Draft Version

MACHINE LEARNING YEARNING

Technical Strategy for AI Engineers, In the Era of Deep Learning

ANDREW NG

deeplearning.ai

Machine Learning Yearning is a
deeplearning.ai project.

Table of Contents

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!

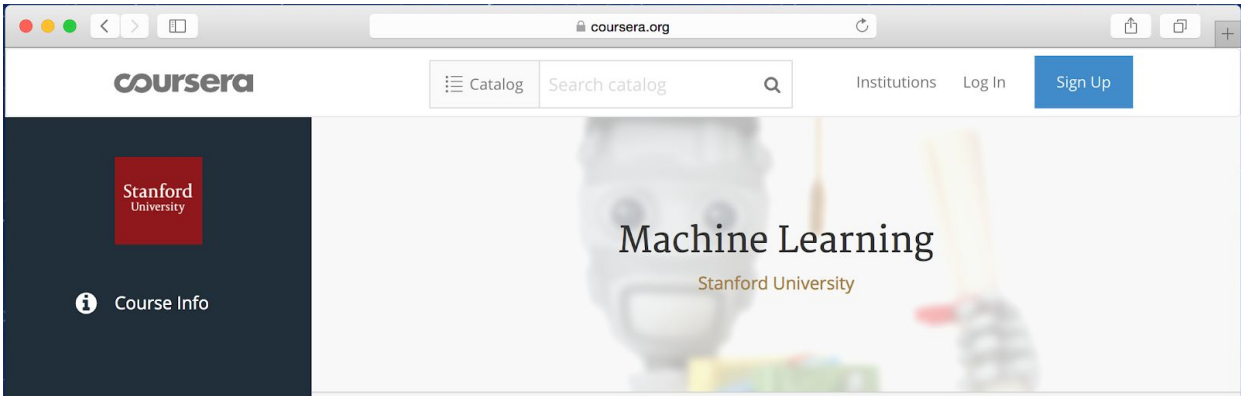
3 Prerequisites and Notation

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



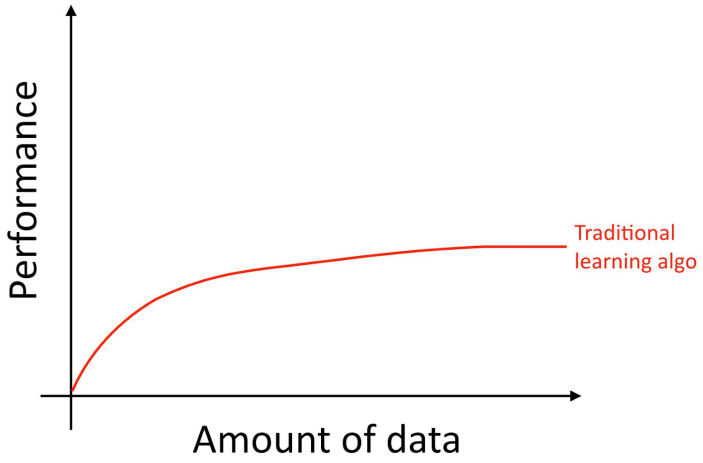
4 Scale drives machine learning progress

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:

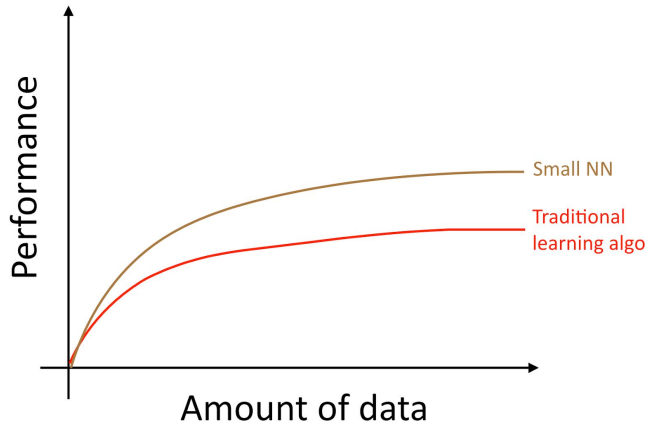
- **Data availability.** People are now spending more time on digital devices (laptops, mobile devices). Their digital activities generate huge amounts of data that we can feed to our learning algorithms.
- **Computational scale.** We started just a few years ago to be able to train neural networks that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

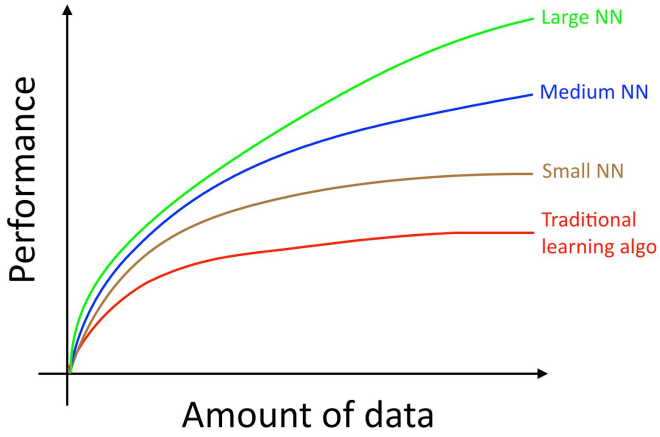


It was as if the older algorithms didn't know what to do with all the data we now have.

If you train a small neutral 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 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 sets 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.

Classifier	Precision	Recall	F1 score
A	95%	90%	**92.4%**

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

[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.

score," which is a modified way of computing their average, and works better than simply taking the mean.[4]

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.

[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)).

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. Lets 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

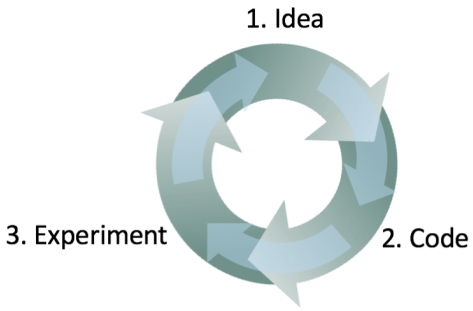
to minimize the false negative rate (optimizing metric), subject to there being no more than one false positive every 24 hours of operation (satisficing metric).

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 many 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 envelope/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 just 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.

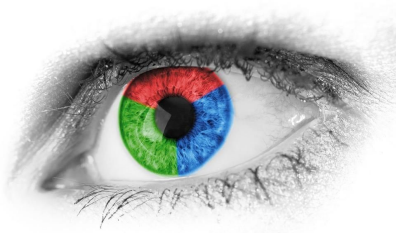
Whatever process you apply to fixing dev set labels, remember to apply it to the test set labels too so that your dev and test sets continue to be drawn from the same distribution. Fixing your dev and test sets together would prevent the problem we discussed in Chapter 6, where your team optimizes for 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.



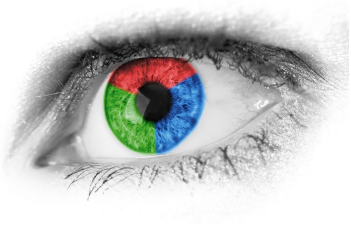
Let's continue our example above, in which the algorithm is misclassifying 1,000 out of 5,000 dev set examples. Suppose we want to manually examine about 100 errors for error analysis (10% of the errors). You should randomly select 10% of the dev set and place that into what we'll call an **Eyeball dev 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**.[6]

[6] 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.[7] 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.

[7] 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.[8]
- **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:[9]

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

[8] 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.

[9] 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.

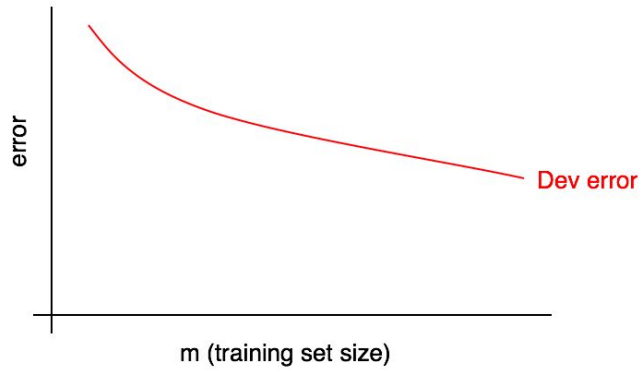
- **Modify model architecture** (such as neural network architecture) so that it is more suitable for your problem: This technique can affect both bias and variance.

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:



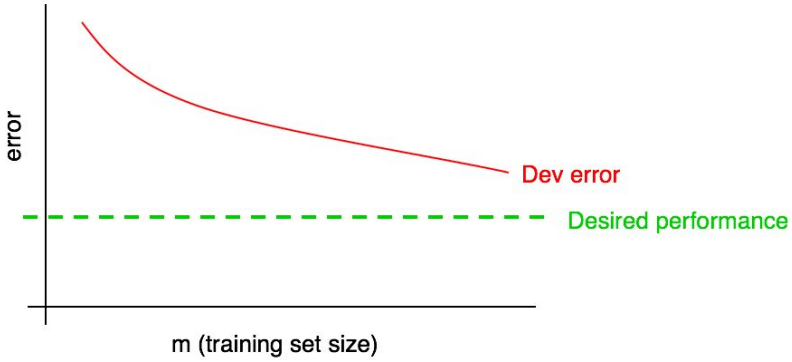
As the training set size increases, the dev set error should decrease.

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.

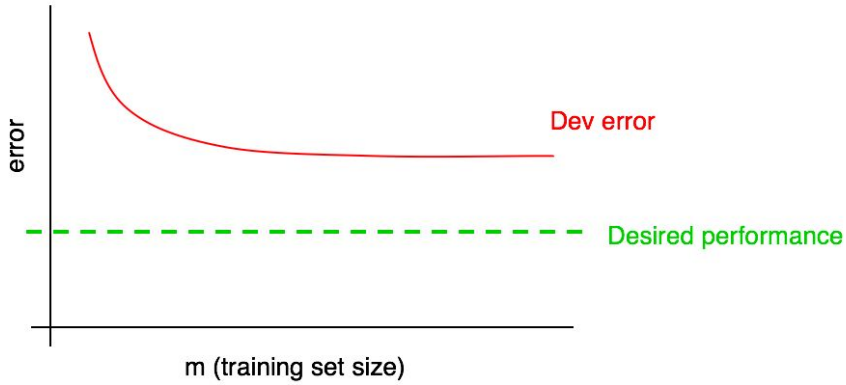
- If you have worked on a important application for a long time, then you might have intuition about how much more progress you can reasonably make in the next quarter/year.

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

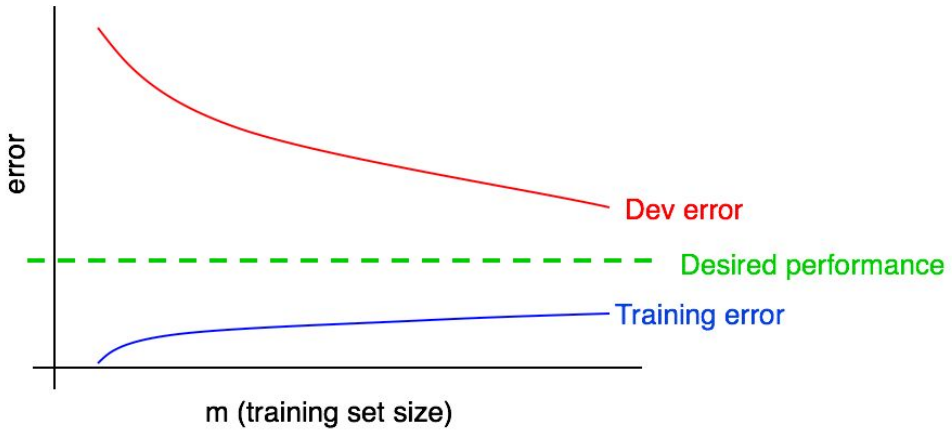
Your dev set (and test set) error should decrease as the training set size grows. But your training set error usually *increases* as the training set size grows.

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:

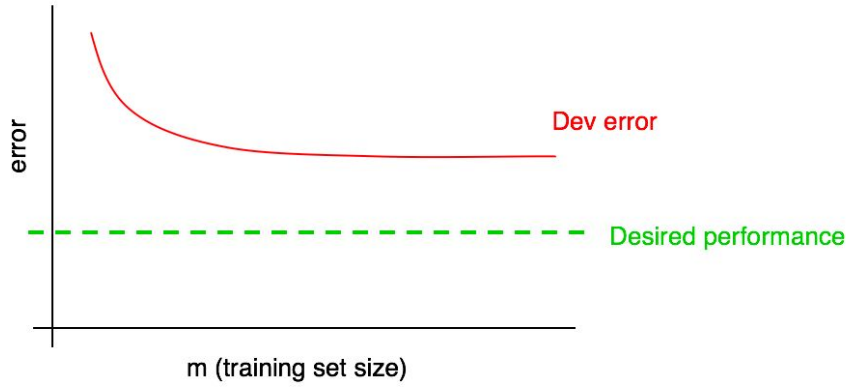


You can see that the blue "training error" curve increases with the size of the training set. Furthermore, your algorithm usually does better on the training set than on the dev set; thus the red dev error curve usually lies strictly above the blue training error curve.

Let's discuss next how to interpret these plots.

30 Interpreting learning curves: High bias

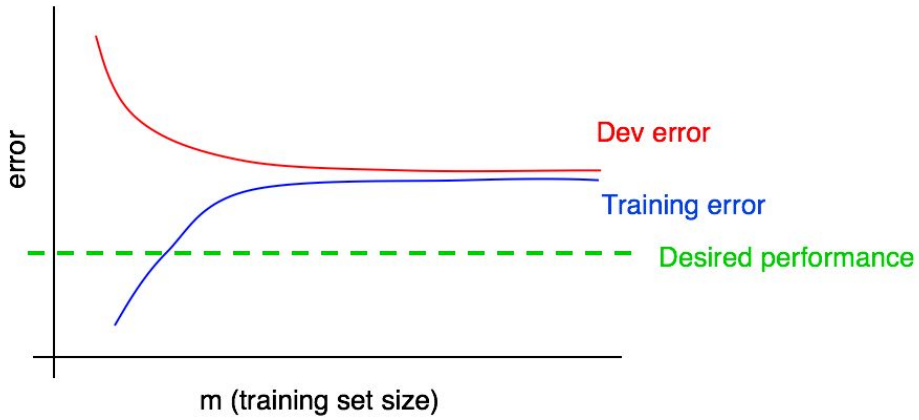
Suppose your dev error curve looks like this:



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.