are a few hyperparameters you can tune with decision trees models, including `max_depth` and `max_leaf_nodes`. See the [scikit-learn module](#) on decision trees for advice on defining these parameters.

Decision trees are effective because they are easy to read, powerful even with messy data, and computationally cheap to deploy once after training. Decision trees are also good for handling mixed data (numerical or categorical).

That said, decision trees are computationally expensive to train, carry a big risk of overfitting, and tend to find local optima because they can't go back after they have made a split. To address these weaknesses, we turn to a method that illustrates the power of combining many decision trees into one model.

Random forest: an ensemble of decision trees

A model comprised of many models is called an ensemble model, and this is usually a winning strategy.

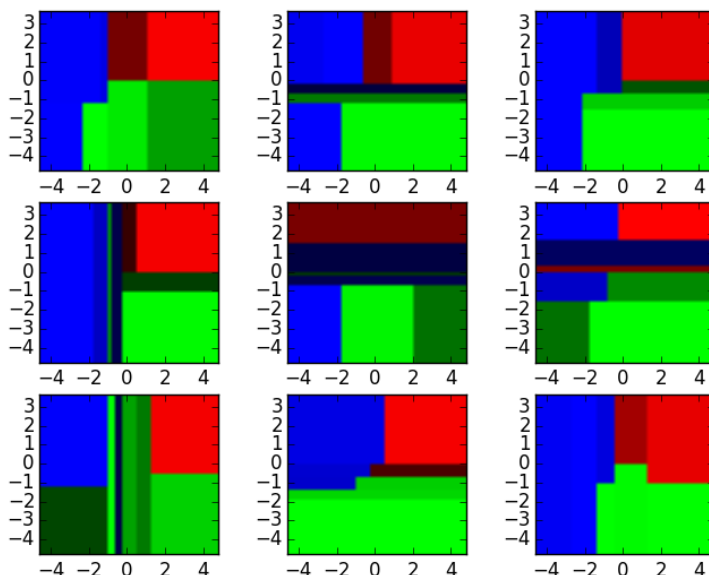
A single decision tree can make a lot of wrong calls because it has very black-and-white judgments. A **random forest** is a meta-estimator that aggregates many decision trees, with some helpful modifications:

1. The number of features that can be split on at each node is limited to some percentage of the total (this is a hyperparameter you can choose—see scikit-learn documentation for details). This ensures that the ensemble model does not rely too heavily on any individual feature, and makes fair use of all potentially predictive features.

2. Each tree draws a random sample from the original data set when generating its splits, adding a further element of randomness that prevents overfitting.

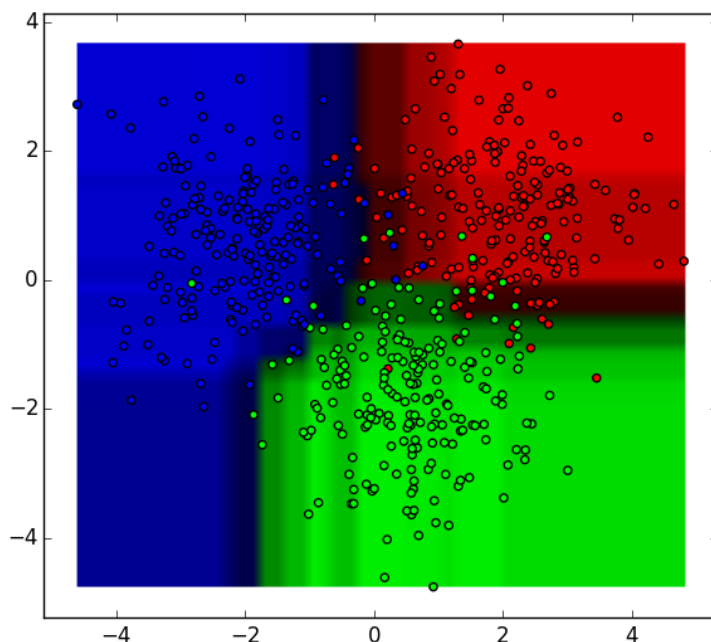
These modifications also prevent the trees from being too highly correlated. Without #1 and #2 above, every tree would be identical, since recursive binary splitting is deterministic.

To illustrate, see these nine decision tree classifiers below.



Source: <http://xenon.stanford.edu/~jianzh/ml/>

These decision tree classifiers can be aggregated into a random forest ensemble which combines their input. Think of the horizontal and vertical axes of each decision tree output as features x_1 and x_2 . At certain values of each feature, the decision tree outputs a classification of "blue", "green", "red", etc.



Source: <http://xenon.stanford.edu/~jianzh/ml/>

These results are aggregated, through modal votes or averaging, into a single ensemble model that ends up outperforming any individual decision tree's output.

Random forests are an excellent starting point for the modeling process, since they tend to have strong performance with a high tolerance for less-cleaned data and can be useful for figuring out which features actually matter among many features.

There are many other clever ensemble models that combine decision trees and yield excellent performance—check out [XGBoost](#) (Extreme Gradient Boosting) as an example.

And with that, we conclude our study of supervised learning!

Nice work. In this section we've covered:

- Two **non-parametric** supervised learning algorithms: **k-NN** and **decision trees**
- Measures of **distance** and **information gain**
- **Random forests**, which are an example of an **ensemble model**
- **Cross-validation** and **hyperparameter tuning**

Hopefully, you now have some solid intuitions for how we learn f given a training data set and use this to make predictions with the test data.

Next, we'll talk about how to approach problems where we don't have any labeled training data to work with, in [Part 3: Unsupervised Learning](#).

Practice materials & further reading

2.3a—Implementing k-NN

Try this [walkthrough](#) for implementing k-NN from scratch in Python. You may also want to take a look at the [scikit-learn](#) documentation to get a sense of how pre-built implementations work.

2.3b—Decision trees

Try the decision trees lab in Chapter 8 of [An Introduction to Statistical Learning](#). You can also play with the [Titanic](#) dataset, and check out this [tutorial](#) which covers the same concepts as above with accompanying code. Here is the [scikit-learn implementation](#) of random forest for out-of-the-box use on data sets.

Part 3: Unsupervised Learning

Clustering and dimensionality reduction: k-means clustering, hierarchical clustering, principal component analysis (PCA), singular value decomposition (SVD)

How do you find the underlying structure of a dataset? How do you summarize it and group it most usefully? How do you effectively represent data in a compressed format? These are the goals of unsupervised learning, which is called “unsupervised” because you start with **unlabeled data** (there’s no Y).

The two unsupervised learning tasks we will explore are clustering the data into groups by similarity and reducing dimensionality to compress the data while maintaining its structure and usefulness.

Examples of where unsupervised learning methods might be useful:

- An advertising platform segments the U.S. population into smaller groups with similar demographics and purchasing habits so that advertisers can reach their target market with relevant ads.
- Airbnb groups its housing listings into neighborhoods so that users can navigate listings more easily.
- A data science team reduces the number of dimensions in a large data set to simplify modeling and reduce file size.

In contrast to supervised learning, it's not always easy to come up with metrics for how well an unsupervised learning algorithm is doing. "Performance" is often subjective and domain-specific.

Clustering

An interesting example of clustering in the real world is marketing data provider Acxiom's life stage clustering system, Personix. This service segments U.S. households into 70 distinct clusters within 21 life stage groups that are used by advertisers when targeting Facebook ads, display ads, direct mail campaigns, etc.

1Y STARTING OUT Cluster 39 Setting Goals Cluster 45 Offices & Entertainment Cluster 57 Collegiate Crowd Cluster 58 Outdoor Fervor Cluster 67 First Steps 2Y TAKING HOLD Cluster 18 Climbing the Ladder Cluster 21 Children First Cluster 24 Career Building Cluster 30 Out & About 3Y SETTLING DOWN Cluster 34 Outward Bound	8X LARGE HOUSEHOLDS Cluster 11 Schools & Shopping Cluster 12 On the Go Cluster 19 Country Comfort Cluster 27 Tenured Proprietors 9B COMFORTABLE INDEPENDENCE Cluster 29 City Mixers Cluster 35 Working & Active Cluster 56 Metro Active 10B RURAL-METRO MIX Cluster 47 Rural Parents Cluster 53 Metro Strivers Cluster 60 Rural & Mobile	15M TOP WEALTH Cluster 2 Established Elite Cluster 3 Corporate Connected 16M LIVING WELL Cluster 14 Career Centered Cluster 15 Country Ways Cluster 23 Good Neighbors 17M BARGAIN HUNTERS Cluster 43 Work & Causes Cluster 44 Open Houses Cluster 55 Community Life Cluster 63 Staying Home Cluster 68 Staying Healthy
---	---	---

A selection of Personix demographic clusters

Their white paper reveals that they used **centroid clustering** and **principal components analysis**, both of which are techniques covered in this section.

You can imagine how having access to these clusters is extremely useful for advertisers who want to (1) understand their existing customer base and (2) use their ad spend effectively by targeting potential new customers with relevant demographics, interests, and lifestyles.

What's My Cluster ?



Cluster #24: Career Building

Career Building singles are young, but well compensated. While repaying their education loans they are beginning to save and invest. They favor trendy stores that cater to their age range, incomes and aspirations, such as Express, H&M and Sephora. They enjoy new technology, and read magazines on mobile devices. They visit The Apple Store.

They use the Internet extensively for entertainment news, music, podcasts and services. Sports are important, too, either as a fan or a participant. They listen to football, watch MMA and have fun skiing and playing volleyball.

You can actually find out which cluster you personally would belong to by answering a few simple questions in Acxiom's "What's My Cluster?" tool.

Let's walk through a couple of clustering methods to develop intuition for how this task can be performed.

k-means clustering

"And k rings were given to the race of Centroids, who above all else, desire power."

The goal of clustering is to create groups of data points such that points in different clusters are dissimilar while points within a cluster are similar.

With **k-means clustering**, we want to cluster our data points into k groups. A larger k creates smaller groups with more granularity, a lower k means larger groups and less granularity.

The output of the algorithm would be a set of "labels" assigning each data point to one of the k groups. In k-means clustering, the way these groups are defined is by creating a **centroid** for each group. The centroids are like the heart of the cluster, they "capture" the points closest to them and add them to the cluster.

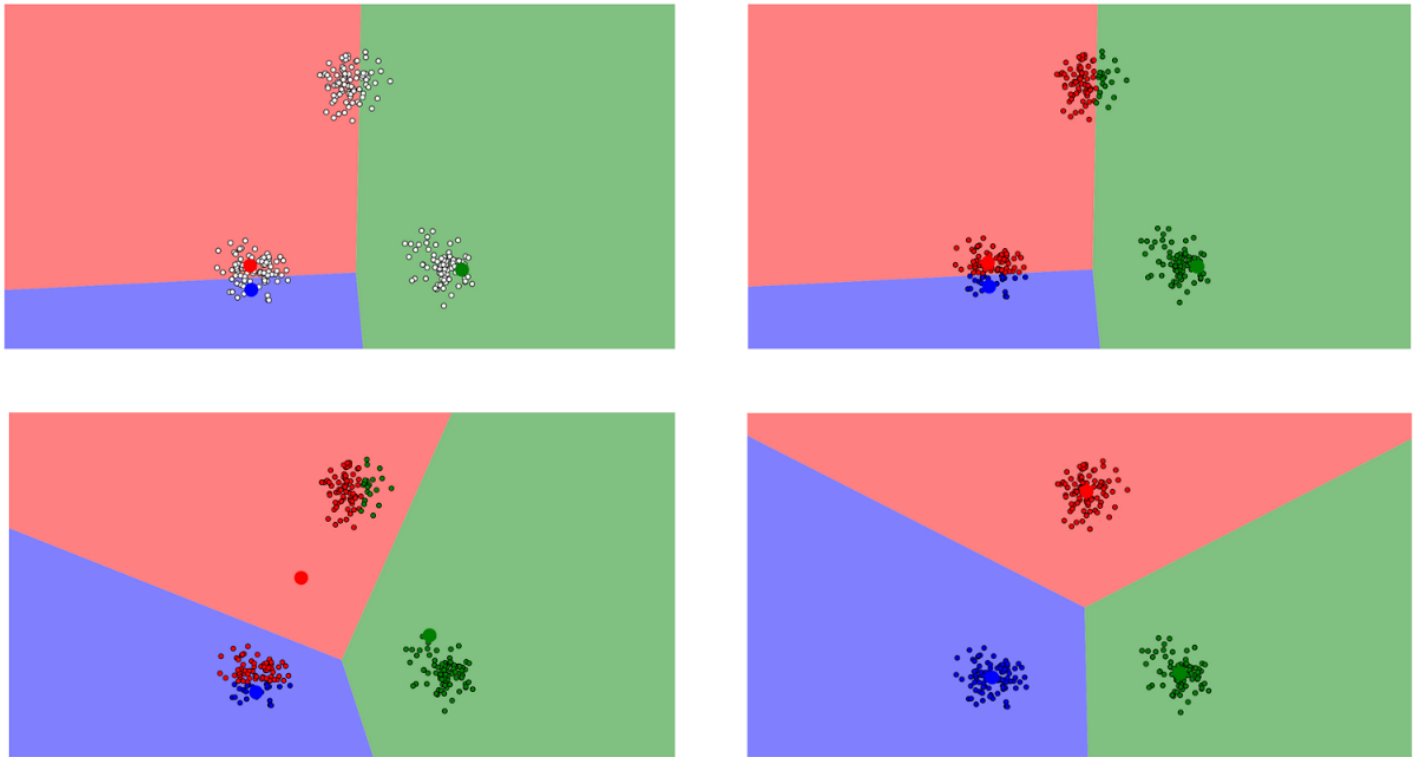
Think of these as the people who show up at a party and soon become the centers of attention because they're so magnetic. If there's just one of them, everyone will gather around; if there are lots, many smaller centers of activity will form.

Here are the steps to k-means clustering:

1. Define the k centroids. Initialize these at random (there are also fancier algorithms for initializing the centroids that end up converging more effectively).
2. Find the closest centroid & update cluster assignments. Assign each data point to one of the k clusters. Each data point is assigned to the nearest centroid's cluster. Here, the measure of "nearness" is a hyperparameter — often Euclidean distance.
3. Move the centroids to the center of their clusters. The new position of each centroid is calculated as the average position of all the points in its cluster.

Keep repeating steps 2 and 3 until the centroid stop moving a lot at each iteration (i.e., until the algorithm converges)

That, in short, is how k-means clustering works! Check out this [visualization](#) of the algorithm—read it like a comic book. Each point in the plane is colored according the centroid that it is closest to at each moment. You'll notice that the centroids (the larger blue, red, and green circles) start randomly and then quickly adjust to capture their respective clusters.



Another real-life application of k-means clustering is classifying handwritten digits. Suppose we have images of the digits as a long vector of pixel brightnesses. Let's say the images are black and white and are 64x64 pixels. Each pixel represents a dimension. So the world these images live in has $64 \times 64 = 4,096$ dimensions. In this 4,096-dimensional world, k-means clustering allows us to group the images that are close together and assume they represent the same digit, which can achieve [pretty good results](#) for digit recognition.

Hierarchical clustering

"Let's make a million options become seven options. Or five. Or twenty? Meh, we can decide later."

Hierarchical clustering is similar to regular clustering, except that you're aiming to build a hierarchy of clusters. This can be useful when you want flexibility in how many clusters you ultimately want. For example, imagine grouping items on an online marketplace like Etsy or Amazon. On the homepage you'd want a few broad categories of items for

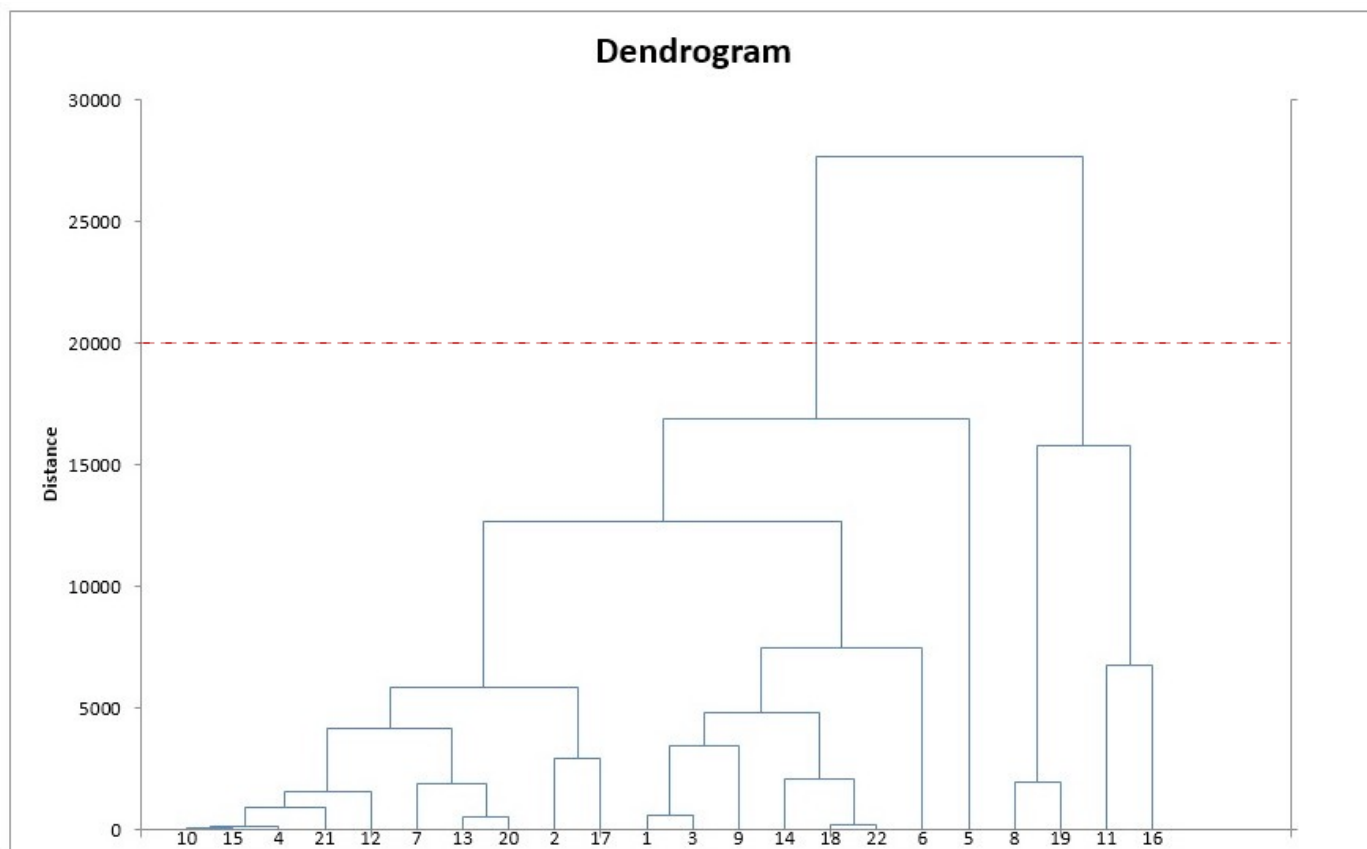
simple navigation, but as you go into more specific shopping categories you'd want increasing levels of granularity, i.e. more distinct clusters of items.

In terms of outputs from the algorithm, in addition to cluster assignments you also build a nice tree that tells you about the hierarchies between the clusters. You can then pick the number of clusters you want from this tree.

Here are the steps for hierarchical clustering:

- 1. Start with N clusters,** one for each data point.
- 2. Merge the two clusters that are closest to each other.** Now you have $N-1$ clusters.
- 3. Recompute the distances between the clusters.** There are several ways to do this (see this tutorial for more details). One of them (called average-linkage clustering) is to consider the distance between two clusters to be the average distance between all their respective members.
- 4. Repeat steps 2 and 3 until you get one cluster of N data points.** You get a tree (also known as a dendrogram) like the one below.
- 5. Pick a number of clusters and draw a horizontal line in the dendrogram.**

For example, if you want $k=2$ clusters, you should draw a horizontal line around "distance=20000." You'll get one cluster with data points 8, 9, 11, 16 and one cluster with the rest of the data points. In general, the number of clusters you get is the number of intersection points of your horizontal line with the vertical lines in the dendrogram.



Source: [Solver.com](https://solver.com). For more detail on hierarchical clustering, you can check [this video](#) out.

Dimensionality reduction

"It is not the daily increase, but the daily decrease. Hack away at the unessential."—Bruce Lee

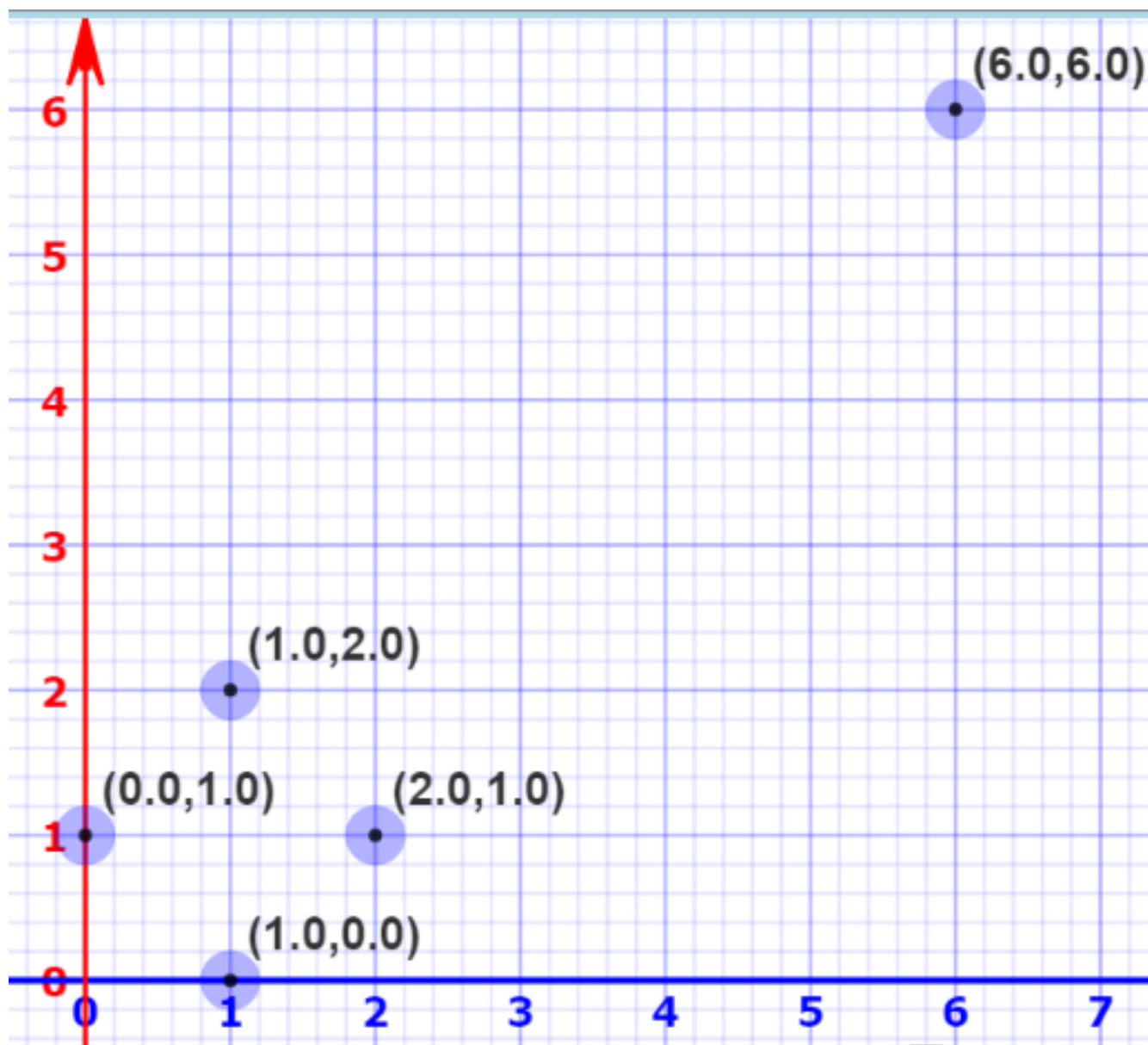
Dimensionality reduction looks a lot like compression. This is about trying to reduce the complexity of the data while keeping as much of the relevant structure as possible. If you take a simple 128 x 128 x 3 pixels image (length x width x RGB value), that's 49,152 dimensions of data. If you're able to reduce the dimensionality of the space in which these images live without destroying too much of the meaningful content in the images, then you've done a good job at dimensionality reduction.

We'll take a look at two common techniques in practice: **principal component analysis** and **singular value decomposition**.

Principal component analysis (PCA)

First, a little linear algebra refresher—let's talk about **spaces** and **bases**.

You're familiar with the coordinate plane with origin $O(0,0)$ and **basis vectors** $i(1,0)$ and $j(0,1)$. It turns out you can choose a completely different basis and still have all the math work out. For example, you can keep O as the origin and choose the basis to vectors $i'=(2,1)$ and $j'=(1,2)$. If you have the patience for it, you'll convince yourself that the point labeled $(2,2)$ in the i', j' coordinate system is labeled $(6, 6)$ in the i, j system.



Plotted using Mathisfun's "Interactive Cartesian Coordinates"

This means we can change the basis of a space. Now imagine much higher-dimensional space. Like, 50K dimensions. You can select a basis for that space, and then select only the 200 most significant vectors of that basis. These basis vectors are called **principal components**, and the subset you select constitute a new space that is smaller in dimensionality than the original space but maintains as much of the complexity of the data as possible.

To select the most significant principal components, we look at how much of the data's variance they capture and order them by that metric.

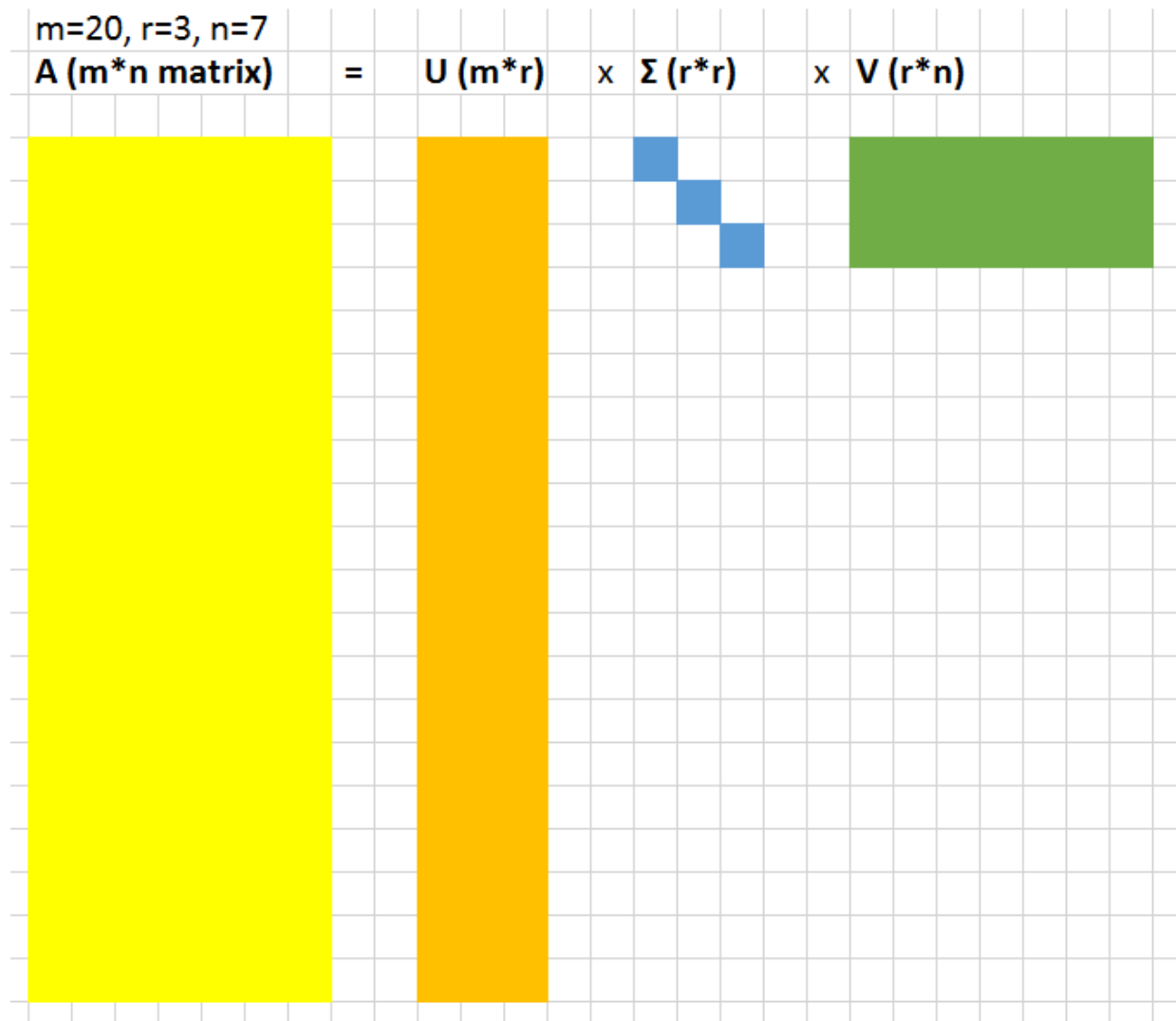
Another way of thinking about this is that PCA remaps the space in which our data exists to make it more compressible. The transformed dimension is smaller than the original dimension.

By making use of the first several dimensions of the remapped space only, we can start gaining an understanding of the dataset's organization. This is the promise of dimensionality reduction: reduce complexity (dimensionality in this case) while maintaining structure (variance). Here's a [fun paper](#) Samer wrote on using PCA (and diffusion mapping, another technique) to try to make sense of the Wikileaks cable release.

Singular value decomposition (SVD)

Let's represent our data like a big $A = m \times n$ matrix. SVD is a computation that allows us to decompose that big matrix into a product of 3 smaller matrices ($U = m \times r$, diagonal matrix $\Sigma = r \times r$, and $V = r \times n$ where r is a small number).

Here's a more visual illustration of that product to start with:

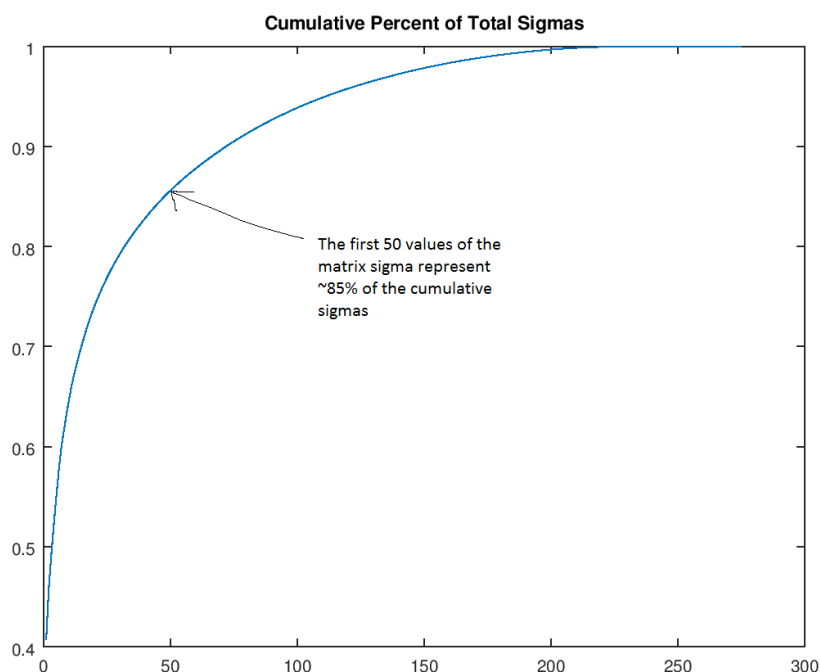


The values in the $r \times r$ diagonal matrix Σ are called singular values. What's cool about them is that these singular values can be used to compress the original matrix. If you drop the smallest 20% of singular values and the associated columns in matrices U and V , you save quite a bit of space and still get a decent representation of the underlying matrix.

To examine what that means more precisely, let's work with this image of a dog:



We'll use the code written in Andrew Gibiansky's post on SVD. First, we show that if we rank the singular values (the values of the matrix Σ) by magnitude, the first 50 singular values contain 85% of the magnitude of the whole matrix Σ .



We can use this fact to discard the next 250 values of sigma (i.e., set them to 0) and just keep a “rank 50” version of the image of the dog. Here, we create a rank 200, 100, 50, 30, 20, 10, and 3 dog. Obviously, the picture is smaller, but let’s agree that the rank 30 dog is still good. Now let’s see how much compression we achieve with this dog. The original image matrix is $305 \times 275 = 83,875$ values. The rank 30 dog is $305 \times 30 + 30 + 30 \times 275 = 17,430$ —almost 5 times fewer values with very little loss in image quality. The reason for the calculation above is that we also discard the parts of the matrix U and V that get multiplied by zeros when the operation $U\Sigma'V$ is carried out (where Σ' is the modified version of Σ that only has the first 30 values in it).

Full-Rank Dog Rank 200 Dog



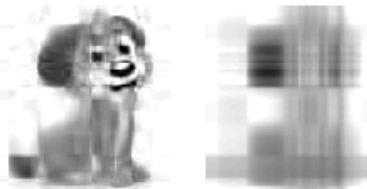
Rank 100 Dog Rank 50 Dog



Rank 30 Dog Rank 20 Dog



Rank 10 Dog Rank 3 Dog



Unsupervised learning is often used to preprocess the data. Usually, that means compressing it in some meaning-preserving way like with PCA or SVD before feeding it to a deep neural net or another supervised learning algorithm.

Onwards!

Now that you've finished this section, you've earned an awful, horrible, never-to-be-mentioned-again joke about unsupervised learning. Here goes...

Person-in-joke-#1: Y would u ever need to use unsupervised tho?

Person-in-joke-#2: Y? there's no Y.

Next up... [Part 4: Neural Networks & Deep Learning!](#)

Practice materials & further reading

3a—k-means clustering

Play around with this clustering [visualization](#) to build intuition for how the algorithm works. Then, take a look at this implementation of [k-means clustering for handwritten digits](#) and the associated tutorial.

3b—SVD

For a good reference on SVD, go no further than Andrew Gibiansky's [post](#).

Part 4: Neural Networks & Deep Learning

Where, why, where, and how deep neural networks work. Drawing inspiration from the brain. Convolutional neural networks (CNNs) and recurrent neural networks (RNNs). Real-world applications.

With deep learning, we're still learning a function f to map input X to output Y with minimal loss on the test data, just as we've been doing all along. Recall our initial "problem statement" from [Part 2.1](#) on supervised learning:

$$Y = f(X) + \epsilon$$

Training: machine learns f from labeled training data

Testing: machine predicts Y from unlabeled testing data

The real world is messy, so sometimes f is complicated. In natural language problems large vocabulary sizes mean lots of features. Vision problems involve lots of visual information about pixels. Playing games requires making a decision based on complex scenarios with many possible futures. The learning techniques we've covered so far do well when the data we're working with is not insanely complex, but it's not clear how they'd generalize to scenarios like these.

Deep learning is really good at learning f , particularly in situations where the data is complex. In fact, artificial neural networks are known as **universal function approximators** because they're able to [learn any function](#), no matter how wiggly, with just a single **hidden layer**.

Let's look at the problem of image classification. We take an image as an input, and output a class (e.g., dog, cat, car).

Graphically, a deep neural network solving image classification looks something like this:

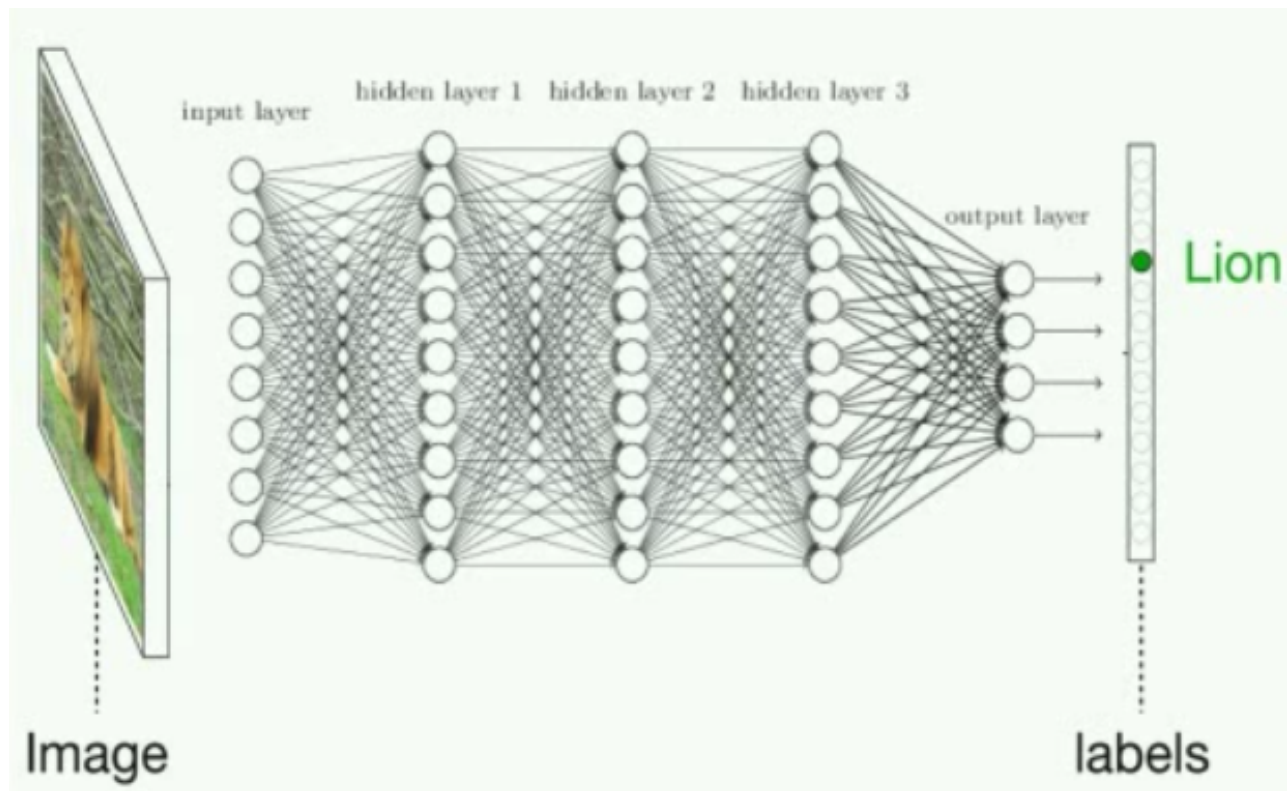


Image from Jeff Clune's 1-hour [Deep Learning Overview](#) on YouTube

But really, this is a giant mathematical equation with millions of terms and lots of parameters. The input X is, say, a greyscale image represented by a w -by- h matrix of pixel brightnesses. The output Y is a vector of class probabilities. This means we have as an output the probability of each class being the correct label. If this neural net is working well, the highest probability should be for the correct class. And the layers in the middle are just doing a bunch of matrix multiplication by summing **activations** \times **weights** with non-linear transformations (**activation functions**) after every hidden layer to enable the network to learn a **non-linear function**.

Incredibly, you can use **gradient descent** in the exact same way that we did with linear regression in [Part 2.1](#) to train these parameters in a way that minimizes loss. So with a lot of examples and a lot of gradient descent, the model can learn how to classify images of animals correctly. And that, in a nutshell's nutshell, is "deep learning".

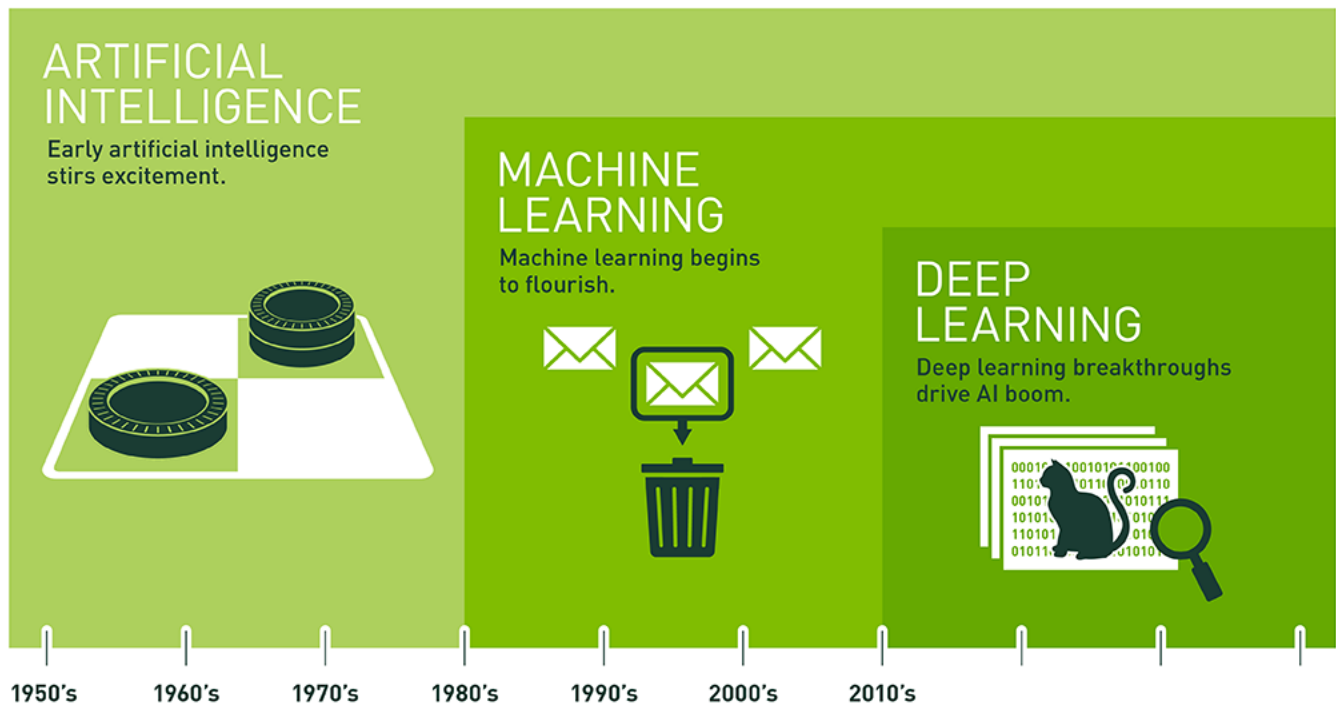
Where deep learning does well, and some history

Artificial neural networks have actually been around for a long time. Their application has been historically referred to as **cybernetics** (1940s-1960s), **connectionism** (1980s-1990s), and then came into vogue as **deep learning** circa 2006 when neural networks started getting, well, “deeper” (Goodfellow et al., 2016). But only recently have we really started to scratch the surface of their full potential.

As described by Andrej Karpathy (Director of AI at Tesla, whom we tend to think of as the Shaman of Deep Learning), there are generally “four separate factors that hold back AI:

- 1. Compute (the obvious one: Moore’s Law, GPUs, ASICs),*
- 2. Data (in a nice form, not just out there somewhere on the internet—e.g. ImageNet),*
- 3. Algorithms (research and ideas, e.g. backprop, CNN, LSTM), and*
- 4. Infrastructure (software under you—Linux, TCP/IP, Git, ROS, PR2, AWS, AMT, TensorFlow, etc.)” (Karpathy, 2016).*

In the past decade or so, the full potential of deep learning is finally being unlocked by advances in (1) and (2), which in turn has led to further breakthroughs in (3) and (4)—and so the cycle continues, with exponentially more humans rallying to the frontlines of deep learning research along the way (just think about what you’re doing right now!)



Since an early flush of optimism in the 1950s, smaller subsets of artificial intelligence – first machine learning, then deep learning, a subset of machine learning – have created ever larger disruptions.

Illustration by [NVIDIA](#), a leading maker of graphics processing units (GPUs) which were originally built for for gaming but turned out to be well-suited to the type of [parallel computing](#) required by deep neural networks

In the rest of this section, we'll provide some background from biology and statistics to explain what happens inside neural nets, and then talk through some amazing applications of deep learning. Finally, we'll link to a few resources so you can apply deep learning yourself, even sitting on the couch in your pajamas with a laptop, to quickly achieve greater-than-human-level performance on certain types of problems.

Drawing inspiration from the brain (or is it just statistics?)—what happens inside neural nets

Neurons, feature learning, and layers of abstraction

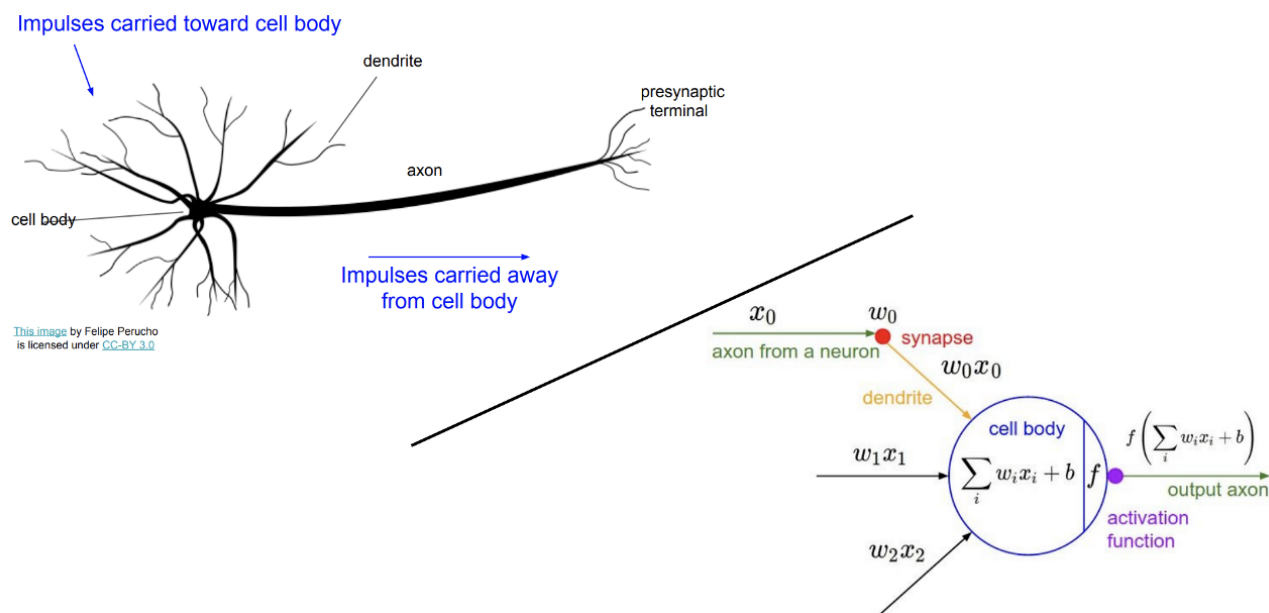
As you read these words you aren't examining every letter of every word, or every pixel making up each letter, to derive the meaning of the words. You're abstracting

away from the details and grouping things into higher-level concepts: words, phrases, sentences, paragraphs.

This ability to examine higher-level features is what allows you to understand what is happening in this sentence without too much trouble (or maybe you need too many drink texts).

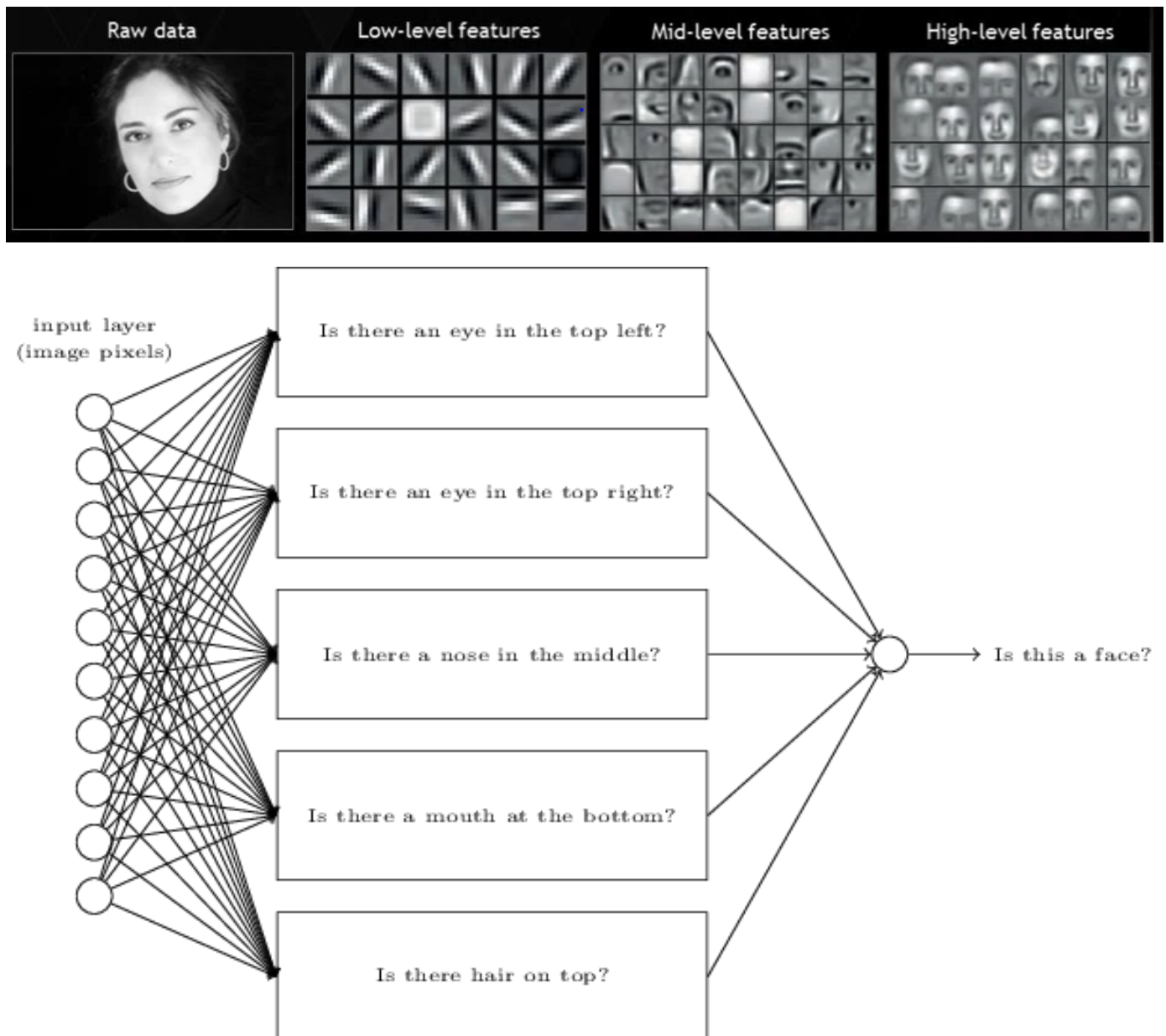
The same thing happens in vision, not just in humans but in animals' visual systems generally.

Brains are made up of neurons which "fire" by emitting electrical signals to other neurons after being sufficiently "activated". These neurons are malleable in terms of how much a signal from other neurons will add to the activation level of the neuron (vaguely speaking, the weights connecting neurons to each other end up being trained to make the neural connections more useful, just like the parameters in a linear regression can be trained to improve the mapping from input to output).



Side-by-side illustrations of biological and artificial neurons, via [Stanford's CS231n](#). This analogy can't be taken too literally—biological neurons can do things that artificial neurons can't, and vice versa—but it's useful to understand the biological inspiration. See Wikipedia's description of [biological vs. artificial neurons](#) for more detail.

Our biological networks are arranged in a hierarchical manner, so that certain neurons end up detecting not extremely specific features of the world around us, but rather more abstract features, i.e. patterns or groupings of more low-level features. For example, the [fusiform face area](#) in the human visual system is specialized for facial recognition.



Top: Illustration of learning increasingly abstract features, via [NVIDIA](#). Bottom: diagram of how an artificial neural network takes raw pixel inputs, develops intermediate “neurons” to detect higher-level features (e.g. presence of a nose), and combines the outputs of these to create a final output. Illustration from Neural Networks and Deep Learning ([Nielsen, 2017](#)).

This hierarchical structure exhibited by biological neural networks was discovered in the 1950s when researchers David Hubel and Torsten Wiesel were studying neurons in the visual cortex of cats. They were unable to observe neural activation after exposing the cat to a variety of stimuli: dark spots, light spots, hand-waving, and even pictures of women in magazines. But in their frustration, as they removed a slide from the projector at a diagonal angle, they noticed some neural activity! It turned out that diagonal edges at a very particular angle were causing certain neurons to be activated.



Background via [Knowing Neurons](#)

This makes sense evolutionarily since natural environments are generally noisy and random (imagine a grassy plain or a rocky terrain). So when a feline in the wild perceives an “edge”, i.e. a line that contrasts from its background, this might indicate that an object or creature is in the visual field. When a certain combination of edge neurons are activated, those activations will combine to yield a yet more abstract activation, and so on, until the final abstraction is a useful concept, like “bird” or “wolf”.

The idea behind a deep neural network is to mimic a similar structure with layers of artificial neurons.

Why linear models don't work

To draw from Stanford's excellent deep learning course, [CS231n: Convolutional Neural Networks and Visual Recognition](#), imagine that we want to train a neural network to classify images with the correct one of the following labels: `["plane", "car", "bird", "cat", "deer", "dog", "frog", "horse", "ship", "truck"]`.

One approach could be to construct a “template”, or average image, of each class of image using the training examples, and then use a nearest-neighbors algorithm during

testing to measure the distance of each unclassified image's pixel values, in aggregate, to each template. This approach involves no layers of abstraction. It's a linear model that combines all the different orientations of each type of image into one averaged blur.

For instance, it would take all the cars—regardless of whether they're facing left, right, center, and regardless of their color—and average them. The template then ends up looking rather vague and blurry.



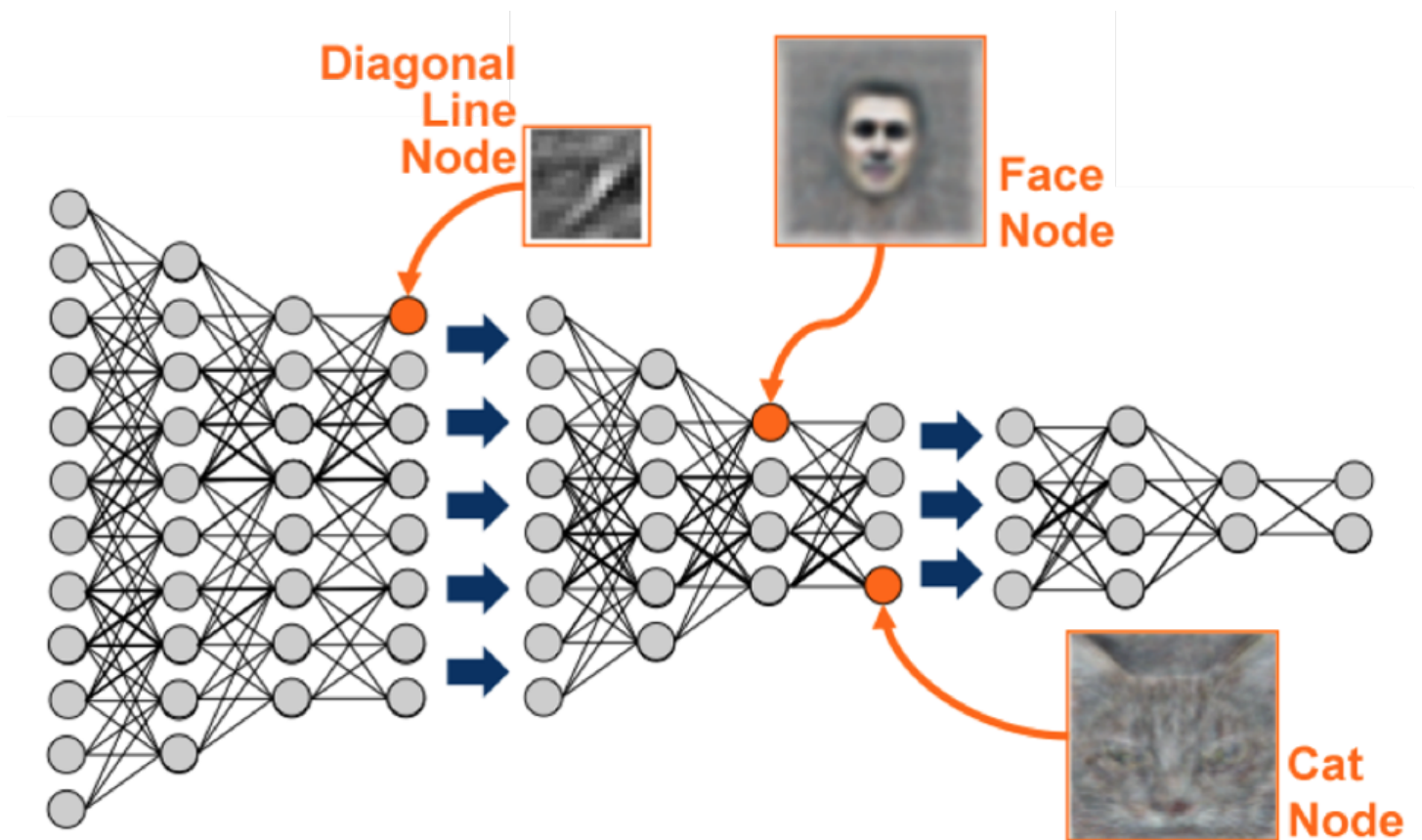
Example drawn from Stanford's [CS231n: Convolutional Neural Networks and Visual Recognition](#), Lecture 2.

Notice that the horse template above appears to have two heads. This doesn't really help us: we want to be able to detect right-facing horse or a left-facing horse separately, and then if either one of those features is detected, then we want to say we're looking at a horse. This flexibility is provided by deep neural nets, as we will see in the next section.

Deep neural networks approach the image classification problem using layers of abstraction

To repeat what we explained earlier in this section: the input layer will take raw pixel brightnesses of an image. The final layer will be an output vector of **class probabilities** (i.e. the probability of the image being a "cat", "car", "horse", etc.)

But instead of learning a simple linear model relating input to output, we'll instead construct intermediate **hidden layers** of the network will learn increasingly abstract features, which enables us to not lose all the nuance in the complex data.



Source: [Analytics Vidhya](#)

Just as we described animal brains detecting abstract features, the artificial neurons in the hidden layers will learn to detect abstract concepts—whichever concepts are ultimately most useful for capturing the most information and minimizing loss in the accuracy of the network’s output (this is an instance of unsupervised learning happening within the network).

This comes at the cost of model interpretability, since as you add in more hidden layers the neurons start representing more and more abstract and ultimately unintelligible features—to the point that you may hear deep learning referred to as “black box optimization”, where you basically are just trying stuff somewhat at random and seeing what comes out, without really understanding what’s happening inside.

Linear regression is interpretable because you decided which features to include in the model. Deep neural networks are harder to interpret because the features are learned and aren't explained anywhere in English. It's all in the machine's imagination.

Some extensions and further concepts worth noting

- **Deep learning software packages.** You'll rarely need to implement all the parts of neural networks from scratch because of existing libraries and tools that make deep learning implementations easier. There are many of these: TensorFlow, Caffe, Torch, Theano, and more.
- **Convolutional neural networks (CNNs).** CNNs are designed specifically for taking images as input, and are effective for computer vision tasks. They are also instrumental in deep reinforcement learning. CNNs are specifically inspired by the way animal visual cortices work, and they're the focus of the deep learning course we've been referencing throughout this article, Stanford's CS231n.
- **Recurrent neural networks (RNNs).** RNNs have a sense of built-in memory and are well-suited for language problems. They're also important in reinforcement learning since they enable the agent to keep track of where things are and what happened historically even when those elements aren't all visible at once. Christopher Olah wrote an [excellent walkthrough](#) of RNNs and LSTMs in the context of language problems.
- **Deep reinforcement learning.** This is one of the most exciting areas of deep learning research, at the heart of recent achievements like OpenAI defeating professional Dota 2 players and DeepMind's AlphaGo surpassing humans in the game of Go. We'll dive deeper in [Part 5](#), but essentially the goal is to apply all of the techniques in this post to the problem of teaching an agent to maximize reward. This can be applied in any context that can be gamified—from actual games like Counter Strike or Pacman, to self-driving cars, to trading stocks, to (ultimately) real life and the real world.

Deep learning applications

Deep learning is reshaping the world in virtually every domain. Here are a few examples of the incredible things that deep learning can do...

- *Facebook trained a neural network augmented by short-term memory to intelligently answer questions about the plot of Lord of the Rings.*

Facebook AI Research
@FBAIRResearch

- Home
- About
- Photos
- Reviews
- Videos
- Posts**
- Community

Create a Page

Here is an example of what the system can do. After having been trained, it was fed the following short story containing key events in JRR Tolkien's Lord of the Rings:

Bilbo travelled to the cave.
Gollum dropped the ring there.
Bilbo took the ring.
Bilbo went back to the Shire.
Bilbo left the ring there.
Frodo got the ring.
Frodo journeyed to Mount-Doom.
Frodo dropped the ring there.
Sauron died.
Frodo went back to the Shire.
Bilbo travelled to the Grey-havens.
The End.

After seeing this text, the system was asked a few questions, to which it provided the following answers:

Q: Where is the ring?
A: Mount-Doom
Q: Where is Bilbo now?
A: Grey-havens
Q: Where is Frodo now?
A: Shire

Research from **FAIR** (Facebook AI Research) applying deep neural networks augmented by separate short-term memory to intelligently answer questions about the LOTR storyline. This is the definition of epic.

- *Self-driving cars rely on deep learning for visual tasks like understanding road signs, detecting lanes, and recognizing obstacles.*



Source: [Business Insider](#)

- *Deep learning can be used for fun stuff like art generation. A tool called **neural style** can impressively mimic an artist's style and use it to remix another image.*



The style of Van Gogh's **Starry Night** applied to a picture of Stanford's campus, via Justin Johnson's neural style implementation: <https://github.com/jcjohnson/neural-style>

Other noteworthy examples include:

- *Predicting molecule bioactivity for drug discovery*
- *Face and object recognition for photo and video tagging*
- *Powering Google search results*
- *Natural language understanding and generation, e.g. Google Translate*
- *The Mars explorer robot Curiosity is autonomously selecting inspection-worthy soil targets based on visual examination*

...and many, many, more.

Now go do it!

We haven't gone into as much detail here on how neural networks are set up in practice because it's much easier to understand the details by implementing them yourself. Here are some amazing hands-on resources for getting started.

- Play around with the architecture of neural networks to see how different configurations affect network performance with the Google's [Neural Network Playground](#).
- Get up-and-running quickly with this tutorial by Google: [TensorFlow and deep learning, without a PhD](#). Classify handwritten digits at >99% accuracy, get familiar with TensorFlow, and learn deep learning concepts within 3 hours.
- Then, work through at least the first few lectures of [Stanford's CS231n](#) and the first assignment of building a two-layer neural network from scratch to really solidify the concepts covered in this article.

Further resources

Deep learning is an expansive subject area. Accordingly, we've also compiled some of the best resources we've encountered on the topic, in case you'd like to go... deeper.

- [Deeplearning.ai](#), Andrew Ng's new deep learning course with a comprehensive syllabus on the subject
- [CS231n: Convolutional Neural Networks for Visual Recognition](#), Stanford's deep learning course. One of the best treatments we've seen, with excellent lectures and illustrative problem sets
- Deep Learning & Neural Networks—accessible but rigorous
- [Deep Learning Book](#)—foundational, more mathematical
- [Fast.ai](#)—less theoretical, much more applied and black-boxy
- See Greg Brockman (CTO of OpenAI)'s answer to the question "What are the best ways to pick up Deep Learning skills as an engineer?" on [Quora](#)

Next up: time to play some games!

Last, but most certainly not least, is [Part 5: Reinforcement Learning](#).

Part 5: Reinforcement Learning

Exploration and exploitation. Markov decision processes. Q-learning, policy learning, and deep reinforcement learning.

"I just ate some chocolate for finishing the last section."

In supervised learning, training data comes with an answer key from some godlike "supervisor". If only life worked that way!

In **reinforcement learning (RL)** there's no answer key, but your reinforcement learning agent still has to decide how to act to perform its task. In the absence of existing training data, the **agent** learns from experience. It collects the training examples ("this action was good, that action was bad") through **trial-and-error** as it attempts its task, with the goal of maximizing long-term **reward**.

In this final section of Machine Learning for Humans, we will explore:

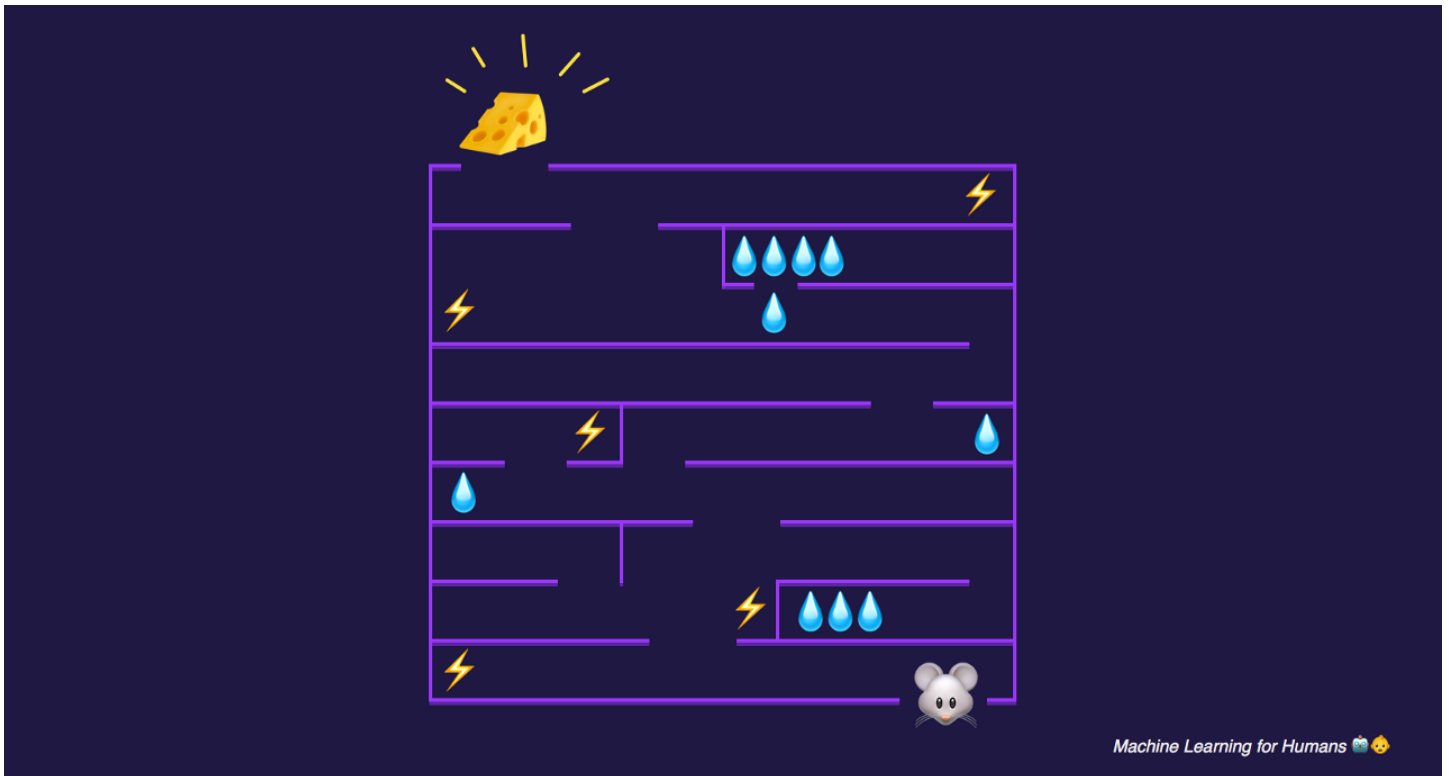
- *The **exploration/exploitation** tradeoff*
- ***Markov Decision Processes (MDPs)**, the classic setting for RL tasks*
- ***Q-learning, policy learning, and deep reinforcement learning***
- *and lastly, the **value learning problem***

At the end, as always, we've compiled some favorite resources for further exploration.

Let's put a robot mouse in a maze

The easiest context in which to think about reinforcement learning is in games with a clear objective and a point system.

Say we're playing a game where our mouse 🐭 is seeking the ultimate reward of cheese at the end of the maze (🧀 +1000 points), or the lesser reward of water along the way (💧 +10 points). Meanwhile, robo-mouse wants to avoid locations that deliver an electric shock (⚡ -100 points).



The reward is cheese.

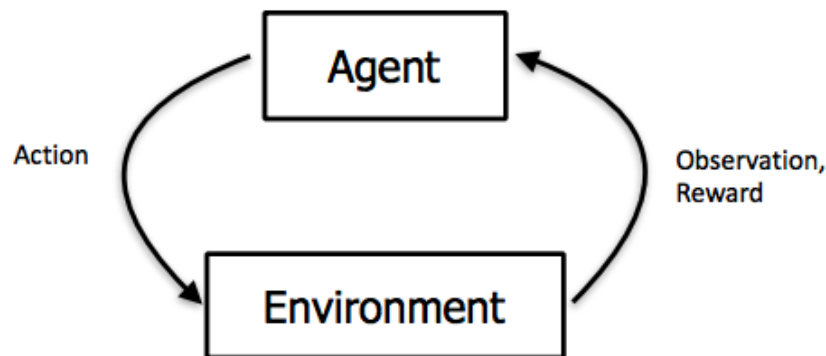
After a bit of **exploration**, the mouse might find the mini-paradise of three water sources clustered near the entrance and spend all its time **exploiting** that discovery by continually racking up the small rewards of these water sources and never going further into the maze to pursue the larger prize.

But as you can see, the mouse would then miss out on an even better oasis further in the maze, or the ultimate reward of cheese at the end!

This brings up the **exploration/exploitation** tradeoff. One simple strategy for exploration would be for the mouse to take the best known action most of the time (say, 80% of the time), but occasionally explore a new, randomly selected direction even though it might be walking away from known reward.

This strategy is called the **epsilon-greedy** strategy, where **epsilon** is the percent of the time that the agent takes a randomly selected action rather than taking the action that is most likely to maximize reward given what it knows so far (in this case, 20%). We usually start with a lot of exploration (i.e. a higher value for epsilon). Over time, as the mouse learns more about the maze and which actions yield the most long-term reward, it would make sense to steadily reduce epsilon to 10% or even lower as it settles into exploiting what it knows.

It's important to keep in mind that the reward is not always immediate: in the robot-mouse example, there might be a long stretch of the maze you have to walk through and several decision points before you reach the cheese.



The agent observes the environment, takes an action to interact with the environment, and receives positive or negative reward. Diagram from Berkeley's [CS 294: Deep Reinforcement Learning](#) by John Schulman & Pieter Abbeel

Markov Decision Processes (MDPs)

The mouse's wandering through the maze can be formalized as a Markov Decision Process, which is a process that has specified transition probabilities from state to state. We will explain it by referring to our robot-mouse example. MDPs include:

- 1. A finite set of states.** *These are the possible positions of our mouse within the maze.*
- 2. A set of actions available in each state.** *This is {forward, back} in a corridor and {forward, back, left, right} at a crossroads.*

3. Transitions between states. For example, if you go left at a crossroads you end up in a new position. These can be a set of probabilities that link to more than one possible state (e.g. when you use an attack in a game of Pokémon you can either miss, inflict some damage, or inflict enough damage to knock out your opponent).

4. Rewards associated with each transition. In the robot-mouse example, most of the rewards are 0, but they're positive if you reach a point that has water or cheese and negative if you reach a point that has an electric shock.

5. A discount factor γ between 0 and 1. This quantifies the difference in importance between immediate rewards and future rewards. For example, if γ is .9, and there's a reward of 5 after 3 steps, the present value of that reward is $.9^3 \cdot 5$.

6. Memorylessness. Once the current state is known, the history of the mouse's travels through the maze can be erased because the current Markov state contains all useful information from the history. In other words, "the future is independent of the past given the present".

Now that we know what an MDP is, we can formalize the mouse's objective. We're trying to maximize the sum of rewards in the long term:

$$\sum_{t=0}^{t=\infty} \gamma^t r(x(t), a(t))$$

Let's look at this sum term by term. First of all, we're summing across all time steps t . Let's set γ at 1 for now and forget about it. $r(x,a)$ is a reward function. For state x and action a (i.e., go left at a crossroads) it gives you the reward associated with taking that action a at state x . Going back to our equation, we're trying to maximize the sum of future rewards by taking the best action in each state.

Now that we've set up our reinforcement learning problem and formalized the goal, let's explore some possible solutions.

Q-learning: learning the action-value function

Q-learning is a technique that evaluates which action to take based on an **action-value function** that determines the value of being in a certain state and taking a certain action at that state.

We have a function Q that takes as an input one state and one action and returns the **expected reward** of that action (and all subsequent actions) at that state. Before we explore the environment, Q gives the same (arbitrary) fixed value. But then, as we explore the environment more, Q gives us a better and better approximation of the value of an action a at a state s . We update our function Q as we go.

This equation from the [Wikipedia page on Q-learning](#) explains it all very nicely. It shows how we update the value of Q based on the reward we get from our environment:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)$$

Let's ignore the discount factor γ by setting it to 1 again. First, keep in mind that Q is supposed to show you the full sum of rewards from choosing action Q and all the optimal actions afterward.

Now let's go through the equation from left to right. When we take action a_t in state s_t , we update our value of $Q(s_t, a_t)$ by adding a term to it. This term contains:

- *Learning rate **alpha**: this is how aggressive we want to be when updating our value. When alpha is close to 0, we're not updating very aggressively. When alpha is close to 1, we're simply replacing the old value with the updated value.*
- *The **reward** is the reward we got by taking action a_t at state s_t . So we're adding*

this reward to our old estimate.

- *We're also adding the **estimated future reward**, which is the maximum achievable reward Q for all available actions at x_{t+1} .*
- *Finally, we subtract the old value of Q to make sure that we're only incrementing or decrementing by the difference in the estimate (multiplied by alpha of course).*

Now that we have a value estimate for each state-action pair, we can select which action to take according to our **action-selection strategy** (we don't necessarily just choose the action that leads to the most expected reward every time, e.g. with an epsilon-greedy exploration strategy we'd take a random action some percentage of the time).

In the robot mouse example, we can use Q-learning to figure out the value of each position in the maze and the value of the actions {forward, backward, left, right} at each position. Then we can use our action-selection strategy to choose what the mouse actually does at each time step.

Policy learning: a map from state to action

In the Q-learning approach, we learned a value function that estimated the value of each state-action pair.

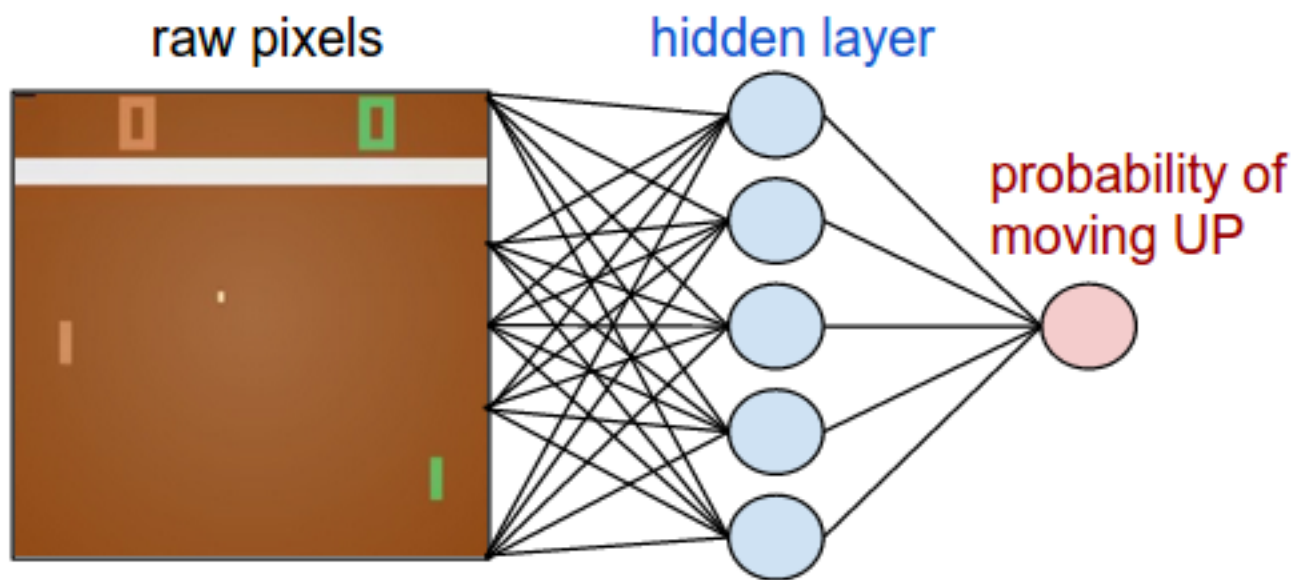
Policy learning is a more straightforward alternative in which we learn a **policy function**, π , which is a direct map from each state to the best corresponding action at that state. Think of it as a behavioral policy: "when I observe state s , the best thing to do is take action a ". For example, an autonomous vehicle's policy might effectively include something like: "if I see a yellow light and I am more than 100 feet from the intersection, I should brake. Otherwise, keep moving forward."

$$a = \pi(s)$$

A policy is a map from state to action.

So we're learning a function that will maximize expected reward. What do we know that's really good at learning complex functions? Deep neural networks!

Andrej Karpathy's [Pong from Pixels](#) provides an excellent walkthrough on using **deep reinforcement learning** to learn a policy for the Atari game Pong that takes raw pixels from the game as the input (state) and outputs a probability of moving the paddle up or down (action).



In a policy gradient network, the agent learns the optimal policy by adjusting its weights through gradient descent based on reward signals from the environment. Image via <http://karpathy.github.io/2016/05/31/rl/>

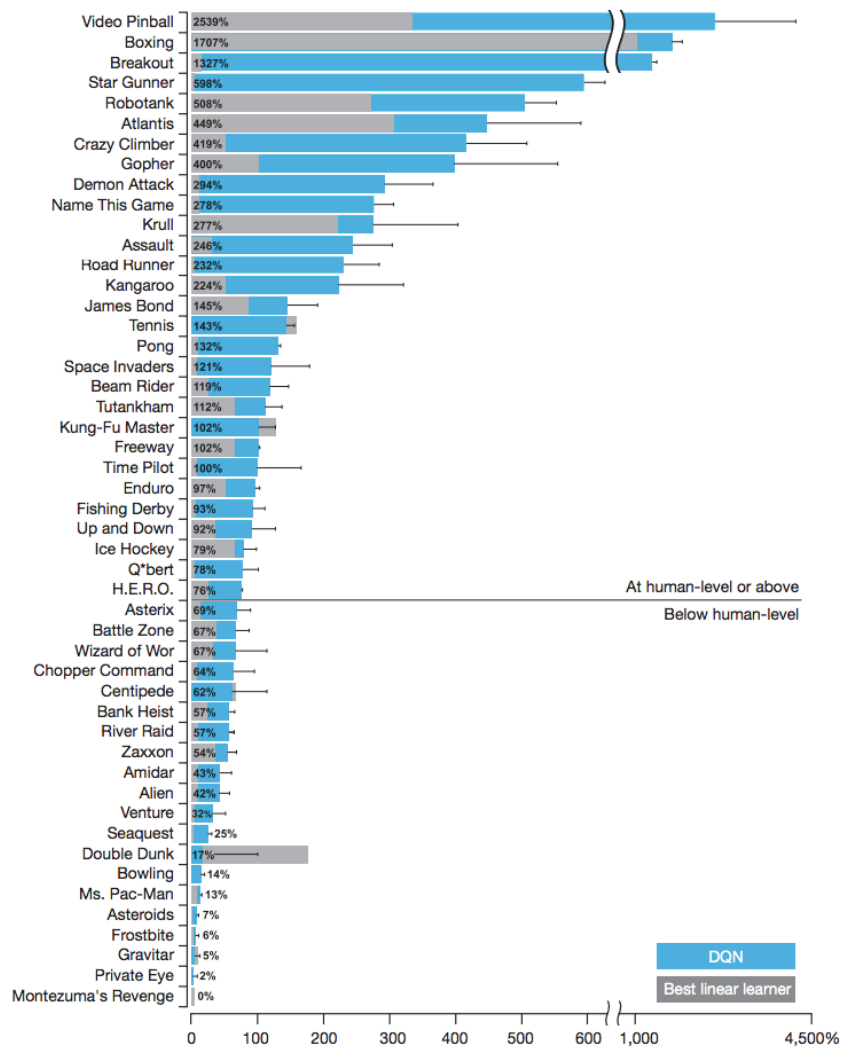
If you want to get your hands dirty with deep RL, work through Andrej's post. You will implement a 2-layer policy network in 130 lines of code, and will also learn how to plug into OpenAI's [Gym](#), which allows you to quickly get up and running with your first reinforcement learning algorithm, test it on a variety of games, and see how its performance compares to other submissions.

DQNs, A3C, and advancements in deep RL

In 2015, DeepMind used a method called deep Q-networks (DQN), an approach that approximates Q-functions using deep neural networks, to beat human benchmarks across many Atari games:

We demonstrate that the deep Q -network agent, receiving only the pixels and the game score as inputs, was able to surpass the performance of all previous algorithms and achieve a level comparable to that of a professional human games tester across a set of 49 games, using the same algorithm, network architecture and hyperparameters. This work bridges the divide between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to excel at a diverse array of challenging tasks. (Silver et al., 2015)

Here is a snapshot of where DQN agents stand relative to linear learners and humans in various domains:



These are normalized with respect to professional human games testers: 0% = random play, 100% = human performance.

Source: DeepMind's DQN paper, [Human-level control through deep reinforcement learning](#)

To help you build some intuition for how advancements are made in RL research, here are some examples of improvements on attempts at non-linear Q-function approximators that can improve performance and stability:

- **Experience replay**, which learns by randomizing over a longer sequence of previous observations and corresponding reward to avoid overfitting to recent experiences. This idea is inspired by biological brains: rats traversing mazes, for example, “replay” patterns of neural activity during sleep in order to optimize future behavior in the maze.
- **Recurrent neural networks (RNNs)** augmenting DQNs. When an agent can only see its immediate surroundings (e.g. robot-mouse only seeing a certain segment of the maze vs. a birds-eye view of the whole maze), the agent needs to remember the bigger picture so it remembers where things are. This is similar to how humans babies develop object permanence to know things exist even if they leave the baby’s visual field. RNNs are “recurrent”, i.e. they allow information to persist on a longer-term basis. Here’s an impressive video of a deep recurrent Q-network (DQRN) playing Doom.



Paper: <https://arxiv.org/abs/1609.05521>. Source: Arthur Juliani’s [Simple Reinforcement Learning with Tensorflow](#) series

In 2016, just one year after the DQN paper, DeepMind revealed another algorithm called **Asynchronous Advantage Actor-Critic (A3C)** that surpassed state-of-the-art performance on Atari games after training for half as long (Mnih et al., 2016). A3C is an **actor-critic** algorithm that combines the best of both approaches we explored earlier: it uses an **actor** (a policy network that decides how to act) AND a **critic** (a Q-network that decides how valuable things are). Arthur Juliani has a nice [writeup](#) on how A3C works specifically. A3C is now OpenAI’s [Universe Starter Agent](#).

Since then, there have been countless fascinating breakthroughs—from AIs [inventing their own language](#) to [teaching themselves to walk in a variety of terrains](#). This series only scratches the surface on the cutting edge of RL, but hopefully it will serve as a starting point for further exploration!

As a parting note, we'd like to share this incredible video of DeepMind's agents that learned to walk... with added sound. Grab some popcorn, turn up the volume, and witness the full glory of artificial intelligence.



Practice materials & further reading

Code

- *Andrej Karpathy's [Pong from Pixels](#) will get you up-and-running quickly with your first reinforcement learning agent. As the article describes, "we'll learn to play an Atari game (Pong!) with PG, from scratch, from pixels, with a deep neural network, and the whole thing is 130 lines of Python only using numpy as a dependency ([Gist link](#))."*
- *Next, we'd highly recommend Arthur Juliani's [Simple Reinforcement Learning with Tensorflow](#) tutorial. It walks through DQNs, policy learning, actor-critic methods, and strategies for exploration with implementations using TensorFlow. Try understanding and then re-implementing the methods covered.*

Reading + lectures

- *Richard Sutton's book, [Reinforcement Learning: An Introduction](#)—a fantastic book, very readable*
- *John Schulman's [CS 294: Deep Reinforcement Learning](#) (Berkeley)*
- *David Silver's [Reinforcement Learning](#) course (UCL)*

YOU DID IT!

If you've made it this far, that is all the reward we could hope for. We hope you enjoyed the series as an introduction to machine learning. We've compiled some of our favorite ML resources in the [Appendix](#) if you're ready to see how deep this rabbit hole goes.

Please don't hesitate to reach out with thoughts, questions, feedback, or your favorite GIFs!

Until next time,

Vishal and Samer

Closing thoughts:

There is a fundamental question that inspired this series, and we'd like to pose it to you as well.

What is our objective function, as humans? How do we define the reward that we maximize in our real lives? Beyond base pleasure and pain, our definition of reward also tends to include messy things like right and wrong, fulfillment, love, spirituality, and purpose.

There has been an intellectual field dedicated to the question of what our objective functions are or should be since ancient times, and it's called **moral philosophy**. The central question of moral philosophy is: what ought we do? How should we live? Which actions are right or wrong? The answer is, quite clearly: it depends on your values.

As we create more and more advanced AI, it will start to depart from the realm of toy problems like Atari games, where "reward" is cleanly defined by how many points are won in the game, and exist more and more in the real world. Autonomous vehicles, for example, have to make decisions with a somewhat more complex definition of reward. At first, reward might be tied to something like "getting safely to the destination". But if forced to choose between staying the course and hitting five pedestrians or swerving and hitting one, should the vehicle swerve? What if the one pedestrian is a child, or a gunman on the loose, or the next Einstein? How does that change the decision, and why? What if swerving also destroys a piece of valuable art? Suddenly we have a much more complex problem when we try to define the objective function, and the answers are not as simple.

In this series, we explored why it's difficult to specify explicitly to a computer what a cat looks like - if asked how we know ourselves, the answer is, most simply, "intuition" - but we've explored **machine vision** approaches to teaching the machine to learn this intuition by itself. Similarly, in the domain of **machine morality**, it might be difficult to specify exactly how to evaluate the rightness or wrongness of one action vs. another, but perhaps it is possible for a machine to learn these values in some way. This is called the **values learning problem**, and it may be one of the most important technical problems humans will ever have to solve.

For more on this topic, see this synoptic post on the [Risks of Artificial Intelligence](#). And as you go forth into the world of making machine smarter and smarter, we'd encourage you to keep in mind that AI progress is a double-edged sword, of particular keenness on both sides.

Appendix: The Best Machine Learning Resources

A compendium of resources for crafting a curriculum on artificial intelligence, machine learning, and deep learning.

This article is an addendum to the series [Machine Learning for Humans](#) 🤖 😊, a guide for getting up-to-speed on machine learning concepts in 2-3 hours.

General advice on crafting a curriculum

Going to school for a formal degree program for isn't always possible or desirable. For those considering an autodidactic alternative, this is for you.

1. Build foundations, and then specialize in areas of interest.

You can't go deeply into every machine learning topic. There's too much to learn, and the field is advancing rapidly. Master foundational concepts and then focus on projects in a specific domain of interest—whether it's natural language understanding, computer vision, deep reinforcement learning, robotics, or whatever else.

2. Design your curriculum around topics that personally excite you.

Motivation is far more important than micro-optimizing a learning strategy for some long-term academic or career goal. If you're having fun, you'll make fast progress. If you're trying to force yourself forward, you'll slow down.

Foundations

Programming

Syntax and basic concepts: [Google's Python Class](#), [Learn Python the Hard Way](#).

Practice: [Coderbyte](#), [Codewars](#), [HackerRank](#).

Linear algebra

[Deep Learning Book, Chapter 2: Linear Algebra](#). A quick review of the linear algebra concepts relevant to machine learning.

[A First Course in Linear Model Theory](#) by Nalini Ravishanker and Dipak Dey. Textbook introducing linear algebra in a statistical context.

Probability & statistics

MIT 18.05, [Introduction to Probability and Statistics](#), taught by Jeremy Orloff and Jonathan Bloom. Provides intuition for probabilistic reasoning & statistical inference, which is invaluable for understanding how machines think, plan, and make decisions.

[All of Statistics: A Concise Course in Statistical Inference](#), by Larry Wasserman.

Introductory text on statistics.

Calculus

Khan Academy: [Differential Calculus](#). Or, any introductory calculus course or textbook.

Stanford CS231n: [Derivatives, Backpropagation, and Vectorization](#), prepared by Justin Johnson.

Machine learning

Courses

Andrew Ng's [Machine Learning](#) course on Coursera (or, for more rigor, [Stanford CS229](#)).

Machine learning bootcamps: [Galvanize](#) (full-time, 3 months, \$\$\$\$), [Thinkful](#) (flexible schedule, 6 months, \$\$).

Textbook

[An Introduction to Statistical Learning](#) by Gareth James et al. Excellent reference for essential machine learning concepts, available free online.

Deep learning

Courses

[Deeplearning.ai](#), Andrew Ng's introductory deep learning course.

[CS231n: Convolutional Neural Networks for Visual Recognition](#), Stanford's deep learning course. Helpful for building foundations, with engaging lectures and illustrative problem sets.

Projects

[Fast.ai](#), a fun and hands-on project-based course. Projects include classifying images of dogs vs. cats and generating Nietzschean writing.

[MNIST handwritten digit classification with TensorFlow](#). Classify handwritten digits with >99% accuracy in 3 hours with this tutorial by Google.

Try your hand at [a Kaggle competition](#). Implement a deep learning paper that you found interesting, using other versions on GitHub as reference material.

Reading

[Deep Learning Book](#), a.k.a. the Bible of Deep Learning, authored by Ian Goodfellow, Yoshua Bengio, and Aaron Courville.

[Neural Networks and Deep Learning](#), a clear and accessible online deep learning text by Michael Nielsen. Ends with commentary on reaching human-level intelligence.

[Deep Learning Papers Reading Roadmap](#), a compilation of key papers organized by chronology and research area.

Reinforcement learning

Courses

John Schulman's [CS 294: Deep Reinforcement Learning](#) at Berkeley.

David Silver's [Reinforcement Learning](#) course at University College London.

[Deep RL Bootcamp](#), organized by OpenAI and UC Berkeley. Applications are currently closed, but it's worth keeping an eye out for future sessions.

Projects

Andrej Karpathy's [Pong from Pixels](#). Implement a Pong-playing agent from scratch in 130 lines of code.

Arthur Juliani's [Simple Reinforcement Learning with Tensorflow](#) series. Implement Q-learning, policy-learning, actor-critic methods, and strategies for exploration using TensorFlow.

See OpenAI's [requests for research](#) for more project ideas.

Reading

Richard Sutton's book, [Reinforcement Learning: An Introduction](#).

Artificial intelligence

[Artificial Intelligence: A Modern Approach](#) by Stuart Russell and Peter Norvig.

Sebastian Thrun's Udacity course, [Intro to Artificial Intelligence](#).

Fellowships: [Insight AI Fellows Program](#), [Google Brain Residency Program](#)

Artificial intelligence safety

For the short version, read: (1) Johannes Heidecke's [Risks of Artificial Intelligence](#), (2) OpenAI and Google Brain's collaboration on [Concrete Problems in AI Safety](#), and (3) Wait But Why's article on the [AI Revolution](#).

For the longer version, see Nick Bostrom's [Superintelligence](#).

Check out the research published by the [Machine Intelligence Research Institute](#) (MIRI) and [Future of Humanity Institute](#) (FHI) on AI safety.

Keep up-to-date with [/r/ControlProblem](#) on Reddit.

Newsletters

[Import AI](#), weekly AI newsletter covering the latest developments in the industry.

Prepared by Jack Clark of OpenAI.

[Machine Learnings](#), prepared by Sam DeBrule. Frequent guest appearances from experts in the field.

[Nathan.ai](#), covering recent news and commenting on AI/ML from a venture capital perspective.

Advice from others

"What is the best way to learn machine learning without taking any online courses? — answered by Eric Jang, Google Brain

What are the best ways to pick up deep learning skills as an engineer?" - answered by Greg Brockman, CTO of OpenAI

A16z's [AI Playbook](#), a more code-based introduction to AI

[AI safety syllabus](#), designed by 80,000 Hours



"You take the blue pill, the story ends. You wake up in your bed and believe whatever you want to believe. You take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole goes." —Morpheus



On Twitter? So are we. Feel free to keep in touch—[Vishal](#) and [Samer](#)
Contact: ml4humans@gmail.com