Machine Learning Yearning is a
deeplearning.ai project.

Table of Contents (Draft)

1 Why Machine Learning Strategy

Machine learning is the foundation of countless important applications, including web search, email anti-spam, speech recognition, product recommendations, and more. I assume that you or your team is working on a machine learning application, and that you want to make rapid progress. This book will help you do so.

Example: Building a cat picture startup

Say you're building a startup that will provide an endless stream of cat pictures to cat lovers.

You use a neural network to build a computer vision system for detecting cats in pictures.

But tragically, your learning algorithm's accuracy is not yet good enough. You are under tremendous pressure to improve your cat detector. What do you do?

Your team has a lot of ideas, such as:

- Get more data: Collect more pictures of cats.
- Collect a more diverse training set. For example, pictures of cats in unusual positions; cats with unusual coloration; pictures shot with a variety of camera settings; ….
- Train the algorithm longer, by running more gradient descent iterations.
- Try a bigger neural network, with more layers/hidden units/parameters.

- Try a smaller neural network.
- Try adding regularization (such as L2 regularization).
- Change the neural network architecture (activation function, number of hidden units, etc.)
- …

If you choose well among these possible directions, you'll build the leading cat picture platform, and lead your company to success. If you choose poorly, you might waste months. How do you proceed?

This book will tell you how. Most machine learning problems leave clues that tell you what's useful to try, and what's not useful to try. Learning to read those clues will save you months or years of development time.

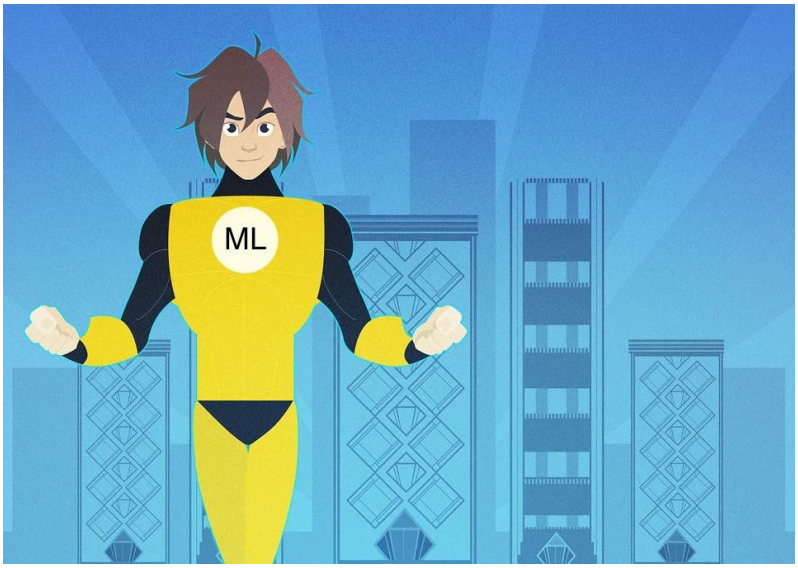
2 How to use this book to help your team

After finishing this book, you will have a deep understanding of how to set technical direction for a machine learning project.

But your teammates might not understand why you're recommending a particular direction. Perhaps you want your team to define a single-number evaluation metric, but they aren't convinced. How do you persuade them?

That's why I made the chapters short: So that you can print them out and get your teammates to read just the 1-2 pages you need them to know.

A few changes in prioritization can have a huge effect on your team's productivity. By helping your team with a few such changes, I hope that you can become the superhero of your team!



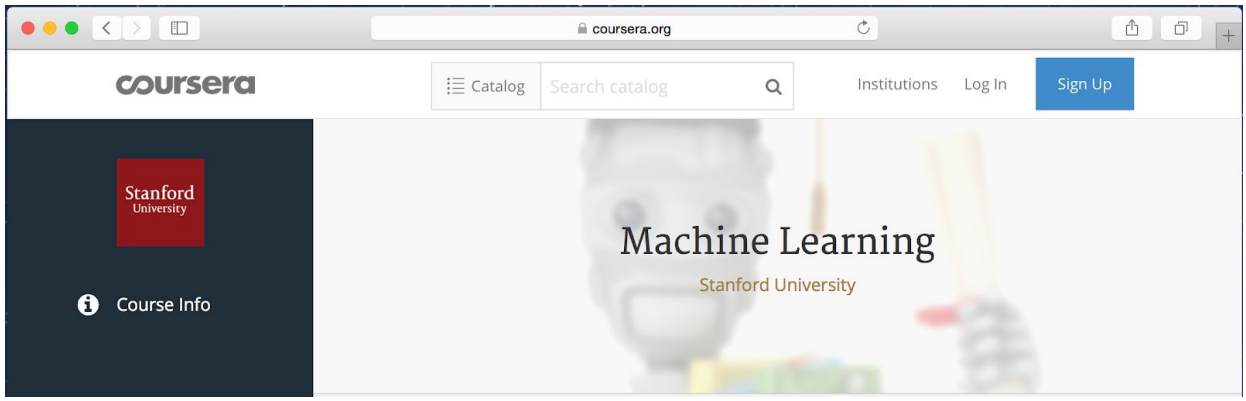
3 Prerequisites and Notation

If you have taken a Machine Learning course such as my machine learning MOOC on Coursera, or if you have experience applying supervised learning, you will be able to understand this text.

I assume you are familiar with **supervised learning**: learning a function that maps from x to y, using labeled training examples (x,y). Supervised learning algorithms include linear regression, logistic regression, and neural networks. There are many forms of machine learning, but the majority of Machine Learning's practical value today comes from supervised learning.

I will frequently refer to neural networks (also known as "deep learning"). You'll only need a basic understanding of what they are to follow this text.

If you are not familiar with the concepts mentioned here, watch the first three weeks of videos in the Machine Learning course on Coursera at http://ml-class.org



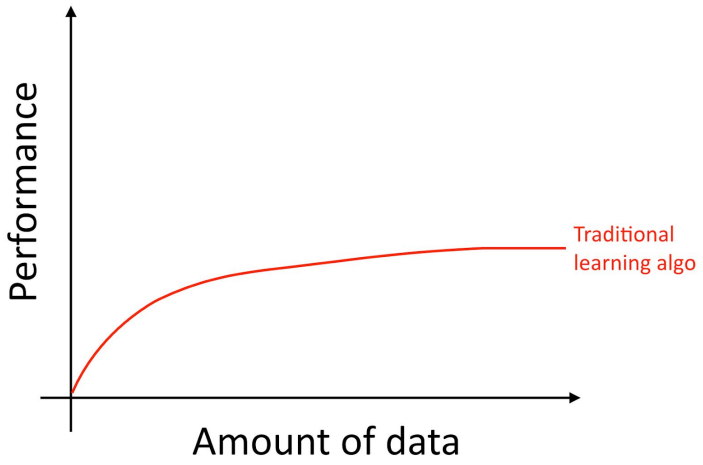
4 Scale drives machine learning progress

Many of the ideas of deep learning (neural networks) have been around for decades. Why are these ideas taking off now?

Two of the biggest drivers of recent progress have been:

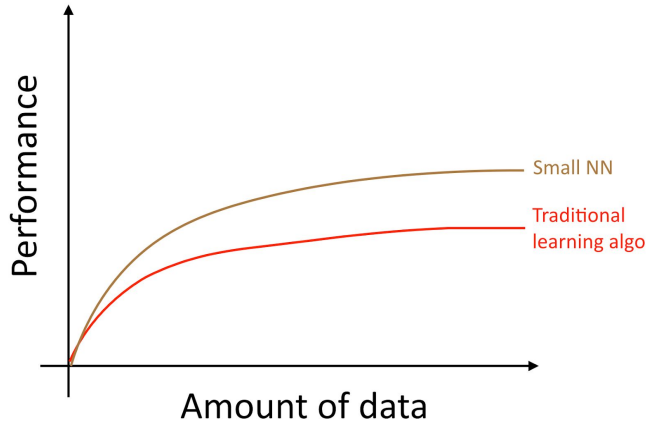
- **Data availability.** People are now spending more time on digital devices (laptops, mobile devices). Their digital activities generate huge amounts of data that we can feed to our learning algorithms.
- **Computational scale.** We started just a few years ago to be able to train neural networks that are big enough to take advantage of the huge datasets we now have.

In detail, even as you accumulate more data, usually the performance of older learning algorithms, such as logistic regression, "plateaus." This means its learning curve "flattens out," and the algorithm stops improving even as you give it more data:

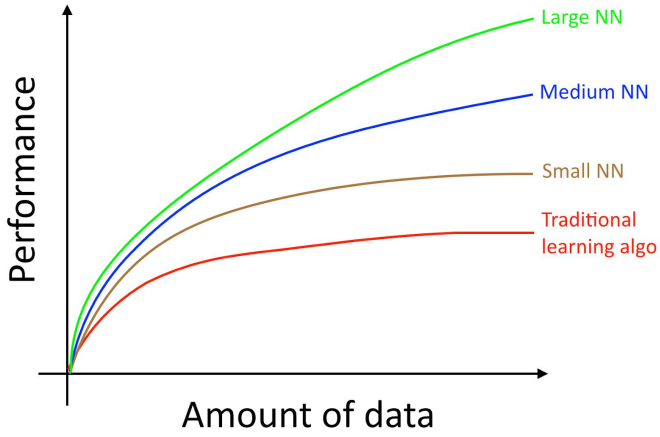


It was as if the older algorithms didn't know what to do with all the data we now have.

If you train a small neural network (NN) on the same supervised learning task, you might get slightly better performance:



Here, by "Small NN" we mean a neural network with only a small number of hidden units/layers/parameters. Finally, if you train larger and larger neural networks, you can obtain even better performance:[1]



Thus, you obtain the best performance when you (i) Train a very large neural network, so that you are on the green curve above; (ii) Have a huge amount of data.

Many other details such as neural network architecture are also important, and there has been much innovation here. But one of the more reliable ways to improve an algorithm's performance today is still to (i) train a bigger network and (ii) get more data.

[1] This diagram shows NNs doing better in the regime of small datasets. This effect is less consistent than the effect of NNs doing well in the regime of huge datasets. In the small data regime, depending on how the features are hand-engineered, traditional algorithms may or may not do better. For example, if you have 20 training examples, it might not matter much whether you use logistic regression or a neural network; the hand-engineering of features will have a bigger effect than the choice of algorithm. But if you have 1 million examples, I would favor the neural network.

The process of how to accomplish (i) and (ii) are surprisingly complex. This book will discuss the details at length. We will start with general strategies that are useful for both traditional learning algorithms and neural networks, and build up to the most modern strategies for building deep learning systems.

Setting up development and test sets

5 Your development and test sets

Let's return to our earlier cat pictures example: You run a mobile app, and users are uploading pictures of many different things to your app. You want to automatically find the cat pictures.

Your team gets a large training set by downloading pictures of cats (positive examples) and non-cats (negative examples) off of different websites. They split the dataset 70%/30% into training and test sets. Using this data, they build a cat detector that works well on the training and test sets.

But when you deploy this classifier into the mobile app, you find that the performance is really poor!

What happened?

You figure out that the pictures users are uploading have a different look than the website images that make up your training set: Users are uploading pictures taken with mobile phones, which tend to be lower resolution, blurrier, and poorly lit. Since your training/test sets were made of website images, your algorithm did not generalize well to the actual distribution you care about: mobile phone pictures.

Before the modern era of big data, it was a common rule in machine learning to use a random 70%/30% split to form your training and test sets. This practice can work, but it's a bad idea in more and more applications where the training distribution (website images in

our example above) is different from the distribution you ultimately care about (mobile phone images).

We usually define:

- **Training set** — Which you run your learning algorithm on.
- **Dev (development) set** — Which you use to tune parameters, select features, and make other decisions regarding the learning algorithm. Sometimes also called the **hold-out cross validation set**.
- **Test set** — which you use to evaluate the performance of the algorithm, but not to make any decisions regarding what learning algorithm or parameters to use.

Once you define a dev set (development set) and test set, your team will try a lot of ideas, such as different learning algorithm parameters, to see what works best. The dev and test sets allow your team to quickly see how well your algorithm is doing.

In other words, **the purpose of the dev and test sets are to direct your team toward the most important changes to make to the machine learning system**.

So, you should do the following:

> Choose dev and test sets to reflect data you expect to get in the future and want to do well on.

In other words, your test set should not simply be 30% of the available data, especially if you expect your future data (mobile phone images) to be different in nature from your training set (website images).

If you have not yet launched your mobile app, you might not have any users yet, and thus might not be able to get data that accurately reflects what you have to do well on in the future. But you might still try to approximate this. For example, ask your friends to take mobile phone pictures of cats and send them to you. Once your app is launched, you can update your dev/test sets using actual user data.

If you really don't have any way of getting data that approximates what you expect to get in the future, perhaps you can start by using website images. But you should be aware of the risk of this leading to a system that doesn't generalize well.

It requires judgment to decide how much to invest in developing great dev and test sets. But don't assume your training distribution is the same as your test distribution. Try to pick test

examples that reflect what you ultimately want to perform well on, rather than whatever data you happen to have for training.

6 Your dev and test sets should come from the same distribution

You have your cat app image data segmented into four regions, based on your largest markets: (i) US, (ii) China, (iii) India, and (iv) Other. To come up with a dev set and a test set, say we put US and India in the dev set; China and Other in the test set. In other words, we can randomly assign two of these segments to the dev set, and the other two to the test set, right?

Once you define the dev and test sets, your team will be focused on improving dev set performance. Thus, the dev set should reflect the task you want to improve on the most: To do well on all four geographies, and not only two.

There is a second problem with having different dev and test set distributions: There is a chance that your team will build something that works well on the dev set, only to find that it does poorly on the test set. I've seen this result in much frustration and wasted effort. Avoid letting this happen to you.

As an example, suppose your team develops a system that works well on the dev set but not the test set. If your dev and test sets had come from the same distribution, then you would have a very clear diagnosis of what went wrong: You have overfit the dev set. The obvious cure is to get more dev set data.

But if the dev and test sets come from different distributions, then your options are less clear. Several things could have gone wrong:

1. You had overfit to the dev set.
2. The test set is harder than the dev set. So your algorithm might be doing as well as could be expected, and there's no further significant improvement is possible.

3. The test set is not necessarily harder, but just different, from the dev set. So what works well on the dev set just does not work well on the test set. In this case, a lot of your work to improve dev set performance might be wasted effort.

Working on machine learning applications is hard enough. Having mismatched dev and test sets introduces additional uncertainty about whether improving on the dev set distribution also improves test set performance. Having mismatched dev and test sets makes it harder to figure out what is and isn't working, and thus makes it harder to prioritize what to work on.

If you are working on a 3rd party benchmark problem, their creator might have specified dev and test sets that come from different distributions. Luck, rather than skill, will have a greater impact on your performance on such benchmarks compared to if the dev and test sets come from the same distribution. It is an important research problem to develop learning algorithms that are trained on one distribution and generalize well to another. But if your goal is to make progress on a specific machine learning application rather than make research progress, I recommend trying to choose dev and test sets that are drawn from the same distribution. This will make your team more efficient.

7 How large do the dev/test sets need to be?

The dev set should be large enough to detect differences between algorithms that you are trying out. For example, if classifier A has an accuracy of 90.0% and classifier B has an accuracy of 90.1%, then a dev set of 100 examples would not be able to detect this 0.1% difference. Compared to other machine learning problems I've seen, a 100 example dev set is small. Dev sets with sizes from 1,000 to 10,000 examples are common. With 10,000 examples, you will have a good chance of detecting an improvement of 0.1%.[2]

For mature and important applications—for example, advertising, web search, and product recommendations—I have also seen teams that are highly motivated to eke out even a 0.01% improvement, since it has a direct impact on the company's profits. In this case, the dev set could be much larger than 10,000, in order to detect even smaller improvements.

How about the size of the test set? It should be large enough to give high confidence in the overall performance of your system. One popular heuristic had been to use 30% of your data for your test set. This works well when you have a modest number of examples—say 100 to 10,000 examples. But in the era of big data where we now have machine learning problems with sometimes more than a billion examples, the fraction of data allocated to dev/test sets has been shrinking, even as the absolute number of examples in the dev/test sets has been growing. There is no need to have excessively large dev/test beyond what is needed to evaluate the performance of your algorithms.

[2] In theory, one could also test if a change to an algorithm makes a statistically significant difference on the dev set. In practice, most teams don't bother with this (unless they are publishing academic research papers), and I usually do not find statistical significance tests useful for measuring interim progress.

8 Establish a single-number evaluation metric for your team to optimize

Classification accuracy is an example of a **single-number evaluation metric**: You run your classifier on the dev set (or test set), and get back a single number about what fraction of examples it classified correctly. According to this metric, if classifier A obtains 97% accuracy, and classifier B obtains 90% accuracy, then we judge classifier A to be superior.

In contrast, Precision and Recall[3] is not a single-number evaluation metric: It gives two numbers for assessing your classifier. Having multiple-number evaluation metrics makes it harder to compare algorithms. Suppose your algorithms perform as follows:

Classifier	Precision	Recall
A	95%	90%
B	98%	85%

Here, neither classifier is obviously superior, so it doesn't immediately guide you toward picking one.

During development, your team will try a lot of ideas about algorithm architecture, model parameters, choice of features, etc. Having a **single-number evaluation metric** such as accuracy allows you to sort all your models according to their performance on this metric, and quickly decide what is working best.

If you really care about both Precision and Recall, I recommend using one of the standard ways to combine them into a single number. For example, one could take the average of precision and recall, to end up with a single number. Alternatively, you can compute the "F1 score," which is a modified way of computing their average, and works better than simply taking the mean.[4]

[3] The Precision of a cat classifier is the fraction of images in the dev (or test) set it labeled as cats that really are cats. Its Recall is the percentage of all cat images in the dev (or test) set that it correctly labeled as a cat. There is often a tradeoff between having high precision and high recall.

[4] If you want to learn more about the F1 score, see https://en.wikipedia.org/wiki/F1_score. It is the "harmonic mean" between Precision and Recall, and is calculated as 2/((1/Precision)+(1/Recall)).

Classifier	Precision	Recall	F1 score
A	95%	90%	**92.4%**
B	98%	85%	**91.0%**

Having a single-number evaluation metric speeds up your ability to make a decision when you are selecting among a large number of classifiers. It gives a clear preference ranking among all of them, and therefore a clear direction for progress.

As a final example, suppose you are separately tracking the accuracy of your cat classifier in four key markets: (i) US, (ii) China, (iii) India, and (iv) Other. This gives four metrics. By taking an average or weighted average of these four numbers, you end up with a single number metric. Taking an average or weighted average is one of the most common ways to combine multiple metrics into one.

9 Optimizing and satisficing metrics

Here's another way to combine multiple evaluation metrics.

Suppose you care about both the accuracy and the running time of a learning algorithm. You need to choose from these three classifiers:

Classifier	Accuracy	Running time
A	90%	80ms
B	92%	95ms
C	95%	1,500ms

It seems unnatural to derive a single metric by putting accuracy and running time into a single formula, such as:

Accuracy - 0.5*RunningTime

Here's what you can do instead: First, define what is an "acceptable" running time. Let's say anything that runs in 100ms is acceptable. Then, maximize accuracy, subject to your classifier meeting the running time criteria. Here, running time is a "satisficing metric"—your classifier just has to be "good enough" on this metric, in the sense that it should take at most 100ms. Accuracy is the "optimizing metric."

If you are trading off N different criteria, such as binary file size of the model (which is important for mobile apps, since users don't want to download large apps), running time, and accuracy, you might consider setting N-1 of the criteria as "satisficing" metrics. I.e., you simply require that they meet a certain value. Then define the final one as the "optimizing" metric. For example, set a threshold for what is acceptable for binary file size and running time, and try to optimize accuracy given those constraints.

As a final example, suppose you are building a hardware device that uses a microphone to listen for the user saying a particular "wakeword," that then causes the system to wake up. Examples include Amazon Echo listening for "Alexa"; Apple Siri listening for "Hey Siri"; Android listening for "Okay Google"; and Baidu apps listening for "Hello Baidu." You care about both the false positive rate—the frequency with which the system wakes up even when no one said the wakeword—as well as the false negative rate—how often it fails to wake up when someone says the wakeword. One reasonable goal for the performance of this system is

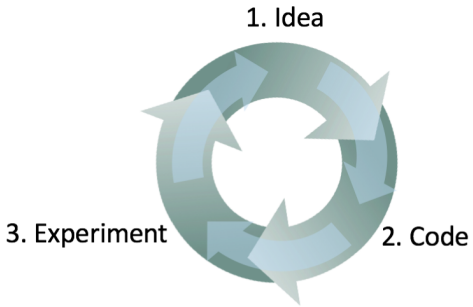
to minimize the false negative rate (optimizing metric), subject to there being no more than one false positive every 24 hours of operation (satisficing metric).

Once your team is aligned on the evaluation metric to optimize, they will be able to make faster progress.

10 Having a dev set and metric speeds up iterations

It is very difficult to know in advance what approach will work best for a new problem. Even experienced machine learning researchers will usually try out dozens of ideas before they discover something satisfactory. When building a machine learning system, I will often:

1. Start off with some **idea** on how to build the system.
2. Implement the idea in **code**.
3. Carry out an **experiment** which tells me how well the idea worked. (Usually my first few ideas don't work!) Based on these learnings, go back to generate more ideas, and keep on iterating.



This is an iterative process. The faster you can go round this loop, the faster you will make progress. This is why having dev/test sets and a metric are important: Each time you try an idea, measuring your idea's performance on the dev set lets you quickly decide if you're heading in the right direction.

In contrast, suppose you don't have a specific dev set and metric. So each time your team develops a new cat classifier, you have to incorporate it into your app, and play with the app for a few hours to get a sense of whether the new classifier is an improvement. This would be incredibly slow! Also, if your team improves the classifier's accuracy from 95.0% to 95.1%, you might not be able to detect that 0.1% improvement from playing with the app. Yet a lot of progress in your system will be made by gradually accumulating dozens of these 0.1% improvements. Having a dev set and metric allows you to very quickly detect which ideas are successfully giving you small (or large) improvements, and therefore lets you quickly decide what ideas to keep refining, and which ones to discard.

11 When to change dev/test sets and metrics

When starting out on a new project, I try to quickly choose dev/test sets, since this gives the team a well-defined target to aim for.

I typically ask my teams to come up with an initial dev/test set and an initial metric in less than one week—rarely longer. It is better to come up with something imperfect and get going quickly, rather than overthink this. But this one week timeline does not apply to mature applications. For example, anti-spam is a mature deep learning application. I have seen teams working on already-mature systems spend months to acquire even better dev/test sets.

If you later realize that your initial dev/test set or metric missed the mark, by all means change them quickly. For example, if your dev set + metric ranks classifier A above classifier B, but your team thinks that classifier B is actually superior for your product, then this might be a sign that you need to change your dev/test sets or your evaluation metric.

There are three main possible causes of the dev set/metric incorrectly rating classifier A higher:

1. The actual distribution you need to do well on is different from the dev/test sets.

Suppose your initial dev/test set had mainly pictures of adult cats. You ship your cat app, and find that users are uploading a lot more kitten images than expected. So, the dev/test set distribution is not representative of the actual distribution you need to do well on. In this case, update your dev/test sets to be more representative.

2. You have overfit to the dev set.

The process of repeatedly evaluating ideas on the dev set causes your algorithm to gradually "overfit" to the dev set. When you are done developing, you will evaluate your system on the test set. If you find that your dev set performance is much better than your test set performance, it is a sign that you have overfit to the dev set. In this case, get a fresh dev set.

If you need to track your team's progress, you can also evaluate your system regularly—say once per week or once per month—on the test set. But do not use the test set to make any decisions regarding the algorithm, including whether to roll back to the previous week's system. If you do so, you will start to overfit to the test set, and can no longer count on it to give a completely unbiased estimate of your system's performance (which you would need if you're publishing research papers, or perhaps using this metric to make important business decisions).

3. The metric is measuring something other than what the project needs to optimize.

Suppose that for your cat application, your metric is classification accuracy. This metric currently ranks classifier A as superior to classifier B. But suppose you try out both algorithms, and find classifier A is allowing occasional pornographic images to slip through. Even though classifier A is more accurate, the bad impression left by the occasional pornographic image means its performance is unacceptable. What do you do?

Here, the metric is failing to identify the fact that Algorithm B is in fact better than Algorithm A for your product. So, you can no longer trust the metric to pick the best algorithm. It is time to change evaluation metrics. For example, you can change the metric to heavily penalize letting through pornographic images. I would strongly recommend picking a new metric and using the new metric to explicitly define a new goal for the team, rather than proceeding for too long without a trusted metric and reverting to manually choosing among classifiers.

It is quite common to change dev/test sets or evaluation metrics during a project. Having an initial dev/test set and metric helps you iterate quickly. If you ever find that the dev/test sets or metric are no longer pointing your team in the right direction, it's not a big deal! Just change them and make sure your team knows about the new direction.

12 Takeaways: Setting up development and test sets

- Choose dev and test sets from a distribution that reflects what data you expect to get in the future and want to do well on. This may not be the same as your training data's distribution.
- Choose dev and test sets from the same distribution if possible.
- Choose a single-number evaluation metric for your team to optimize. If there are multiple goals that you care about, consider combining them into a single formula (such as averaging multiple error metrics) or defining satisficing and optimizing metrics.
- Machine learning is a highly iterative process: You may try many dozens of ideas before finding one that you're satisfied with.
- Having dev/test sets and a single-number evaluation metric helps you quickly evaluate algorithms, and therefore iterate faster.
- When starting out on a brand new application, try to establish dev/test sets and a metric quickly, say in less than a week. It might be okay to take longer on mature applications.
- The old heuristic of a 70%/30% train/test split does not apply for problems where you have a lot of data; the dev and test sets can be much less than 30% of the data.
- Your dev set should be large enough to detect meaningful changes in the accuracy of your algorithm, but not necessarily much larger. Your test set should be big enough to give you a confident estimate of the final performance of your system.
- If your dev set and metric are no longer pointing your team in the right direction, quickly change them: (i) If you had overfit the dev set, get more dev set data. (ii) If the actual distribution you care about is different from the dev/test set distribution, get new dev/test set data. (iii) If your metric is no longer measuring what is most important to you, change the metric.

Basic Error Analysis

13 Build your first system quickly, then iterate

You want to build a new email anti-spam system. Your team has several ideas:

- Collect a huge training set of spam email. For example, set up a "honeypot": deliberately send fake email addresses to known spammers, so that you can automatically harvest the spam messages they send to those addresses.
- Develop features for understanding the text content of the email.
- Develop features for understanding the email header features to show what set of internet servers the message went through.
- and more.

Even though I have worked extensively on anti-spam, I would still have a hard time picking one of these directions. It is even harder if you are not an expert in the application area.

So don't start off trying to design and build the perfect system. Instead, build and train a basic system quickly—perhaps in just a few days.[5] Even if the basic system is far from the "best" system you can build, it is valuable to examine how the basic system functions: you will quickly find clues that show you the most promising directions in which to invest your time. These next few chapters will show you how to read these clues.

[5] This advice is meant for readers wanting to build AI applications, rather than those whose goal is to publish academic papers. I will later return to the topic of doing research.

14 Error analysis: Look at dev set examples to evaluate ideas

When you play with your cat app, you notice several examples where it mistakes dogs for cats. Some dogs do look like cats!

A team member proposes incorporating 3rd party software that will make the system do better on dog images. These changes will take a month, and the team member is enthusiastic. Should you ask them to go ahead?

Before investing a month on this task, I recommend that you first estimate how much it will actually improve the system's accuracy. Then you can more rationally decide if this is worth the month of development time, or if you're better off using that time on other tasks.

In detail, here's what you can do:

1. Gather a sample of 100 dev set examples that your system *misclassified*. I.e., examples that your system made an error on.
2. Look at these examples manually, and count what fraction of them are dog images.

The process of looking at misclassified examples is called **error analysis**. In this example, if you find that only 5% of the misclassified images are dogs, then no matter how much you improve your algorithm's performance on dog images, you won't get rid of more than 5% of your errors. In other words, 5% is a "ceiling" (meaning maximum possible amount) for how much the proposed project could help. Thus, if your overall system is currently 90% accurate (10% error), this improvement is likely to result in at best 90.5% accuracy (or 9.5% error, which is 5% less error than the original 10% error).

In contrast, if you find that 50% of the mistakes are dogs, then you can be more confident that the proposed project will have a big impact. It could boost accuracy from 90% to 95% (a 50% relative reduction in error, from 10% down to 5%).

This simple counting procedure of error analysis gives you a quick way to estimate the possible value of incorporating the 3rd party software for dog images. It provides a quantitative basis on which to decide whether to make this investment.

Error analysis can often help you figure out how promising different directions are. I've seen many engineers reluctant to carry out error analysis. It often feels more exciting to jump in and implement some idea, rather than question if the idea is worth the time investment. This is a common mistake: It might result in your team spending a month only to realize afterward that it resulted in little benefit.

Manually examining 100 examples does not take long. Even if you take one minute per image, you'd be done in under two hours. These two hours could save you a month of wasted effort.

Error Analysis refers to the process of examining dev set examples that your algorithm misclassified, so that you can understand the underlying causes of the errors. This can help you prioritize projects—as in this example—and inspire new directions, which we will discuss next. The next few chapters will also present best practices for carrying out error analyses.

15 Evaluating multiple ideas in parallel during error analysis

Your team has several ideas for improving the cat detector:

- Fix the problem of your algorithm recognizing *dogs* as cats.
- Fix the problem of your algorithm recognizing *great cats* (lions, panthers, etc.) as house cats (pets).
- Improve the system's performance on *blurry* images.
- ...

You can efficiently evaluate all of these ideas in parallel. I usually create a spreadsheet and fill it out while looking through ~100 misclassified dev set images. I also jot down comments that might help me remember specific examples. To illustrate this process, let's look at a spreadsheet you might produce with a small dev set of four examples:

Image	Dog	Great cat	Blurry	Comments
1	✔			Unusual pitbull color
2			✔	
3		✔	✔	Lion; picture taken at zoo on rainy day
4		✔		Panther behind tree
% of total	25%	50%	50%	

Image #3 above has both the Great Cat and the Blurry columns checked. Furthermore, because it is possible for one example to be associated with multiple categories, the percentages at the bottom may not add up to 100%.

Although you may first formulate the categories (Dog, Great cat, Blurry) then categorize the examples by hand, in practice, once you start looking through examples, you will probably be inspired to propose new error categories. For example, say you go through a dozen images and realize a lot of mistakes occur with Instagram-filtered pictures. You can go back and add a new "Instagram" column to the spreadsheet. Manually looking at examples that the algorithm misclassified and asking how/whether you as a human could have labeled the

picture correctly will often inspire you to come up with new categories of errors and solutions.

The most helpful error categories will be ones that you have an idea for improving. For example, the Instagram category will be most helpful to add if you have an idea to "undo" Instagram filters and recover the original image. But you don't have to restrict yourself only to error categories you know how to improve; the goal of this process is to build your intuition about the most promising areas to focus on.

Error analysis is an iterative process. Don't worry if you start off with no categories in mind. After looking at a couple of images, you might come up with a few ideas for error categories. After manually categorizing some images, you might think of new categories and re-examine the images in light of the new categories, and so on.

Suppose you finish carrying out error analysis on 100 misclassified dev set examples and get the following:

Image	Dog	Great cat	Blurry	Comments
1	✔			Usual pitbull color
2			✔	
3		✔	✔	Lion; picture taken at zoo on rainy day
4		✔		Panther behind tree
...	...	...	...	...
% of total	8%	43%	61%	

You now know that working on a project to address the Dog mistakes can eliminate 8% of the errors at most. Working on Great Cat or Blurry image errors could help eliminate more errors. Therefore, you might pick one of the two latter categories to focus on. If your team has enough people to pursue multiple directions in parallel, you can also ask some engineers to work on Great Cats and others to work on Blurry images.

Error analysis does not produce a rigid mathematical formula that tells you what the highest priority task should be. You also have to take into account how much progress you expect to make on different categories and the amount of work needed to tackle each one.

16 Cleaning up mislabeled dev and test set examples

During error analysis, you might notice that some examples in your dev set are mislabeled. When I say "mislabeled" here, I mean that the pictures were already mislabeled by a human labeler even before the algorithm encountered it. I.e., the class label in an example *(x,y)* has an incorrect value for *y*. For example, perhaps some pictures that are not cats are mislabeled as containing a cat, and vice versa. If you suspect the fraction of mislabeled images is significant, add a category to keep track of the fraction of examples mislabeled:

Image	Dog	Great cat	Blurry	Mislabeled	Comments
...					
98				✔	Labeler missed cat in background
99		✔			
100				✔	Drawing of a cat; not a real cat.
% of total	8%	43%	61%	6%	

Should you correct the labels in your dev set? Remember that the goal of the dev set is to help you quickly evaluate algorithms so that you can tell if Algorithm A or B is better. If the fraction of the dev set that is mislabeled impedes your ability to make these judgments, then it is worth spending time to fix the mislabeled dev set labels.

For example, suppose your classifier's performance is:

- Overall accuracy on dev set..... 90% (10% overall error.)
- Errors due to mislabeled examples..... 0.6% (6% of dev set errors.)
- Errors due to other causes..... 9.4% (94% of dev set errors)

Here, the 0.6% inaccuracy due to mislabeling might not be significant enough relative to the 9.4% of errors you could be improving. There is no harm in manually fixing the mislabeled images in the dev set, but it is not crucial to do so: It might be fine not knowing whether your system has 10% or 9.4% overall error.

Suppose you keep improving the cat classifier and reach the following performance:

- Overall accuracy on dev set..... 98.0% (2.0% overall error.)
- Errors due to mislabeled examples..... 0.6%. (30% of dev set errors.)
- Errors due to other causes..... 1.4% (70% of dev set errors)

30% of your errors are due to the mislabeled dev set images, adding significant error to your estimates of accuracy. It is now worthwhile to improve the quality of the labels in the dev set. Tackling the mislabeled examples will help you figure out if a classifier's error is closer to 1.4% or 2%—a significant relative difference.

It is not uncommon to start off tolerating some mislabeled dev/test set examples, only later to change your mind as your system improves so that the fraction of mislabeled examples grows relative to the total set of errors.

The last chapter explained how you can improve error categories such as Dog, Great Cat and Blurry through algorithmic improvements. You have learned in this chapter that you can work on the Mislabeled category as well—through improving the data's labels.

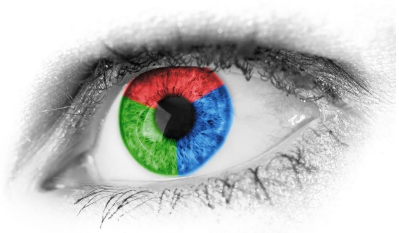
Whatever process you apply to fixing dev set labels, remember to apply it to the test set labels too so that your dev and test sets continue to be drawn from the same distribution. Fixing your dev and test sets together would prevent the problem we discussed in Chapter 6, where your team optimizes for dev set performance only to realize later that they are being judged on a different criterion based on a different test set.

If you decide to improve the label quality, consider double-checking both the labels of examples that your system misclassified as well as labels of examples it correctly classified. It is possible that both the original label and your learning algorithm were wrong on an example. If you fix only the labels of examples that your system had misclassified, you might introduce bias into your evaluation. If you have 1,000 dev set examples, and if your classifier has 98.0% accuracy, it is easier to examine the 20 examples it misclassified than to examine all 980 examples classified correctly. Because it is easier in practice to check only the misclassified examples, bias does creep into some dev sets. This bias is acceptable if you are interested only in developing a product or application, but it would be a problem if you plan to use the result in an academic research paper or need a completely unbiased measure of test set accuracy.

17 If you have a large dev set, split it into two subsets, only one of which you look at

Suppose you have a large dev set of 5,000 examples in which you have a 20% error rate. Thus, your algorithm is misclassifying ~1,000 dev images. It takes a long time to manually examine 1,000 images, so we might decide not to use all of them in the error analysis.

In this case, I would explicitly split the dev set into two subsets, one of which you look at, and one of which you don't. You will more rapidly overfit the portion that you are manually looking at. You can use the portion you are not manually looking at to tune parameters.



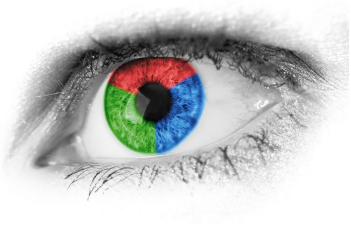
Let's continue our example above, in which the algorithm is misclassifying 1,000 out of 5,000 dev set examples. Suppose we want to manually examine about 100 errors for error analysis (10% of the errors). You should randomly select 10% of the dev set and place that into what we'll call an **Eyeball dev set** to remind ourselves that we are looking at it with our eyes. (For a project on speech recognition, in which you would be listening to audio clips, perhaps you would call this set an Ear dev set instead). The Eyeball dev set therefore has 500 examples, of which we would expect our algorithm to misclassify about 100.

The second subset of the dev set, called the **Blackbox dev set**, will have the remaining 4500 examples. You can use the Blackbox dev set to evaluate classifiers automatically by measuring their error rates. You can also use it to select among algorithms or tune hyperparameters. However, you should avoid looking at it with your eyes. We use the term "Blackbox" because we will only use this subset of the data to obtain "Blackbox" evaluations of classifiers.

Why do we explicitly separate the dev set into Eyeball and Blackbox dev sets? Since you will gain intuition about the examples in the Eyeball dev set, you will start to overfit the Eyeball dev set faster. If you see the performance on the Eyeball dev set improving much more rapidly than the performance on the Blackbox dev set, you have overfit the Eyeball dev set. In this case, you might need to discard it and find a new Eyeball dev set by moving more examples from the Blackbox dev set into the Eyeball dev set or by acquiring new labeled data.

Explicitly splitting your dev set into Eyeball and Blackbox dev sets allows you to tell when your manual error analysis process is causing you to overfit the Eyeball portion of your data.

18 How big should the Eyeball and Blackbox dev sets be?



Your Eyeball dev set should be large enough to give you a sense of your algorithm's major error categories. If you are working on a task that humans do well (such as recognizing cats in images), here are some rough guidelines:

- An eyeball dev set in which your classifier makes 10 mistakes would be considered very small. With just 10 errors, it's hard to accurately estimate the impact of different error categories. But if you have very little data and cannot afford to put more into the Eyeball dev set, it's better than nothing and will help with project prioritization.
- If your classifier makes ~20 mistakes on eyeball dev examples, you would start to get a rough sense of the major error sources.
- With ~50 mistakes, you would get a good sense of the major error sources.
- With ~100 mistakes, you would get a very good sense of the major sources of errors. I've seen people manually analyze even more errors—sometimes as many as 500. There is no harm in this as long as you have enough data.

Say your classifier has a 5% error rate. To make sure you have ~100 mislabeled examples in the Eyeball dev set, the Eyeball dev set would have to have about 2,000 examples (since 0.05*2,000 = 100). The lower your classifier's error rate, the larger your Eyeball dev set needs to be in order to get a large enough set of errors to analyze.

If you are working on a task that even humans cannot do well, then the exercise of examining an Eyeball dev set will not be as helpful because it is harder to figure out why the algorithm didn't classify an example correctly. In this case, you might omit having an Eyeball dev set. We discuss guidelines for such problems in a later chapter.

How about the Blackbox dev set? We previously said that dev sets of around 1,000-10,000 examples are common. To refine that statement, a Blackbox dev set of 1,000-10,000 examples will often give you enough data to tune hyperparameters and select among models, though there is little harm in having even more data. A Blackbox dev set of 100 would be small but still useful.

If you have a small dev set, then you might not have enough data to split into Eyeball and Blackbox dev sets that are both large enough to serve their purposes. Instead, your entire dev set might have to be used as the Eyeball dev set—i.e., you would manually examine all the dev set data.

Between the Eyeball and Blackbox dev sets, I consider the Eyeball dev set more important (assuming that you are working on a problem that humans can solve well and that examining the examples helps you gain insight). If you only have an Eyeball dev set, you can perform error analyses, model selection and hyperparameter tuning all on that set. The downside of having only an Eyeball dev set is that the risk of overfitting the dev set is greater.

If you have plentiful access to data, then the size of the Eyeball dev set would be determined mainly by how many examples you have time to manually analyze. For example, I've rarely seen anyone manually analyze more than 1,000 errors.

19 Takeaways: Basic error analysis

- When you start a new project, especially if it is in an area in which you are not an expert, it is hard to correctly guess the most promising directions.
- So don't start off trying to design and build the perfect system. Instead build and train a basic system as quickly as possible—perhaps in a few days. Then use error analysis to help you identify the most promising directions and iteratively improve your algorithm from there.
- Carry out error analysis by manually examining ~100 dev set examples the algorithm misclassifies and counting the major categories of errors. Use this information to prioritize what types of errors to work on fixing.
- Consider splitting the dev set into an Eyeball dev set, which you will manually examine, and a Blackbox dev set, which you will not manually examine. If performance on the Eyeball dev set is much better than the Blackbox dev set, you have overfit the Eyeball dev set and should consider acquiring more data for it.
- The Eyeball dev set should be big enough so that your algorithm misclassifies enough examples for you to analyze. A Blackbox dev set of 1,000-10,000 examples is sufficient for many applications.
- If your dev set is not big enough to split this way, just use an Eyeball dev set for manual error analysis, model selection, and hyperparameter tuning.

Bias and Variance

20 Bias and Variance: The two big sources of error

Suppose your training, dev and test sets all come from the same distribution. Then you should always try to get more training data, since that can only improve performance, right?

Even though having more data can't hurt, unfortunately it doesn't always help as much as you might hope. It could be a waste of time to work on getting more data. So, how do you decide when to add data, and when not to bother?

There are two major sources of error in machine learning: bias and variance. Understanding them will help you decide whether adding data, as well as other tactics to improve performance, are a good use of time.

Suppose you hope to build a cat recognizer that has 5% error. Right now, your training set has an error rate of 15%, and your dev set has an error rate of 16%. In this case, adding training data probably won't help much. You should focus on other changes. Indeed, adding more examples to your training set only makes it harder for your algorithm to do well on the training set. (We explain why in a later chapter.)

If your error rate on the training set is 15% (or 85% accuracy), but your target is 5% error (95% accuracy), then the first problem to solve is to improve your algorithm's performance on your training set. Your dev/test set performance is usually worse than your training set performance. So if you are getting 85% accuracy on the examples your algorithm has seen, there's no way you're getting 95% accuracy on examples your algorithm hasn't even seen.

Suppose as above that your algorithm has 16% error (84% accuracy) on the dev set. We break the 16% error into two components:

- First, the algorithm's error rate on the training set. In this example, it is 15%. We think of this informally as the algorithm's **bias**.
- Second, how much worse the algorithm does on the dev (or test) set than the training set. In this example, it does 1% worse on the dev set than the training set. We think of this informally as the algorithm's **variance**.[1]

[1] The field of statistics has more formal definitions of Bias and Variance that we won't worry about. Roughly, the Bias is the error rate of your algorithm on your training set when you have a very large training set. The Variance is how much worse you do on the test set compared to the training set in

Some changes to a learning algorithm can address the first component of error—**bias**—and improve its performance on the training set. Some changes address the second component—**variance**—and help it generalize better from the training set to the dev/test sets.[2] To select the most promising changes, it is incredibly useful to understand which of these two components of error is more pressing to address.

Developing good intuition about Bias and Variance will help you choose effective changes for your algorithm.

this setting. When your error metric is mean squared error, you can write down formulas specifying these two quantities, and prove that Total Error = Bias + Variance. But for our purposes of deciding how to make progress on an ML problem, the more informal definition of Bias and Variance given here will suffice.

[2] There are also some methods that can simultaneously reduce Bias and Variance, by making major changes to the system architecture. But these tend to be harder to identify and implement.

21 Examples of Bias and Variance

Consider our cat classification task. An "ideal" classifier (such as a human) might achieve nearly perfect performance in this task.

Suppose your algorithm performs as follows:

- Training error = 1%
- Dev error = 11%

What problem does it have? Applying the definitions from the previous chapter, we estimate the bias as 1%, and the variance as 10% (=11%-1%). Thus, it has **high variance**. The classifier has very low training error, but it is failing to generalize to the dev set. This is also called **overfitting**.

Now consider this:

- Training error = 15%
- Dev error = 16%

We estimate the bias as 15%, and variance as 1%. This classifier is fitting the training set poorly with 15% error, but its error on the dev set is barely higher than the training error. This classifier therefore has **high bias**, but low variance. We say that this algorithm is **underfitting**.

Now, consider this:

- Training error = 15%
- Dev error = 30%

We estimate the bias as 15%, and variance as 15%. This classifier has **high bias and high variance**: It is doing poorly on the training set, and therefore has high bias, and its performance on the dev set is even worse, so it also has high variance. The overfitting/underfitting terminology is hard to apply here since the classifier is simultaneously overfitting and underfitting.

Finally, consider this:

- Training error = 0.5%
- Dev error = 1%

This classifier is doing well, as it has low bias and low variance. Congratulations on achieving this great performance!

22 Comparing to the optimal error rate

In our cat recognition example, the "ideal" error rate—that is, one achievable by an "optimal" classifier—is nearly 0%. A human looking at a picture would be able to recognize if it contains a cat almost all the time; thus, we can hope for a machine that would do just as well.

Other problems are harder. For example, suppose that you are building a speech recognition system, and find that 14% of the audio clips have so much background noise or are so unintelligible that even a human cannot recognize what was said. In this case, even the most "optimal" speech recognition system might have error around 14%.

Suppose that on this speech recognition problem, your algorithm achieves:

- Training error = 15%
- Dev error = 30%

The training set performance is already close to the optimal error rate of 14%. Thus, there is not much room for improvement in terms of bias or in terms of training set performance. However, this algorithm is not generalizing well to the dev set; thus there is ample room for improvement in the errors due to variance.

This example is similar to the third example from the previous chapter, which also had a training error of 15% and dev error of 30%. If the optimal error rate is ~0%, then a training error of 15% leaves much room for improvement. This suggests bias-reducing changes might be fruitful. But if the optimal error rate is 14%, then the same training set performance tells us that there's little room for improvement in the classifier's bias.

For problems where the optimal error rate is far from zero, here's a more detailed breakdown of an algorithm's error. Continuing with our speech recognition example above, the total dev set error of 30% can be broken down as follows (a similar analysis can be applied to the test set error):

- **Optimal error rate ("unavoidable bias")**: 14%. Suppose we decide that, even with the best possible speech system in the world, we would still suffer 14% error. We can think of this as the "unavoidable" part of a learning algorithm's bias.

- **Avoidable bias**: 1%. This is calculated as the difference between the training error and the optimal error rate.[3]
- **Variance**: 15%. The difference between the dev error and the training error.

To relate this to our earlier definitions, Bias and Avoidable Bias are related as follows:[4]

Bias = Optimal error rate ("unavoidable bias") + Avoidable bias

The "avoidable bias" reflects how much worse your algorithm performs on the training set than the "optimal classifier."

The concept of variance remains the same as before. In theory, we can always reduce variance to nearly zero by training on a massive training set. Thus, all variance is "avoidable" with a sufficiently large dataset, so there is no such thing as "unavoidable variance."

Consider one more example, where the optimal error rate is 14%, and we have:

- Training error = 15%
- Dev error = 16%

Whereas in the previous chapter we called this a high bias classifier, now we would say that error from avoidable bias is 1%, and the error from variance is about 1%. Thus, the algorithm is already doing well, with little room for improvement. It is only 2% worse than the optimal error rate.

We see from these examples that knowing the optimal error rate is helpful for guiding our next steps. In statistics, the optimal error rate is also called **Bayes error rate**, or Bayes rate.

How do we know what the optimal error rate is? For tasks that humans are reasonably good at, such as recognizing pictures or transcribing audio clips, you can ask a human to provide labels then measure the accuracy of the human labels relative to your training set. This would give an estimate of the optimal error rate. If you are working on a problem that even

[3] If this number is negative, you are doing better on the training set than the optimal error rate. This means you are overfitting on the training set, and the algorithm has over-memorized the training set. You should focus on variance reduction methods rather than on further bias reduction methods.

[4] These definitions are chosen to convey insight on how to improve your learning algorithm. These definitions are different than how statisticians define Bias and Variance. Technically, what I define here as "Bias" should be called "Error we attribute to bias"; and "Avoidable bias" should be "error we attribute to the learning algorithm's bias that is over the optimal error rate."

humans have a hard time solving (e.g., predicting what movie to recommend, or what ad to show to a user) it can be hard to estimate the optimal error rate.

In the section "Comparing to Human-Level Performance (Chapters 33 to 35), I will discuss in more detail the process of comparing a learning algorithm's performance to human-level performance.

In the last few chapters, you learned how to estimate avoidable/unavoidable bias and variance by looking at training and dev set error rates. The next chapter will discuss how you can use insights from such an analysis to prioritize techniques that reduce bias vs. techniques that reduce variance. There are very different techniques that you should apply depending on whether your project's current problem is high (avoidable) bias or high variance. Read on!

23 Addressing Bias and Variance

Here is the simplest formula for addressing bias and variance issues:

- If you have high avoidable bias, increase the size of your model (for example, increase the size of your neural network by adding layers/neurons).
- If you have high variance, add data to your training set.

If you are able to increase the neural network size and increase training data without limit, it is possible to do very well on many learning problems.

In practice, increasing the size of your model will eventually cause you to run into computational problems because training very large models is slow. You might also exhaust your ability to acquire more training data. (Even on the internet, there is only a finite number of cat pictures!)

Different model architectures—for example, different neural network architectures—will have different amounts of bias/variance for your problem. A lot of recent deep learning research has developed many innovative model architectures. So if you are using neural networks, the academic literature can be a great source of inspiration. There are also many great open-source implementations on github. But the results of trying new architectures are less predictable than the simple formula of increasing the model size and adding data.

Increasing the model size generally reduces bias, but it might also increase variance and the risk of overfitting. However, this overfitting problem usually arises only when you are not using regularization. If you include a well-designed regularization method, then you can usually safely increase the size of the model without increasing overfitting.

Suppose you are applying deep learning, with L2 regularization or dropout, with the regularization parameter that performs best on the dev set. If you increase the model size, usually your performance will stay the same or improve; it is unlikely to worsen significantly. The only reason to avoid using a bigger model is the increased computational cost.

24 Bias vs. Variance tradeoff

You might have heard of the "Bias vs. Variance tradeoff." Of the changes you could make to most learning algorithms, there are some that reduce bias errors but at the cost of increasing variance, and vice versa. This creates a "trade off" between bias and variance.

For example, increasing the size of your model—adding neurons/layers in a neural network, or adding input features—generally reduces bias but could increase variance. Alternatively, adding regularization generally increases bias but reduces variance.

In the modern era, we often have access to plentiful data and can use very large neural networks (deep learning). Therefore, there is less of a tradeoff, and there are now more options for reducing bias without hurting variance, and vice versa.

For example, you can usually increase a neural network size and tune the regularization method to reduce bias without noticeably increasing variance. By adding training data, you can also usually reduce variance without affecting bias.

If you select a model architecture that is well suited for your task, you might also reduce bias and variance simultaneously. Selecting such an architecture can be difficult.

In the next few chapters, we discuss additional specific techniques for addressing bias and variance.

25 Techniques for reducing avoidable bias

If your learning algorithm suffers from high avoidable bias, you might try the following techniques:

- **Increase the model size** (such as number of neurons/layers): This technique reduces bias, since it should allow you to fit the training set better. If you find that this increases variance, then use regularization, which will usually eliminate the increase in variance.
- **Modify input features based on insights from error analysis**: Say your error analysis inspires you to create additional features that help the algorithm eliminate a particular category of errors. (We discuss this further in the next chapter.) These new features could help with both bias and variance. In theory, adding more features could increase the variance; but if you find this to be the case, then use regularization, which will usually eliminate the increase in variance.
- **Reduce or eliminate regularization** (L2 regularization, L1 regularization, dropout): This will reduce avoidable bias, but increase variance.
- **Modify model architecture** (such as neural network architecture) so that it is more suitable for your problem: This technique can affect both bias and variance.

One method that is not helpful:

- **Add more training data**: This technique helps with variance problems, but it usually has no significant effect on bias.

26 Error analysis on the training set

Your algorithm must perform well on the training set before you can expect it to perform well on the dev/test sets.

In addition to the techniques described earlier to address high bias, I sometimes also carry out an error analysis on the *training data*, following a protocol similar to error analysis on the Eyeball dev set. This can be useful if your algorithm has high bias—i.e., if it is not fitting the training set well.

For example, suppose you are building a speech recognition system for an app and have collected a training set of audio clips from volunteers. If your system is not doing well on the training set, you might consider listening to a set of ~100 examples that the algorithm is doing poorly on to understand the major categories of training set errors. Similar to the dev set error analysis, you can count the errors in different categories:

Audio clip	Loud background noise	User spoke quickly	Far from microphone	Comments
1	✔			Car noise
2	✔		✔	Restaurant noise
3		✔	✔	User shouting across living room?
4	✔			Coffeeshop
% of total	75%	25%	50%	

In this example, you might realize that your algorithm is having a particularly hard time with training examples that have a lot of background noise. Thus, you might focus on techniques that allow it to better fit training examples with background noise.

You might also double-check whether it is possible for a person to transcribe these audio clips, given the same input audio as your learning algorithm. If there is so much background noise that it is simply impossible for anyone to make out what was said, then it might be unreasonable to expect any algorithm to correctly recognize such utterances. We will discuss the benefits of comparing your algorithm to human-level performance in a later section.

27 Techniques for reducing variance

If your learning algorithm suffers from high variance, you might try the following techniques:

- **Add more training data**: This is the simplest and most reliable way to address variance, so long as you have access to significantly more data and enough computational power to process the data.
- **Add regularization** (L2 regularization, L1 regularization, dropout): This technique reduces variance but increases bias.
- **Add early stopping** (i.e., stop gradient descent early, based on dev set error): This technique reduces variance but increases bias. Early stopping behaves a lot like regularization methods, and some authors call it a regularization technique.
- **Feature selection to decrease number/type of input features:** This technique might help with variance problems, but it might also increase bias. Reducing the number of features slightly (say going from 1,000 features to 900) is unlikely to have a huge effect on bias. Reducing it significantly (say going from 1,000 features to 100—a 10x reduction) is more likely to have a significant effect, so long as you are not excluding too many useful features. In modern deep learning, when data is plentiful, there has been a shift away from feature selection, and we are now more likely to give all the features we have to the algorithm and let the algorithm sort out which ones to use based on the data. But when your training set is small, feature selection can be very useful.
- **Decrease the model size** (such as number of neurons/layers): *Use with caution*. This technique could decrease variance, while possibly increasing bias. However, I don't recommend this technique for addressing variance. Adding regularization usually gives better classification performance. The advantage of reducing the model size is reducing your computational cost and thus speeding up how quickly you can train models. If speeding up model training is useful, then by all means consider decreasing the model size. But if your goal is to reduce variance, and you are not concerned about the computational cost, consider adding regularization instead.

Here are two additional tactics, repeated from the previous chapter on addressing bias:

- **Modify input features based on insights from error analysis**: Say your error analysis inspires you to create additional features that help the algorithm to eliminate a particular category of errors. These new features could help with both bias and variance. In

theory, adding more features could increase the variance; but if you find this to be the case, then use regularization, which will usually eliminate the increase in variance.

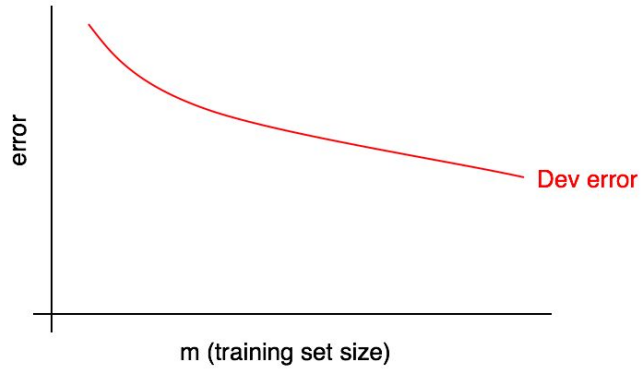
- **Modify model architecture** (such as neural network architecture) so that it is more suitable for your problem: This technique can affect both bias and variance.

Learning curves

28 Diagnosing bias and variance: Learning curves

We've seen some ways to estimate how much error can be attributed to avoidable bias vs. variance. We did so by estimating the optimal error rate and computing the algorithm's training set and dev set errors. Let's discuss a technique that is even more informative: plotting a learning curve.

A learning curve plots your dev set error against the number of training examples. To plot it, you would run your algorithm using different training set sizes. For example, if you have 1,000 examples, you might train separate copies of the algorithm on 100, 200, 300, ..., 1000 examples. Then you could plot how dev set error varies with the training set size. Here is an example:



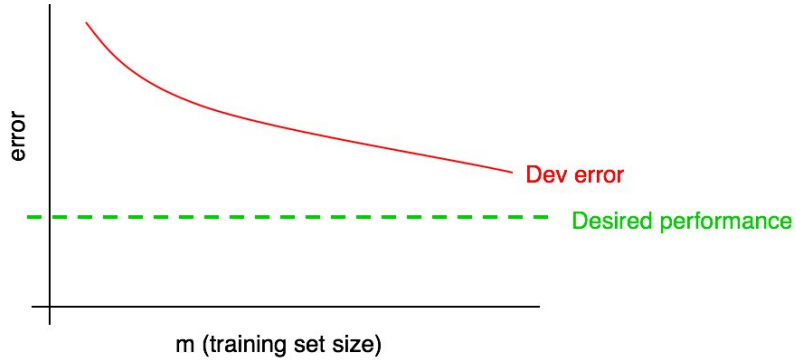
As the training set size increases, the dev set error should decrease.

We will often have some "desired error rate" that we hope our learning algorithm will eventually achieve. For example:

- If we hope for human-level performance, then the human error rate could be the "desired error rate."
- If our learning algorithm serves some product (such as delivering cat pictures), we might have an intuition about what level of performance is needed to give users a great experience.

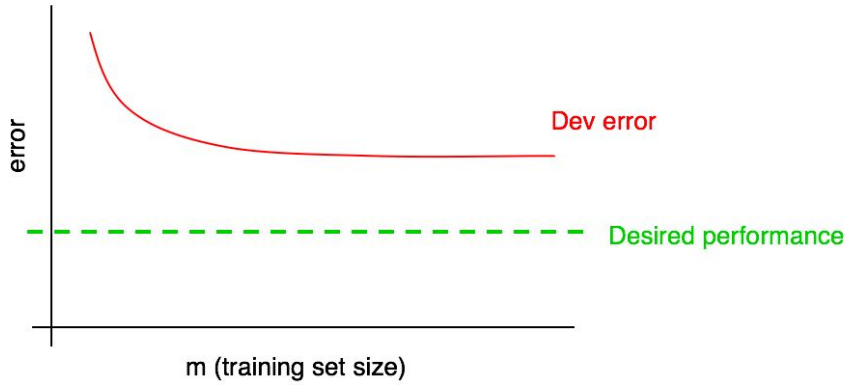
- If you have worked on a important application for a long time, then you might have intuition about how much more progress you can reasonably make in the next quarter/year.

Add the desired level of performance to your learning curve:



You can visually extrapolate the red "dev error" curve to guess how much closer you could get to the desired level of performance by adding more data. In the example above, it looks plausible that doubling the training set size might allow you to reach the desired performance.

But if the dev error curve has "plateaued" (i.e. flattened out), then you can immediately tell that adding more data won't get you to your goal:



Looking at the learning curve might therefore help you avoid spending months collecting twice as much training data, only to realize it does not help.

One downside of this process is that if you only look at the dev error curve, it can be hard to extrapolate and predict exactly where the red curve will go if you had more data. There is one additional plot that can help you estimate the impact of adding more data: the training error.

29 Plotting training error

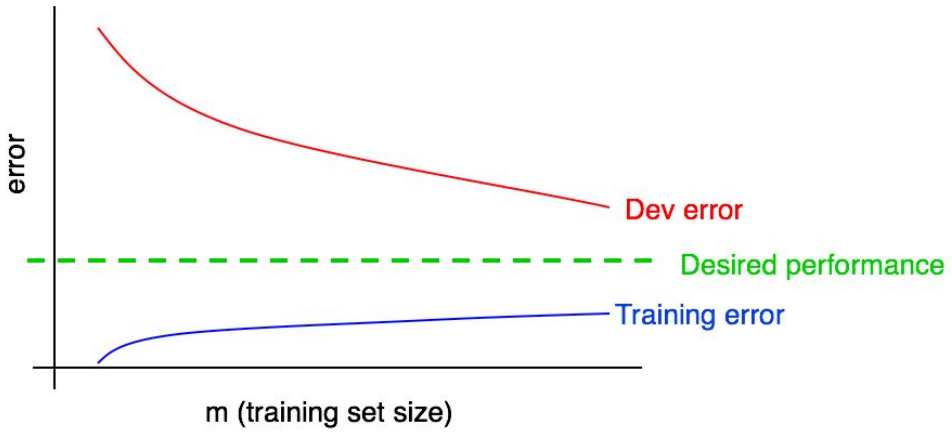
Your dev set (and test set) error should decrease as the training set size grows. But your training set error usually *increases* as the training set size grows.

Let's illustrate this effect with an example. Suppose your training set has only 2 examples: One cat image and one non-cat image. Then it is easy for the learning algorithms to "memorize" both examples in the training set, and get 0% training set error. Even if either or both of the training examples were mislabeled, it is still easy for the algorithm to memorize both labels.

Now suppose your training set has 100 examples. Perhaps even a few examples are mislabeled, or ambiguous—some images are very blurry, so even humans cannot tell if there is a cat. Perhaps the learning algorithm can still "memorize" most or all of the training set, but it is now harder to obtain 100% accuracy. By increasing the training set from 2 to 100 examples, you will find that the training set accuracy will drop slightly.

Finally, suppose your training set has 10,000 examples. In this case, it becomes even harder for the algorithm to perfectly fit all 10,000 examples, especially if some are ambiguous or mislabeled. Thus, your learning algorithm will do even worse on this training set.

Let's add a plot of training error to our earlier figures:

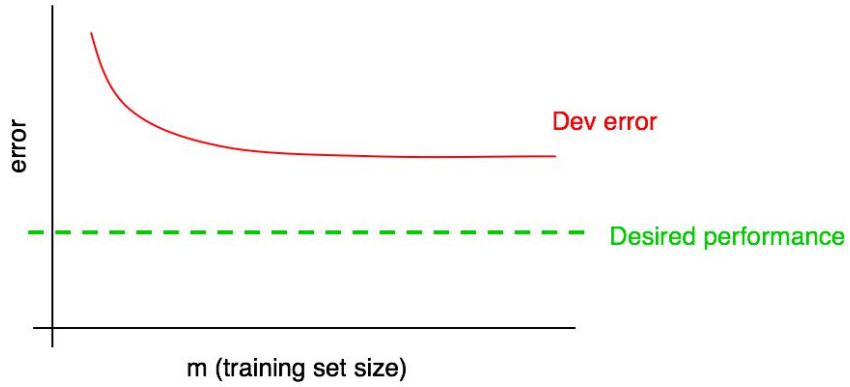


You can see that the blue "training error" curve increases with the size of the training set. Furthermore, your algorithm usually does better on the training set than on the dev set; thus the red dev error curve usually lies strictly above the blue training error curve.

Let's discuss next how to interpret these plots.

30 Interpreting learning curves: High bias

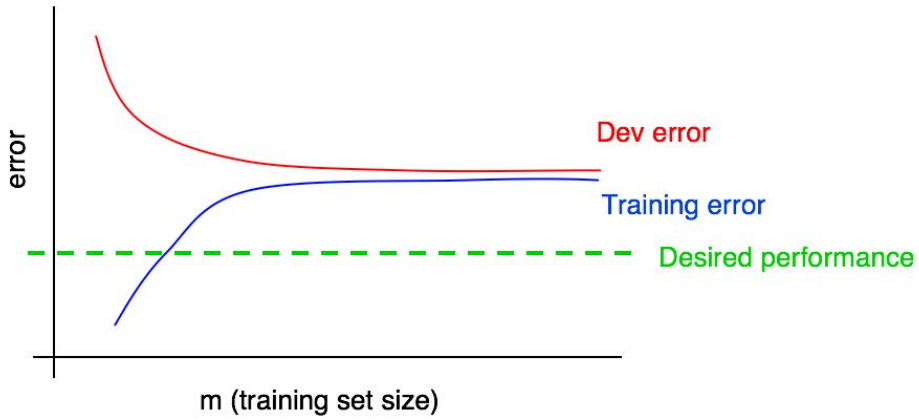
Suppose your dev error curve looks like this:



We previously said that, if your dev error curve plateaus, you are unlikely to achieve the desired performance just by adding data.

But it is hard to know exactly what an extrapolation of the red dev error curve will look like. If the dev set was small, you would be even less certain because the curves could be noisy.

Suppose we add the training error curve to this plot and get the following:



Now, you can be absolutely sure that adding more data will not, by itself, be sufficient. Why is that? Remember our two observations:

- As we add more training data, training error can only get worse. Thus, the blue training error curve can only stay the same or go higher, and thus it can only get further away from the (green line) level of desired performance.
- The red dev error curve is usually higher than the blue training error. Thus, there's almost no way that adding more data would allow the red dev error curve to drop down to the desired level of performance when even the training error is higher than the desired level of performance.

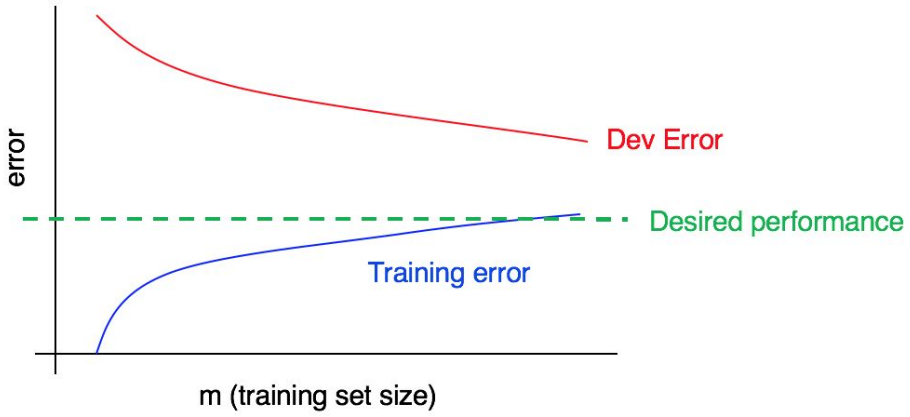
Examining both the dev error curve and the training error curve on the same plot allows us to more confidently extrapolate the dev error curve.

Suppose, for the sake of discussion, that the desired performance is our estimate of the optimal error rate. The figure above is then the standard "textbook" example of what a learning curve with high avoidable bias looks like: At the largest training set size—presumably corresponding to all the training data we have—there is a large gap between the training error and the desired performance, indicating large avoidable bias. Furthermore, the gap between the training and dev curves is small, indicating small variance.

Previously, we were measuring training and dev set error only at the rightmost point of this plot, which corresponds to using all the available training data. Plotting the full learning curve gives us a more comprehensive picture of the algorithms' performance on different training set sizes.

31 Interpreting learning curves: Other cases

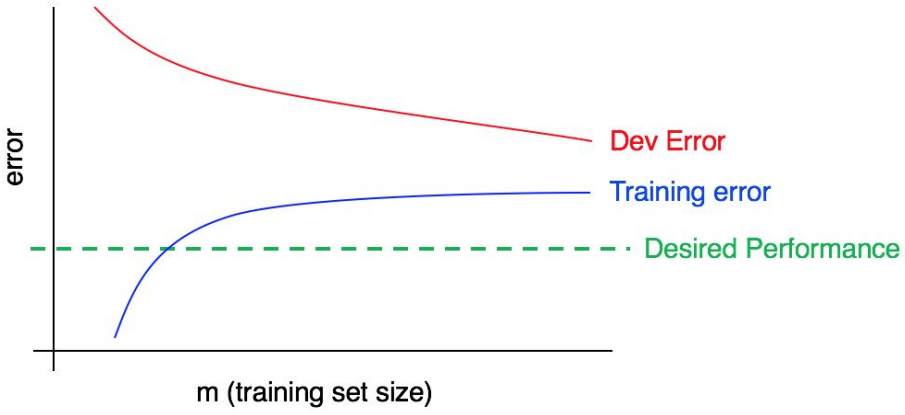
Consider this learning curve:



Does this plot indicate high bias, high variance, or both?

The blue training error curve is relatively low, and the red dev error curve is much higher than the blue training error. Thus, the bias is small, but the variance is large. Adding more training data will probably help close the gap between dev error and training error.

Now, consider this:



This time, the training error is large, as it is much higher than the desired level of performance. The dev error is also much larger than the training error. Thus, you have significant bias and significant variance. You will have to find a way to reduce both bias and variance in your algorithm.

32 Plotting learning curves

Suppose you have a very small training set of 100 examples. You train your algorithm using a randomly chosen subset of 10 examples, then 20 examples, then 30, up to 100, increasing the number of examples by intervals of ten. You then use these 10 data points to plot your learning curve. You might find that the curve looks slightly noisy (meaning that the values are higher/lower than expected) at the smaller training set sizes.

When training on just 10 randomly chosen examples, you might be unlucky and have a particularly "bad" training set, such as one with many ambiguous/mislabeled examples. Or, you might get lucky and get a particularly "good" training set. Having a small training set means that the dev and training errors may randomly fluctuate.

If your machine learning application is heavily skewed toward one class (such as a cat classification task where the fraction of negative examples is much larger than positive examples), or if it has a huge number of classes (such as recognizing 100 different animal species), then the chance of selecting an especially "unrepresentative" or bad training set is also larger. For example, if 80% of your examples are negative examples (y=0), and only 20% are positive examples (y=1), then there is a chance that a training set of 10 examples contains only negative examples, thus making it very difficult for the algorithm to learn something meaningful.

If the noise in the training curve makes it hard to see the true trends, here are two solutions:

- Instead of training just one model on 10 examples, instead select several (say 3-10) different randomly chosen training sets of 10 examples by sampling with replacement[1] from your original set of 100. Train a different model on each of these, and compute the training and dev set error of each of the resulting models. Compute and plot the average training error and average dev set error.
- If your training set is skewed towards one class, or if it has many classes, choose a "balanced" subset instead of 10 training examples at random out of the set of 100. For example, you can make sure that 2/10 of the examples are positive examples, and 8/10 are

[1] Here's what sampling *with replacement* means: You would randomly pick 10 different examples out of the 100 to form your first training set. Then to form the second training set, you would again pick 10 examples, but without taking into account what had been chosen in the first training set. Thus, it is possible for one specific example to appear in both the first and second training sets. In contrast, if you were sampling *without replacement*, the second training set would be chosen from just the 90 examples that had not been chosen the first time around. In practice, sampling with or without replacement shouldn't make a huge difference, but the former is common practice.

negative. More generally, you can make sure the fraction of examples from each class is as close as possible to the overall fraction in the original training set.

I would not bother with either of these techniques unless you have already tried plotting learning curves and concluded that the curves are too noisy to see the underlying trends. If your training set is large—say over 10,000 examples—and your class distribution is not very skewed, you probably won't need these techniques.

Finally, plotting a learning curve may be computationally expensive: For example, you might have to train ten models with 1,000, then 2,000, all the way up to 10,000 examples. Training models with small datasets is much faster than training models with large datasets. Thus, instead of evenly spacing out the training set sizes on a linear scale as above, you might train models with 1,000, 2,000, 4,000, 6,000, and 10,000 examples. This should still give you a clear sense of the trends in the learning curves. Of course, this technique is relevant only if the computational cost of training all the additional models is significant.

Comparing to human-level performance

33 Why we compare to human-level performance

Many machine learning systems aim to automate things that humans do well. Examples include image recognition, speech recognition, and email spam classification. Learning algorithms have also improved so much that we are now surpassing human-level performance on more and more of these tasks.

Further, there are several reasons building an ML system is easier if you are trying to do a task that people can do well:

1. Ease of obtaining data from human labelers. For example, since people recognize cat images well, it is straightforward for people to provide high accuracy labels for your learning algorithm.

2. Error analysis can draw on human intuition. Suppose a speech recognition algorithm is doing worse than human-level recognition. Say it incorrectly transcribes an audio clip as "This recipe calls for a *pear* of apples," mistaking "pair" for "pear." You can draw on human intuition and try to understand what information a person uses to get the correct transcription, and use this knowledge to modify the learning algorithm.

3. Use human-level performance to estimate the optimal error rate and also set a "desired error rate." Suppose your algorithm achieves 10% error on a task, but a person achieves 2% error. Then we know that the optimal error rate is 2% or lower and the avoidable bias is at least 8%. Thus, you should try bias-reducing techniques.

Even though item #3 might not sound important, I find that having a reasonable and achievable target error rate helps accelerate a team's progress. Knowing your algorithm has high avoidable bias is incredibly valuable and opens up a menu of options to try.

There are some tasks that even humans aren't good at. For example, picking a book to recommend to you; or picking an ad to show a user on a website; or predicting the stock market. Computers already surpass the performance of most people on these tasks. With these applications, we run into the following problems:

- **It is harder to obtain labels.** For example, it's hard for human labelers to annotate a database of users with the "optimal" book recommendation. If you operate a website or app that sells books, you can obtain data by showing books to users and seeing what they buy. If you do not operate such a site, you need to find more creative ways to get data.

- **Human intuition is harder to count on.** For example, pretty much no one can predict the stock market. So if our stock prediction algorithm does no better than random guessing, it is hard to figure out how to improve it.
- **It is hard to know what the optimal error rate and reasonable desired error rate is.** Suppose you already have a book recommendation system that is doing quite well. How do you know how much more it can improve without a human baseline?

34 How to define human-level performance

Suppose you are working on a medical imaging application that automatically makes diagnoses from x-ray images. A typical person with no previous medical background besides some basic training achieves 15% error on this task. A junior doctor achieves 10% error. An experienced doctor achieves 5% error. And a small team of doctors that discuss and debate each image achieves 2% error. Which one of these error rates defines "human-level performance"?

In this case, I would use 2% as the human-level performance proxy for our optimal error rate. You can also set 2% as the desired performance level because all three reasons from the previous chapter for comparing to human-level performance apply:

- **Ease of obtaining labeled data from human labelers.** You can get a team of doctors to provide labels to you with a 2% error rate.
- **Error analysis can draw on human intuition.** By discussing images with a team of doctors, you can draw on their intuitions.
- **Use human-level performance to estimate the optimal error rate and also set achievable "desired error rate."** It is reasonable to use 2% error as our estimate of the optimal error rate. The optimal error rate could be even lower than 2%, but it cannot be higher, since it is possible for a team of doctors to achieve 2% error. In contrast, it is not reasonable to use 5% or 10% as an estimate of the optimal error rate, since we know these estimates are necessarily too high.

When it comes to obtaining labeled data, you might not want to discuss every image with an entire team of doctors since their time is expensive. Perhaps you can have a single junior doctor label the vast majority of cases and bring only the harder cases to more experienced doctors or to the team of doctors.

If your system is currently at 40% error, then it doesn't matter much whether you use a junior doctor (10% error) or an experienced doctor (5% error) to label your data and provide intuitions. But if your system is already at 10% error, then defining the human-level reference as 2% gives you better tools to keep improving your system.

35 Surpassing human-level performance

You are working on speech recognition and have a dataset of audio clips. Suppose your dataset has many noisy audio clips so that even humans have 10% error. Suppose your system already achieves 8% error. Can you use any of the three techniques described in Chapter 33 to continue making rapid progress?

If you can identify a subset of data in which humans significantly surpass your system, then you can still use those techniques to drive rapid progress. For example, suppose your system is much better than people at recognizing speech in noisy audio, but humans are still better at transcribing very rapidly spoken speech.

For the subset of data with rapidly spoken speech:

1. You can still obtain transcripts from humans that are higher quality than your algorithm's output.
2. You can draw on human intuition to understand why they correctly heard a rapidly spoken utterance when your system didn't.
3. You can use human-level performance on rapidly spoken speech as a desired performance target.

More generally, so long as there are dev set examples where humans are right and your algorithm is wrong, then many of the techniques described earlier will apply. This is true even if, averaged over the entire dev/test set, your performance is already surpassing human-level performance.

There are many important machine learning applications where machines surpass human level performance. For example, machines are better at predicting movie ratings, how long it takes for a delivery car to drive somewhere, or whether to approve loan applications. Only a subset of techniques apply once humans have a hard time identifying examples that the algorithm is clearly getting wrong. Consequently, progress is usually slower on problems where machines already surpass human-level performance, while progress is faster when machines are still trying to catch up to humans.

Training and testing on different distributions

36 When you should train and test on different distributions

Users of your cat pictures app have uploaded 10,000 images, which you have manually labeled as containing cats or not. You also have a larger set of 200,000 images that you downloaded off the internet. How should you define train/dev/test sets?

Since the 10,000 user images closely reflect the actual probability distribution of data you want to do well on, you might use that for your dev and test sets. If you are training a data-hungry deep learning algorithm, you might give it the additional 200,000 internet images for training. Thus, your training and dev/test sets come from different probability distributions. How does this affect your work?

Instead of partitioning our data into train/dev/test sets, we could take all 210,000 images we have, and randomly shuffle them into train/dev/test sets. In this case, all the data comes from the same distribution. But I recommend against this method, because about 205,000/210,000 ≈ 97.6% of your dev/test data would come from internet images, which does not reflect the actual distribution you want to do well on. Remember our recommendation on choosing dev/test sets:

> Choose dev and test sets to reflect data you expect to get in the future and want to do well on.

Most of the academic literature on machine learning assumes that the training set, dev set and test set all come from the same distribution.[1] In the early days of machine learning, data was scarce. We usually only had one dataset drawn from some probability distribution. So we would randomly split that data into train/dev/test sets, and the assumption that all the data was coming from the same source was usually satisfied.

[1] There is some academic research on training and testing on different distributions. Examples include "domain adaptation," "transfer learning" and "multitask learning." But there is still a huge gap between theory and practice. If you train on dataset A and test on some very different type of data B, luck could have a huge effect on how well your algorithm performs. (Here, "luck" includes the researcher's hand-designed features for the particular task, as well as other factors that we just don't understand yet.) This makes the academic study of training and testing on different distributions difficult to carry out in a systematic way.

But in the era of big data, we now have access to huge training sets, such as cat internet images. Even if the training set comes from a different distribution than the dev/test set, we still want to use it for learning since it can provide a lot of information.

For the cat detector example, instead of putting all 10,000 user-uploaded images into the dev/test sets, we might instead put 5,000 into the dev/test sets. We can put the remaining 5,000 user-uploaded examples into the training set. This way, your training set of 205,000 examples contains some data that comes from your dev/test distribution along with the 200,000 internet images. We will discuss in a later chapter why this method is helpful.

Let's consider a second example. Suppose you are building a speech recognition system to transcribe street addresses for a voice-controlled mobile map/navigation app. You have 20,000 examples of users speaking street addresses. But you also have 500,000 examples of other audio clips with users speaking about other topics. You might take 10,000 examples of street addresses for the dev/test sets, and use the remaining 10,000, plus the additional 500,000 examples, for training.

We will continue to assume that your dev data and your test data come from the same distribution. But it is important to understand that different training and dev/test distributions offer some special challenges.

37 How to decide whether to use all your data

Suppose your cat detector's training set includes 10,000 user-uploaded images. This data comes from the same distribution as a separate dev/test set, and represents the distribution you care about doing well on. You also have an additional 20,000 images downloaded from the internet. Should you provide all 20,000+10,000=30,000 images to your learning algorithm as its training set, or discard the 20,000 internet images for fear of it biasing your learning algorithm?

When using earlier generations of learning algorithms (such as hand-designed computer vision features, followed by a simple linear classifier) there was a real risk that merging both types of data would cause you to perform worse. Thus, some engineers will warn you against including the 20,000 internet images.

But in the modern era of powerful, flexible learning algorithms—such as large neural networks—this risk has greatly diminished. If you can afford to build a neural network with a large enough number of hidden units/layers, you can safely add the 20,000 images to your training set. Adding the images is more likely to increase your performance.

This observation relies on the fact that there is some x —> y mapping that works well for both types of data. In other words, there exists some system that inputs either an internet image or a mobile app image and reliably predicts the label, even without knowing the source of the image.

Adding the additional 20,000 images has the following effects:

1. It gives your neural network more examples of what cats do/do not look like. This is helpful, since internet images and user-uploaded mobile app images do share some similarities. Your neural network can apply some of the knowledge acquired from internet images to mobile app images.

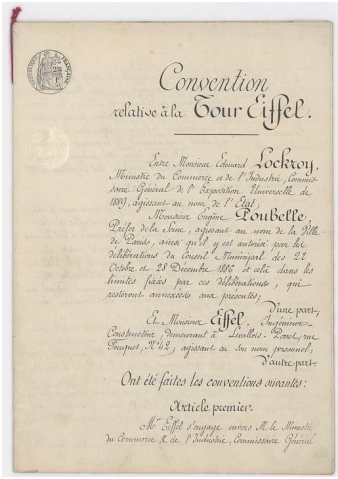
2. It forces the neural network to expend some of its capacity to learn about properties that are specific to internet images (such as higher resolution, different distributions of how the images are framed, etc.) If these properties differ greatly from mobile app images, it will "use up" some of the representational capacity of the neural network. Thus there is less capacity for recognizing data drawn from the distribution of mobile app images, which is what you really care about. Theoretically, this could hurt your algorithms' performance.

To describe the second effect in different terms, we can turn to the fictional character Sherlock Holmes, who says that your brain is like an attic; it only has a finite amount of space. He says that "for every addition of knowledge, you forget something that you knew before. It is of the highest importance, therefore, not to have useless facts elbowing out the useful ones."[2]

Fortunately, if you have the computational capacity needed to build a big enough neural network—i.e., a big enough attic—then this is not a serious concern. You have enough capacity to learn from both internet and from mobile app images, without the two types of data competing for capacity. Your algorithm's "brain" is big enough that you don't have to worry about running out of attic space.

But if you do not have a big enough neural network (or another highly flexible learning algorithm), then you should pay more attention to your training data matching your dev/test set distribution.

If you think you have data that has no benefit,you should just leave out that data for computational reasons. For example, suppose your dev/test sets contain mainly casual pictures of people, places, landmarks, animals. Suppose you also have a large collection of scanned historical documents:


Convention
relative à la Tour Eiffel.

These documents don't contain anything resembling a cat. They also look completely unlike your dev/test distribution. There is no point including this data as negative examples, because the benefit from the first effect above is negligible—there is almost nothing your neural network can learn from this data that it can apply to your dev/test set distribution. Including them would waste computation resources and representation capacity of the neural network.

[2] *A Study in Scarlet* by Arthur Conan Doyle

38 How to decide whether to include inconsistent data

Suppose you want to learn to predict housing prices in New York City. Given the size of a house (input feature x), you want to predict the price (target label y).

Housing prices in New York City are very high. Suppose you have a second dataset of housing prices in Detroit, Michigan, where housing prices are much lower. Should you include this data in your training set?

Given the same size x, the price of a house y is very different depending on whether it is in New York City or in Detroit. If you only care about predicting New York City housing prices, putting the two datasets together will hurt your performance. In this case, it would be better to leave out the inconsistent Detroit data.[3]

How is this New York City vs. Detroit example different from the mobile app vs. internet cat images example?

The cat image example is different because, given an input picture x, one can reliably predict the label y indicating whether there is a cat, even without knowing if the image is an internet image or a mobile app image. I.e., there is a function f(x) that reliably maps from the input x to the target output y, even without knowing the origin of x. Thus, the task of recognition from internet images is "consistent" with the task of recognition from mobile app images. This means there was little downside (other than computational cost) to including all the data, and some possible significant upside. In contrast, New York City and Detroit, Michigan data are not consistent. Given the same x (size of house), the price is very different depending on where the house is.

[3] There is one way to address the problem of Detroit data being inconsistent with New York City data, which is to add an extra feature to each training example indicating the city. Given an input x—which now specifies the city—the target value of y is now unambiguous. However, in practice I do not see this done frequently.

39 Weighting data

Suppose you have 200,000 images from the internet and 5,000 images from your mobile app users. There is a 40:1 ratio between the size of these datasets. In theory, so long as you build a huge neural network and train it long enough on all 205,000 images, there is no harm in trying to make the algorithm do well on both internet images and mobile images.

But in practice, having 40x as many internet images as mobile app images might mean you need to spend 40x (or more) as much computational resources to model both, compared to if you trained on only the 5,000 images.

If you don't have huge computational resources, you could give the internet images a much lower weight as a compromise.

For example, suppose your optimization objective is squared error (This is not a good choice for a classification task, but it will simplify our explanation.) Thus, our learning algorithm tries to optimize:

$$\min_{\theta} \sum_{(x,y)\in \text{MobileImg}} \left(h_\theta\left(x\right) - y\right)^2 + \sum_{(x,y)\in \text{InternetImg}} \left(h_\theta\left(x\right) - y\right)^2$$

The first sum above is over the 5,000 mobile images, and the second sum is over the 200,000 internet images. You can instead optimize with an additional parameter β:

$$\min_{\theta} \sum_{(x,y)\in \text{MobileImg}} \left(h_\theta\left(x\right) - y\right)^2 + \beta \sum_{(x,y)\in \text{InternetImg}} \left(h_\theta\left(x\right) - y\right)^2$$

If you set β=1/40, the algorithm would give equal weight to the 5,000 mobile images and the 200,000 internet images. You can also set the parameter β to other values, perhaps by tuning to the dev set.

By weighting the additional Internet images less, you don't have to build as massive a neural network to make sure the algorithm does well on both types of tasks. This type of re-weighting is needed only when you suspect the additional data (Internet Images) has a very different distribution than the dev/test set, or if the additional data is much larger than the data that came from the same distribution as the dev/test set (mobile images).

40 Generalizing from the training set to the dev set

Suppose you are applying ML in a setting where the training and the dev/test distributions are different. Say, the training set contains Internet images + Mobile images, and the dev/test sets contain only Mobile images. However, the algorithm is not working well: It has a much higher dev/test set error than you would like. Here are some possibilities of what might be wrong:

1. It does not do well on the training set. This is the problem of high (avoidable) bias on the training set distribution.
2. It does well on the training set, but does not generalize well to previously unseen data *drawn from the same distribution as the training set*. This is high variance.
3. It generalizes well to new data drawn from the same distribution as the training set, but not to data drawn from the dev/test set distribution. We call this problem **data mismatch**, since it is because the training set data is a poor match for the dev/test set data.

For example, suppose that humans achieve near perfect performance on the cat recognition task. Your algorithm achieves this:

- 1% error on the training set
- 1.5% error on data drawn from the same distribution as the training set that the algorithm has not seen
- 10% error on the dev set

In this case, you clearly have a data mismatch problem. To address this, you might try to make the training data more similar to the dev/test data. We discuss some techniques for this later.

In order to diagnose to what extent an algorithm suffers from each of the problems 1-3 above, it will be useful to have another dataset. Specifically, rather than giving the algorithm all the available training data, you can split it into two subsets: The actual training set which the algorithm will train on, and a separate set, which we will call the “Training dev” set, that we will not train on.

You now have four subsets of data:

- Training set. This is the data that the algorithm will learn from (e.g., Internet images + Mobile images). This does not have to be drawn from the same distribution as what we really care about (the dev/test set distribution).
- Training dev set: This data is drawn from the same distribution as the training set (e.g., Internet images + Mobile images). This is usually smaller than the training set; it only needs to be large enough to evaluate and track the progress of our learning algorithm.
- Dev set: This is drawn from the same distribution as the test set, and it reflects the distribution of data that we ultimately care about doing well on. (E.g., mobile images.)
- Test set: This is drawn from the same distribution as the dev set. (E.g., mobile images.)

Armed with these four separate datasets, you can now evaluate:

- Training error, by evaluating on the training set.
- The algorithm's ability to generalize to new data drawn from the training set distribution, by evaluating on the training dev set.
- The algorithm's performance on the task you care about, by evaluating on the dev and/or test sets.

Most of the guidelines in Chapters 5-7 for picking the size of the dev set also apply to the training dev set.

41 Identifying Bias, Variance, and Data Mismatch Errors

Suppose humans achieve almost perfect performance (≈0% error) on the cat detection task, and thus the optimal error rate is about 0%. Suppose you have:

- 1% error on the training set.
- 5% error on training dev set.
- 5% error on the dev set.

What does this tell you? Here, you know that you have high variance. The variance reduction techniques described earlier should allow you to make progress.

Now, suppose your algorithm achieves:

- 10% error on the training set.
- 11% error on training dev set.
- 12% error on the dev set.

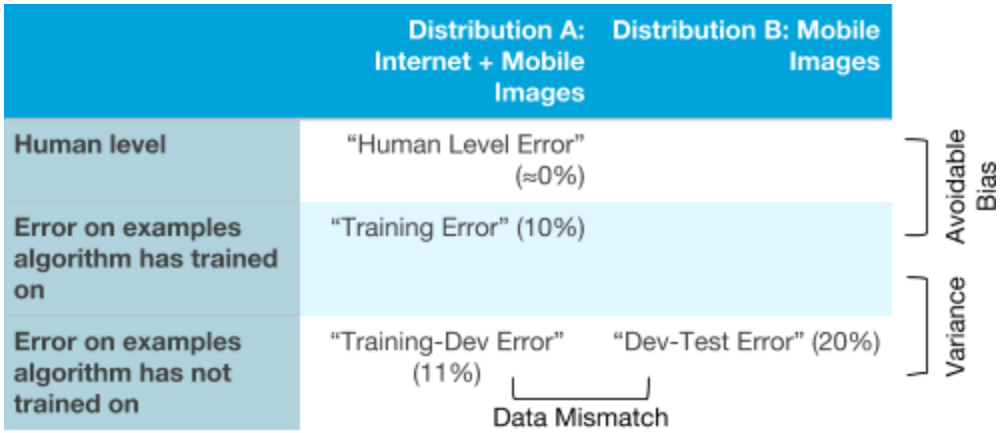
This tells you that you have high avoidable bias on the training set. I.e., the algorithm is doing poorly on the training set. Bias reduction techniques should help.

In the two examples above, the algorithm suffered from only high avoidable bias or high variance. It is possible for an algorithm to suffer from any subset of high avoidable bias, high variance, and data mismatch. For example:

- 10% error on the training set.
- 11% error on training dev set.
- 20% error on the dev set.

This algorithm suffers from high avoidable bias and from data mismatch. It does not, however, suffer from high variance on the training set distribution.

It might be easier to understand how the different types of errors relate to each other by drawing them as entries in a table:



Continuing with the example of the cat image detector, you can see that there are two different distributions of data on the x-axis. On the y-axis, we have three types of error: human level error, error on examples the algorithm has trained on, and error on examples the algorithm has not trained on. We can fill in the boxes with the different types of errors we identified in the previous chapter.

If you wish, you can also fill in the remaining two boxes in this table: You can fill in the upper-right box (Human level performance on Mobile Images) by asking some humans to label your mobile cat images data and measure their error. You can fill in the next box by taking the mobile cat images (Distribution B) and putting a small fraction of into the training set so that the neural network learns on it too. Then you measure the learned model's error on that subset of data. Filling in these two additional entries may sometimes give additional insight about what the algorithm is doing on the two different distributions (Distribution A and B) of data.

By understanding which types of error the algorithm suffers from the most, you will be better positioned to decide whether to focus on reducing bias, reducing variance, or reducing data mismatch.

42 Addressing data mismatch

Suppose you have developed a speech recognition system that does very well on the training set and on the training dev set. However, it does poorly on your dev set: You have a data mismatch problem. What can you do?

I recommend that you: (i) Try to understand what properties of the data differ between the training and the dev set distributions. (ii) Try to find more training data that better matches the dev set examples that your algorithm has trouble with.[1]

For example, suppose you carry out an error analysis on the speech recognition dev set: You manually go through 100 examples, and try to understand where the algorithm is making mistakes. You find that your system does poorly because most of the audio clips in the dev set are taken within a car, whereas most of the training examples were recorded against a quiet background. The engine and road noise dramatically worsen the performance of your speech system. In this case, you might try to acquire more training data comprising audio clips that were taken in a car. The purpose of the error analysis is to understand the significant differences between the training and the dev set, which is what leads to the data mismatch.

If your training and training dev sets include audio recorded within a car, you should also double-check your system's performance on this subset of data. If it is doing well on the car data in the training set but not on car data in the training dev set, then this further validates the hypothesis that getting more car data would help. This is why we discussed the possibility of including in your training set some data drawn from the same distribution as your dev/test set in the previous chapter. Doing so allows you to compare your performance on the car data in the training set vs. the dev/test set.

Unfortunately, there are no guarantees in this process. For example, if you don't have any way to get more training data that better match the dev set data, you might not have a clear path towards improving performance.

[1]There is also some research on "domain adaptation"—how to train an algorithm on one distribution and have it generalize to a different distribution. These methods are typically applicable only in special types of problems and are much less widely used than the ideas described in this chapter.

43 Artificial data synthesis

Your speech system needs more data that sounds as if it were taken from within a car. Rather than collecting a lot of data while driving around, there might be an easier way to get this data: By artificially synthesizing it.

Suppose you obtain a large quantity of car/road noise audio clips. You can download this data from several websites. Suppose you also have a large training set of people speaking in a quiet room. If you take an audio clip of a person speaking and "add" to that to an audio clip of car/road noise, you will obtain an audio clip that sounds as if that person was speaking in a noisy car. Using this process, you can "synthesize" huge amounts of data that sound as if it were collected inside a car.

More generally, there are several circumstances where artificial data synthesis allows you to create a huge dataset that reasonably matches the dev set. Let's use the cat image detector as a second example. You notice that dev set images have much more motion blur because they tend to come from cellphone users who are moving their phone slightly while taking the picture. You can take non-blurry images from the training set of internet images, and add simulated motion blur to them, thus making them more similar to the dev set.

Keep in mind that artificial data synthesis has its challenges: it is sometimes easier to create synthetic data that appears realistic to a person than it is to create data that appears realistic to a computer. For example, suppose you have 1,000 hours of speech training data, but only 1 hour of car noise. If you repeatedly use the same 1 hour of car noise with different portions from the original 1,000 hours of training data, you will end up with a synthetic dataset where the same car noise is repeated over and over. While a person listening to this audio probably would not be able to tell—all car noise sounds the same to most of us—it is possible that a learning algorithm would "overfit" to the 1 hour of car noise. Thus, it could generalize poorly to a new audio clip where the car noise happens to sound different.

Alternatively, suppose you have 1,000 unique hours of car noise, but all of it was taken from just 10 different cars. In this case, it is possible for an algorithm to "overfit" to these 10 cars and perform poorly if tested on audio from a different car. Unfortunately, these problems can be hard to spot.

To take one more example, suppose you are building a computer vision system to recognize cars. Suppose you partner with a video gaming company, which has computer graphics models of several cars. To train your algorithm, you use the models to generate synthetic images of cars. Even if the synthesized images look very realistic, this approach (which has been independently proposed by many people) will probably not work well. The video game might have ~20 car designs in the entire video game. It is very expensive to build a 3D car model of a car; if you were playing the game, you probably wouldn't notice that you're seeing the same cars over and over, perhaps only painted differently. I.e., this data looks very realistic to you. But compared to the set of all cars out on roads—and therefore what you're likely to see in the dev/test sets—this set of 20 synthesized cars captures only a minuscule fraction of the world's distribution of cars. Thus if your 100,000 training examples all come from these 20 cars, your system will "overfit" to these 20 specific car designs, and it will fail to generalize well to dev/test sets that include other car designs.

When synthesizing data, put some thought into whether you're really synthesizing a representative set of examples. Try to avoid giving the synthesized data properties that makes it possible for a learning algorithm to distinguish synthesized from non-synthesized examples—such as if all the synthesized data comes from one of 20 car designs, or all the synthesized audio comes from only 1 hour of car noise. This advice can be hard to follow.

When working on data synthesis, my teams have sometimes taken weeks before we produced data with details that are close enough to the actual distribution for the synthesized data to have a significant effect. But if you are able to get the details right, you can suddenly access a far larger training set than before.

Debugging inference algorithms

44 The Optimization Verification test

Suppose you are building a speech recognition system. Your system works by inputting an audio clip A, and computing some $\text{Score}_A(S)$ for each possible output sentence S. For example, you might try to estimate $\text{Score}_A(S) = P(S|A)$, the probability that the correct output transcription is the sentence S, given that the input audio was A.

Given a way to compute $\text{Score}_A(S)$, you still have to find the English sentence S that maximizes it:

$$\text{Output} = \arg \max_S \text{Score}_A(S)$$

How do you compute the "arg max" above? If the English language has 50,000 words, then there are $(50{,}000)^N$ possible sentences of length N—far too many to exhaustively enumerate. So, you need to apply an approximate search algorithm, to try to find the value of S that optimizes (maximizes) $\text{Score}_A(S)$. One example search algorithm is "beam search," which keeps only K top candidates during the search process. (For the purposes of this chapter, you don't need to understand the details of beam search.) Algorithms like this are not guaranteed to find the value of S that maximizes $\text{Score}_A(S)$.

Suppose that an audio clip A records someone saying "I love machine learning." But instead of outputting the correct transcription, your system outputs the incorrect "I love robots." There are now two possibilities for what went wrong:

1. **Search algorithm problem**. The approximate search algorithm (beam search) failed to find the value of S that maximizes $\text{Score}_A(S)$.
2. **Objective (scoring function) problem.** Our estimates for $\text{Score}_A(S) = P(S|A)$ were inaccurate. In particular, our choice of $\text{Score}_A(S)$ failed to recognize that "I love machine learning" is the correct transcription.

Depending on which of these was the cause of the failure, you should prioritize your efforts very differently. If #1 was the problem, you should work on improving the search algorithm. If #2 was the problem, you should work on the learning algorithm that estimates $\text{Score}_A(S)$.

Facing this situation, some researchers will randomly decide to work on the search algorithm; others will randomly work on a better way to learn values for $\text{Score}_A(S)$. But unless you know which of these is the underlying cause of the error, your efforts could be wasted. How can you decide more systematically what to work on?

Let S_{out} be the output transcription ("I love robots"). Let S* be the correct transcription ("I love machine learning"). In order to understand whether #1 or #2 above is the problem, you can perform the **Optimization Verification test**: First, compute $Score_A(S^*)$ and $Score_A(S_{out})$. Then check whether $Score_A(S^*) > Score_A(S_{out})$. There are two possibilities:

Case 1: $Score_A(S^*) > Score_A(S_{out})$

In this case, your learning algorithm has correctly given S* a higher score than S_{out}. Nevertheless, our approximate search algorithm chose S_{out} rather than S*. This tells you that your approximate search algorithm is failing to choose the value of S that maximizes $Score_A(S)$. In this case, the Optimization Verification test tells you that you have a search algorithm problem and should focus on that. For example, you could try increasing the beam width of beam search.

Case 2: $Score_A(S^*) \leq Score_A(S_{out})$

In this case, you know that the way you're computing $Score_A(.)$ is at fault: It is failing to give a strictly higher score to the correct output S^* than the incorrect S_{out}. The Optimization Verification test tells you that you have an objective (scoring) function problem. Thus, you should focus on improving how you learn or approximate $Score_A(S)$ for different sentences S.

Our discussion has focused on a single example. To apply the Optimization Verification test in practice, you should examine the errors in your dev set. For each error, you would test whether $Score_A(S^*) > Score_A(S_{out})$. Each dev example for which this inequality holds will get marked as an error caused by the optimization algorithm. Each example for which this does not hold ($Score_A(S^*) \leq Score_A(S_{out})$) gets counted as a mistake due to the way you're computing $Score_A(.)$.

For example, suppose you find that 95% of the errors were due to the scoring function $Score_A(.)$, and only 5% due to the optimization algorithm. Now you know that no matter how much you improve your optimization procedure, you would realistically eliminate only ~5% of our errors. Thus, you should instead focus on improving how you estimate $Score_A(.)$.

45 General form of Optimization Verification test

You can apply the Optimization Verification test when, given some input x, you know how to compute $\text{Score}_x(y)$ that indicates how good a response y is to an input x. Furthermore, you are using an approximate algorithm to try to find $\arg\max_y \text{Score}_x(y)$, but suspect that the search algorithm is sometimes failing to find the maximum. In our previous speech recognition example, x=*A* was an audio clip, and y=*S* was the output transcript.

Suppose y* is the "correct" output but the algorithm instead outputs y_{out}. Then the key test is to measure whether $\text{Score}_x(y^*) > \text{Score}_x(y_{out})$. If this inequality holds, then we blame the optimization algorithm for the mistake. Refer to the previous chapter to make sure you understand the logic behind this. Otherwise, we blame the computation of $\text{Score}_x(y)$.

Let's look at one more example. Suppose you are building a Chinese-to-English machine translation system. Your system works by inputting a Chinese sentence *C*, and computing some $\text{Score}_C(E)$ for each possible translation *E*. For example, you might use $\text{Score}_C(E) = P(E|C)$, the probability of the translation being E given that the input sentence was *C*.

Your algorithm translates sentences by trying to compute:

$$\text{Output} = \arg\max_E \text{Score}_C(E)$$

However, the set of all possible English sentences *E* is too large, so you rely on a heuristic search algorithm.

Suppose your algorithm outputs an incorrect translation E_{out} rather than some correct translation *E**. Then the Optimization Verification test would ask you to compute whether $\text{Score}_C(E^*) > \text{Score}_C(E_{out})$. If this inequality holds, then the $\text{Score}_C(.)$ correctly recognized E* as a superior output to E_{out}; thus, you would attribute this error to the approximate search algorithm. Otherwise, you attribute this error to the computation of $\text{Score}_C(.)$.

It is a very common "design pattern" in AI to first learn an approximate scoring function $\text{Score}_x(.)$, then use an approximate maximization algorithm. If you are able to spot this pattern, you will be able to use the Optimization Verification test to understand your source of errors.

46 Reinforcement learning example

Suppose you are using machine learning to teach a helicopter to fly complex maneuvers. Here is a time-lapse photo of a computer-controller helicopter executing a landing with the engine turned off.

This is called an “autorotation” maneuver. It allows helicopters to land even if their engine unexpectedly fails. Human pilots practice this maneuver as part of their training. Your goal is to use a learning algorithm to fly the helicopter through a trajectory *T* that ends in a safe landing.

To apply reinforcement learning, you have to develop a “Reward function” *R*(.) that gives a score measuring how good each possible trajectory *T* is. For example, if *T* results in the helicopter crashing, then perhaps the reward is *R(T)* = -1,000—a huge negative reward. A trajectory *T* resulting in a safe landing might result in a positive *R(T)* with the exact value depending on how smooth the landing was. The reward function *R*(.) is typically chosen by hand to quantify how desirable different trajectories *T* are. It has to trade off how bumpy the landing was, whether the helicopter landed in exactly the desired spot, how rough the ride down was for passengers, and so on. It is not easy to design good reward functions.

Given a reward function *R(T),* the job of the reinforcement learning algorithm is to control the helicopter so that it achieves $\max_T R(T)$. However, reinforcement learning algorithms make many approximations and may not succeed in achieving this maximization.

Suppose you have picked some reward *R(.)* and have run your learning algorithm. However, its performance appears far worse than your human pilot—the landings are bumpier and seem less safe than what a human pilot achieves. How can you tell if the fault is with the reinforcement learning algorithm—which is trying to carry out a trajectory that achieves $\max_T R(T)$—or if the fault is with the reward function—which is trying to measure as well as specify the ideal tradeoff between ride bumpiness and accuracy of landing spot?

To apply the Optimization Verification test, let T_{human} be the trajectory achieved by the human pilot, and let T_{out} be the trajectory achieved by the algorithm. According to our description above, T_{human} is a superior trajectory to T_{out}. Thus, the key test is the following: Does it hold true that $R(T_{\text{human}}) > R(T_{\text{out}})$?

Case 1: If this inequality holds, then the reward function $R(.)$ is correctly rating T_{human} as superior to T_{out}. But our reinforcement learning algorithm is finding the inferior $T_{\text{out.}}$ This suggests that working on improving our reinforcement learning algorithm is worthwhile.

Case 2: The inequality does not hold: $R(T_{\text{human}}) \leq R(T_{\text{out}})$. This means $R(.)$ assigns a worse score to T_{human} even though it is the superior trajectory. You should work on improving $R(.)$ to better capture the tradeoffs that correspond to a good landing.

Many machine learning applications have this "pattern" of optimizing an approximate scoring function $\text{Score}_x(.)$ using an approximate search algorithm. Sometimes, there is no specified input x, so this reduces to just Score(.). In our example above, the scoring function was the reward function $\text{Score}(T)=\text{R}(T)$, and the optimization algorithm was the reinforcement learning algorithm trying to execute a good trajectory T.

One difference between this and earlier examples is that, rather than comparing to an "optimal" output, you were instead comparing to human-level performance T_{human}.We assumed T_{human} is pretty good, even if not optimal. In general, so long as you have some y* (in this example, T_{human}) that is a superior output to the performance of your current learning algorithm—even if it is not the "optimal" output—then the Optimization Verification test can indicate whether it is more promising to improve the optimization algorithm or the scoring function.

End-to-end deep learning

47 The rise of end-to-end learning

Suppose you want to build a system to examine online product reviews and automatically tell you if the writer liked or disliked that product. For example, you hope to recognize the following review as highly positive:

> This is a great mop!

and the following as highly negative:

> This mop is low quality--I regret buying it.

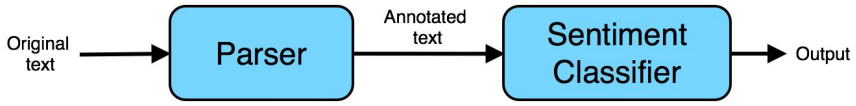
The problem of recognizing positive vs. negative opinions is called "sentiment classification." To build this system, you might build a "pipeline" of two components:

1. Parser: A system that annotates the text with information identifying the most important words.[1] For example, you might use the parser to label all the adjectives and nouns. You would therefore get the following annotated text:

   This is a great$_{\text{Adjective}}$ mop$_{\text{Noun}}$!

2. Sentiment classifier: A learning algorithm that takes as input the annotated text and predicts the overall sentiment. The parser's annotation could help this learning algorithm greatly: By giving adjectives a higher weight, your algorithm will be able to quickly hone in on the important words such as "great," and ignore less important words such as "this."

We can visualize your "pipeline" of two components as follows:



There has been a recent trend toward replacing pipeline systems with a single learning algorithm. An **end-to-end learning algorithm** for this task would simply take as input the raw, original text "This is a great mop!", and try to directly recognize the sentiment:

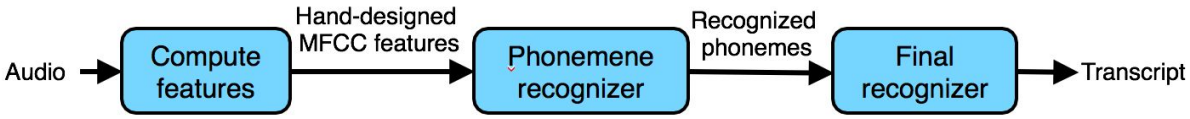
[1] A parser gives a much richer annotation of the text than this, but this simplified description will suffice for explaining end-to-end deep learning.

Neural networks are commonly used in end-to-end learning systems. The term “end-to-end” refers to the fact that we are asking the learning algorithm to go directly from the input to the desired output. I.e., the learning algorithm directly connects the “input end” of the system to the “output end.”

In problems where data is abundant, end-to-end systems have been remarkably successful. But they are not always a good choice. The next few chapters will give more examples of end-to-end systems as well as give advice on when you should and should not use them.

48 More end-to-end learning examples

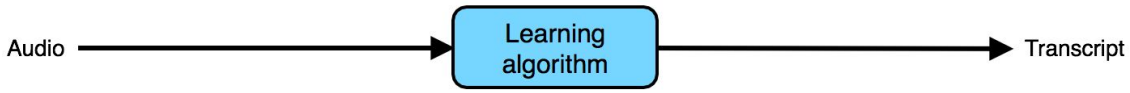
Suppose you want to build a speech recognition system. You might build a system with three components:



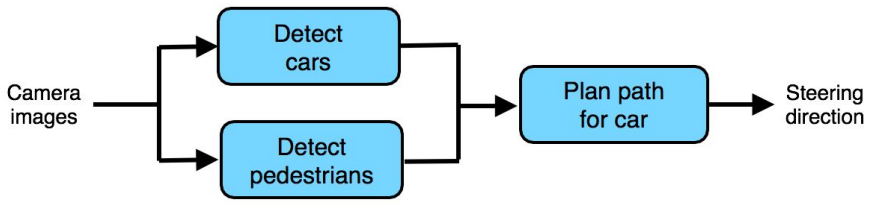
The components work as follows:

1. Compute features: Extract hand-designed features, such as MFCC (Mel-frequency cepstrum coefficients) features, which try to capture the content of an utterance while disregarding less relevant properties, such as the speaker's pitch.
2. Phoneme recognizer: Some linguists believe that there are basic units of sound called "phonemes." For example, the initial "k" sound in "keep" is the same phoneme as the "c" sound in "cake." This system tries to recognize the phonemes in the audio clip.
3. Final recognizer: Take the sequence of recognized phonemes, and try to string them together into an output transcript.

In contrast, an end-to-end system might input an audio clip, and try to directly output the transcript:



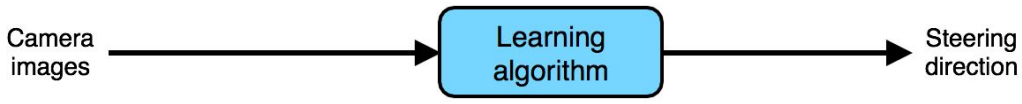
So far, we have only described machine learning "pipelines" that are completely linear: the output is sequentially passed from one staged to the next. Pipelines can be more complex. For example, here is a simple architecture for an autonomous car:



It has three components: One detects other cars using the camera images; one detects pedestrians; then a final component plans a path for our own car that avoids the cars and pedestrians.

Not every component in a pipeline has to be learned. For example, the literature on "robot motion planning" has numerous algorithms for the final path planning step for the car. Many of these algorithms do not involve learning.

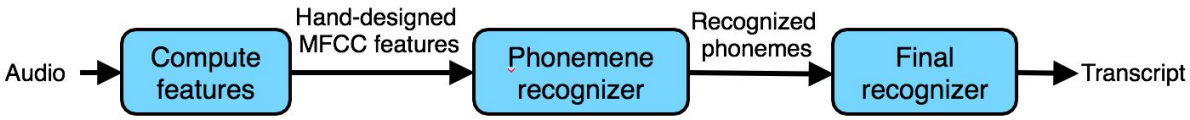
In contrast, and end-to-end approach might try to take in the sensor inputs and directly output the steering direction:



Even though end-to-end learning has seen many successes, it is not always the best approach. For example, end-to-end speech recognition works well. But I'm skeptical about end-to-end learning for autonomous driving. The next few chapters explain why.

49 Pros and cons of end-to-end learning

Consider the same speech pipeline from our earlier example:



Many parts of this pipeline were "hand-engineered":

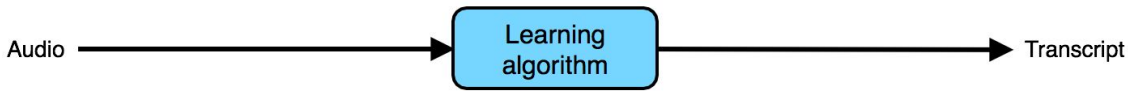
- MFCCs are a set of hand-designed audio features. Although they provide a reasonable summary of the audio input, they also simplify the input signal by throwing some information away.
- Phonemes are an invention of linguists. They are an imperfect representation of speech sounds. To the extent that phonemes are a poor approximation of reality, forcing an algorithm to use a phoneme representation will limit the speech system's performance.

These hand-engineered components limit the potential performance of the speech system. However, allowing hand-engineered components also has some advantages:

- The MFCC features are robust to some properties of speech that do not affect the content, such as speaker pitch. Thus, they help simplify the problem for the learning algorithm.
- To the extent that phonemes are a reasonable representation of speech, they can also help the learning algorithm understand basic sound components and therefore improve its performance.

Having more hand-engineered components generally allows a speech system to learn with less data. The hand-engineered knowledge captured by MFCCs and phonemes "supplements" the knowledge our algorithm acquires from data. When we don't have much data, this knowledge is useful.

Now, consider the end-to-end system:



This system lacks the hand-engineered knowledge. Thus, when the training set is small, it might do worse than the hand-engineered pipeline.

However, when the training set is large, then it is not hampered by the limitations of an MFCC or phoneme-based representation. If the learning algorithm is a large-enough neural network and if it is trained with enough training data, it has the potential to do very well, and perhaps even approach the optimal error rate.

End-to-end learning systems tend to do well when there is a lot of labeled data for "both ends"—the input end and the output end. In this example, we require a large dataset of (audio, transcript) pairs. When this type of data is not available, approach end-to-end learning with great caution.

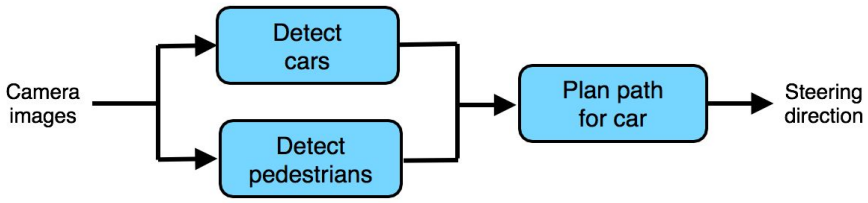
If you are working on a machine learning problem where the training set is very small, most of your algorithm's knowledge will have to come from your human insight. I.e., from your "hand engineering" components.

If you choose not to use an end-to-end system, you will have to decide what are the steps in your pipeline, and how they should plug together. In the next few chapters, we'll give some suggestions for designing such pipelines.

50 Choosing pipeline components: Data availability

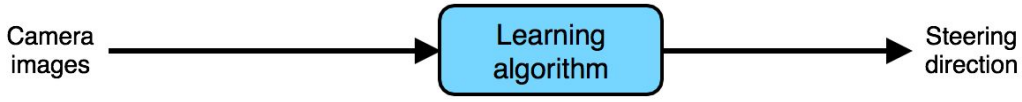
When building a non-end-to-end pipeline system, what are good candidates for the components of the pipeline? How you design the pipeline will greatly impact the overall system's performance. One important factor is whether you can easily collect data to train each of the components.

For example, consider this autonomous driving architecture:



You can use machine learning to detect cars and pedestrians. Further, it is not hard to obtain data for these: There are numerous computer vision datasets with large numbers of labeled cars and pedestrians. You can also use crowdsourcing (such as Amazon Mechanical Turk) to obtain even larger datasets. It is thus relatively easy to obtain training data to build a car detector and a pedestrian detector.

In contrast, consider a pure end-to-end approach:



To train this system, we would need a large dataset of (Image, Steering Direction) pairs. It is very time-consuming and expensive to have people drive cars around and record their steering direction to collect such data. You need a fleet of specially-instrumented cars, and a huge amount of driving to cover a wide range of possible scenarios. This makes an end-to-end system difficult to train. It is much easier to obtain a large dataset of labeled car or pedestrian images.

More generally, if there is a lot of data available for training "intermediate modules" of a pipeline (such as a car detector or a pedestrian detector), then you might consider using a

pipeline with multiple stages. This structure could be superior because you could use all that available data to train the intermediate modules.

Until more end-to-end data becomes available, I believe the non-end-to-end approach is significantly more promising for autonomous driving: Its architecture better matches the availability of data.

51 Choosing pipeline components: Task simplicity

Other than data availability, you should also consider a second factor when picking components of a pipeline: How simple are the tasks solved by the individual components? You should try to choose pipeline components that are individually easy to build or learn. But what does it mean for a component to be "easy" to learn?

Consider these machine learning tasks, listed in order of increasing difficulty:

1. Classifying whether an image is overexposed (like the example above)
2. Classifying whether an image was taken indoor or outdoor
3. Classifying whether an image contains a cat
4. Classifying whether an image contains a cat with both black and white fur
5. Classifying whether an image contains a Siamese cat (a particular breed of cat)

Each of these is a binary image classification task: You have to input an image, and output either 0 or 1. But the tasks earlier in the list seem much "easier" for a neural network to learn. You will be able to learn the easier tasks with fewer training examples.

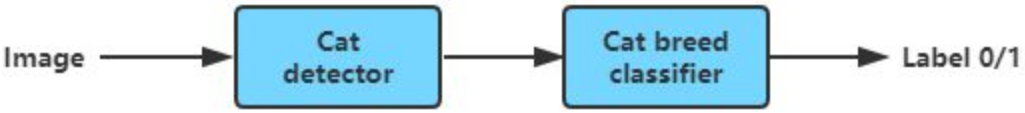
Machine learning does not yet have a good formal definition of what makes a task easy or hard.[1] With the rise of deep learning and multi-layered neural networks, we sometimes say a task is "easy" if it can be carried out with fewer computation steps (corresponding to a shallow neural network), and "hard" if it requires more computation steps (requiring a deeper neural network). But these are informal definitions.

[1]Information theory has the concept of "Kolmogorov Complexity", which says that the complexity of a learned function is the length of the shortest computer program that can produce that function. However, this theoretical concept has found few practical applications in AI. See also: https://en.wikipedia.org/wiki/Kolmogorov_complexity

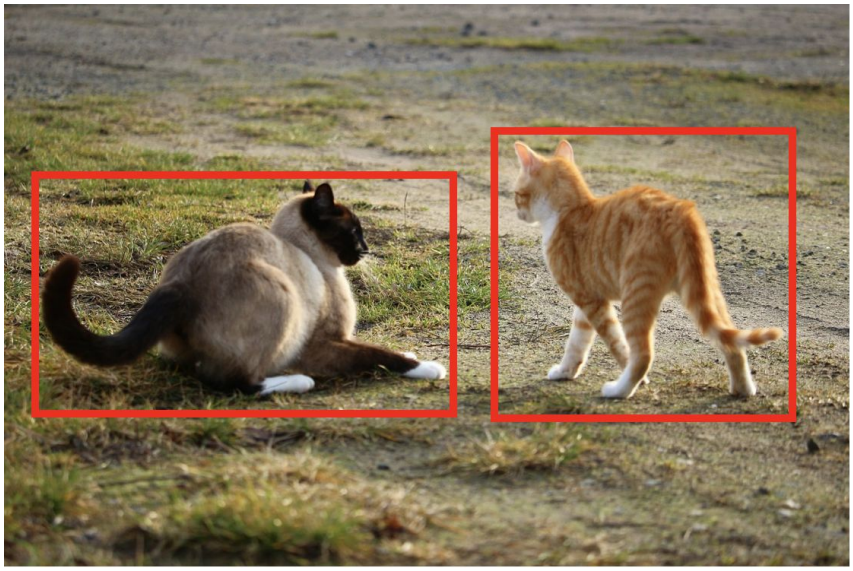
If you are able to take a complex task, and break it down into simpler sub-tasks, then by coding in the steps of the sub-tasks explicitly, you are giving the algorithm prior knowledge that can help it learn a task more efficiently.

Suppose you are building a Siamese cat detector. This is the pure end-to-end architecture:

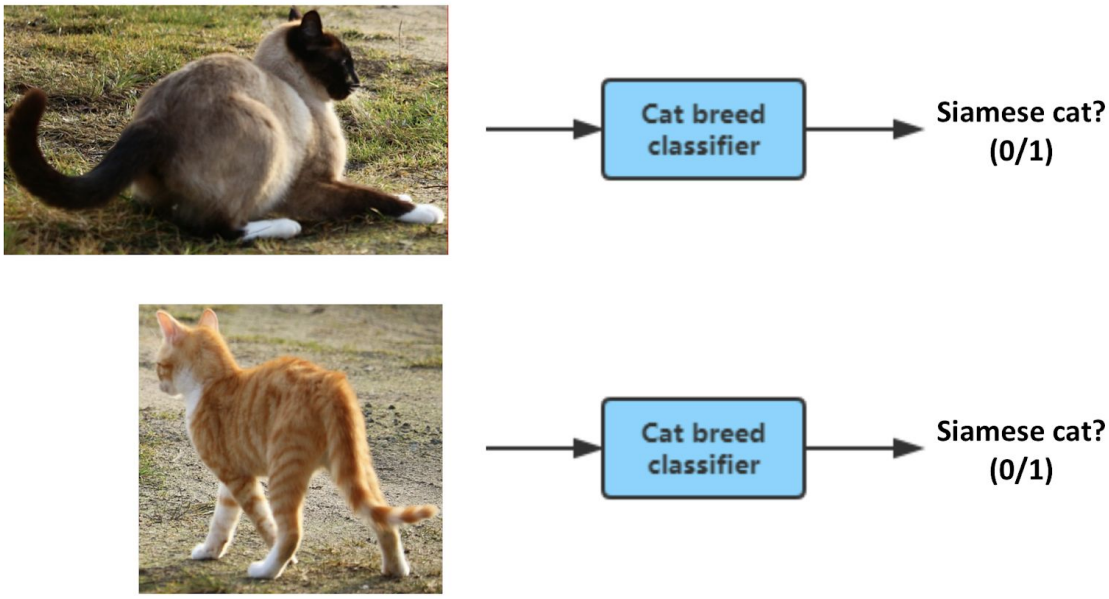
In contrast, you can alternatively use a pipeline with two steps:



The first step (cat detector) detects all the cats in the image.



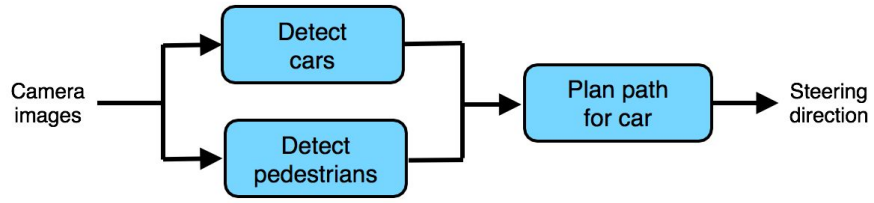
The second step then passes cropped images of each of the detected cats (one at a time) to a cat species classifier, and finally outputs 1 if any of the cats detected is a Siamese cat.



Compared to training a purely end-to-end classifier using just labels 0/1, each of the two components in the pipeline--the cat detector and the cat breed classifier--seem much easier to learn and will require significantly less data.[2]

[2] If you are familiar with practical object detection algorithms, you will recognize that they do not learn just with 0/1 image labels, but are instead trained with bounding boxes provided as part of the training data. A discussion of them is beyond the scope of this chapter. See the Deep Learning specialization on Coursera (http://deeplearning.ai) if you would like to learn more about such algorithms.

As one final example, let's revisit the autonomous driving pipeline.



By using this pipeline, you are telling the algorithm that there are 3 key steps to driving: (1) Detect other cars, (2) Detect pedestrians, and (3) Plan a path for your car. Further, each of these is a relatively simpler function--and can thus be learned with less data--than the purely end-to-end approach.

In summary, when deciding what should be the components of a pipeline, try to build a pipeline where each component is a relatively "simple" function that can therefore be learned from only a modest amount of data.

52 Directly learning rich outputs

An image classification algorithm will input an image x, and output an integer indicating the object category. Can an algorithm instead output an entire sentence describing the image?

For example:

x = 

y = "A yellow bus driving down a road with green trees and green grass in the background."

Traditional applications of supervised learning learned a function $h:X \rightarrow Y$, where the output y was usually an integer or a real number. For example:

Problem	X	Y
Spam classification	Email	Spam/Not spam (0/1)
Image recognition	Image	Integer label
Housing price prediction	Features of house	Price in dollars
Product recommendation	Product & user features	Chance of purchase

One of the most exciting developments in end-to-end deep learning is that it is letting us directly learn y that are much more complex than a number. In the image-captioning example above, you can have a neural network input an image (x) and directly output a caption (y).

Here are more examples:

Problem	X	Y	Example Citation
Image captioning	Image	Text	Mao et al., 2014
Machine translation	English text	French text	Suskever et al., 2014
Question answering	(Text,Question) pair	Answer text	Bordes et al., 2015
Speech recognition	Audio	Transcription	Hannun et al., 2015
TTS	Text features	Audio	van der Oord et al., 2016

This is an accelerating trend in deep learning: When you have the right (input,output) labeled pairs, you can sometimes learn end-to-end even when the output is a sentence, an image, audio, or other outputs that are richer than a single number.