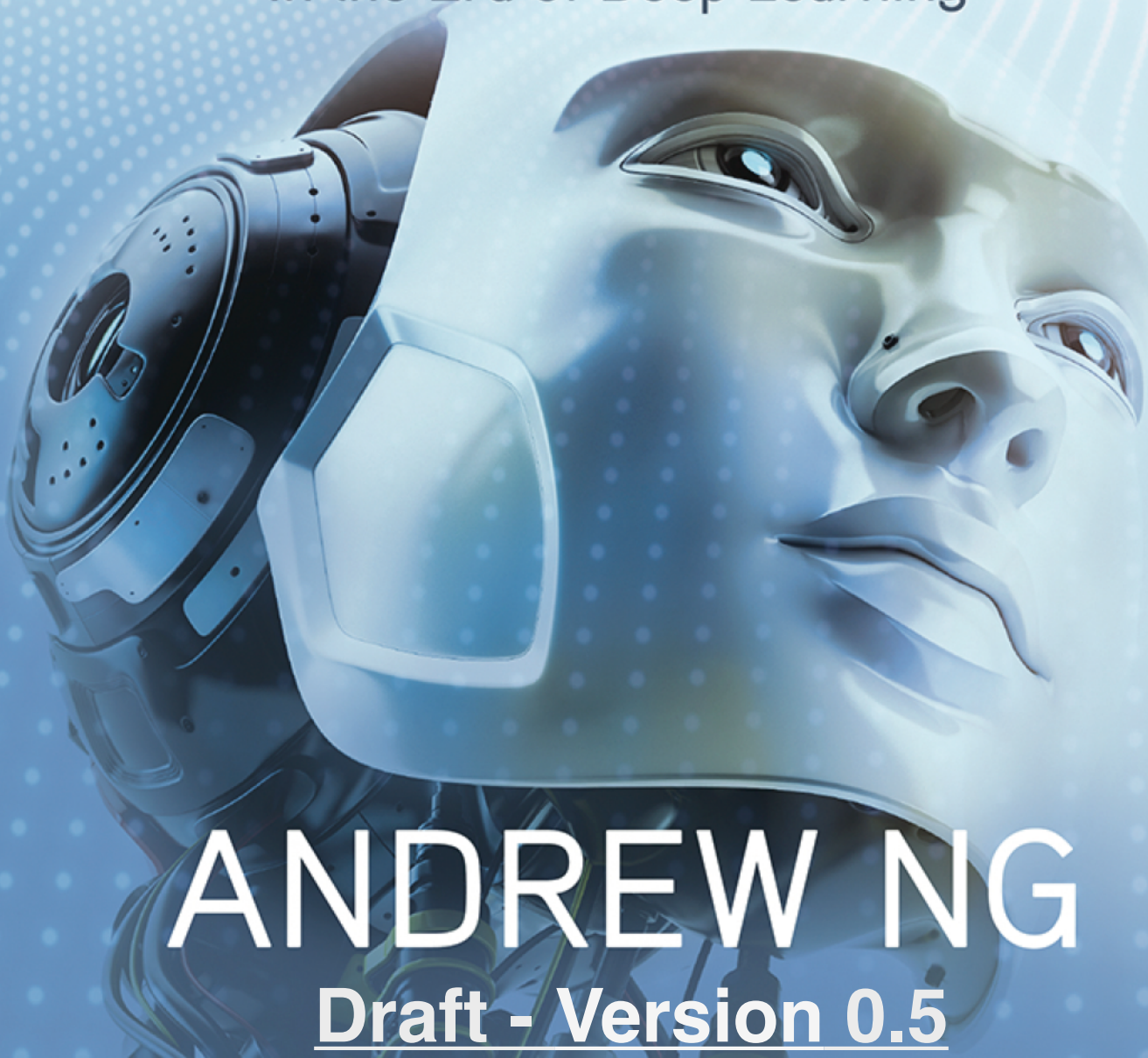
Draft - Version 0.5

MACHINE LEARNING YEARNING

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



ANDREW NG

Draft - Version 0.5

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 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

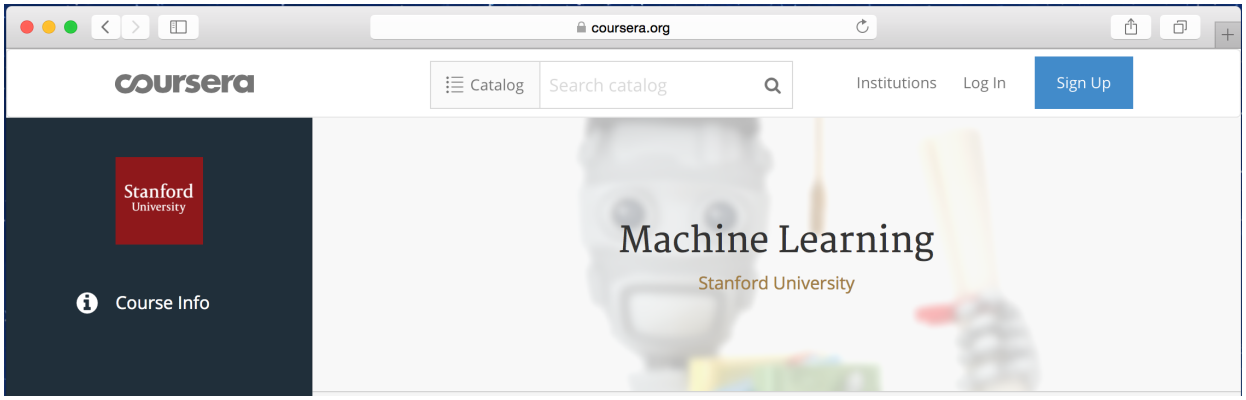
Prerequisites and Notation

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 is from supervised learning.

I will frequently refer to neural networks (also known as "deep learning"). You'll need only 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

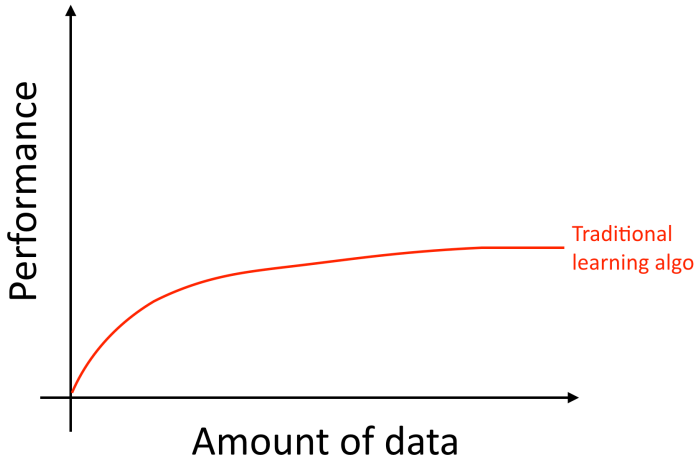
Scale drives machine learning progress

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:

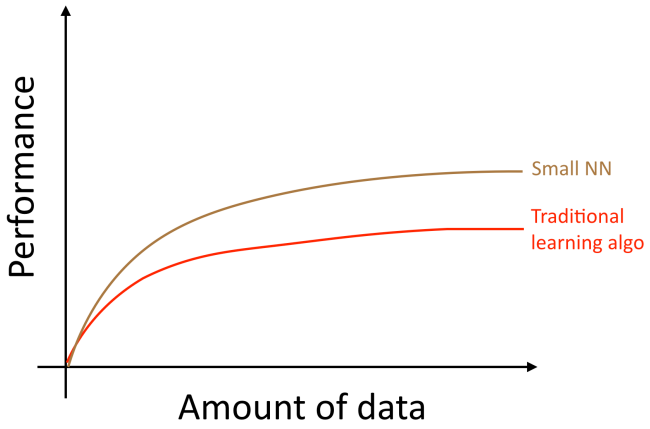
- **Data availability.** People are now spending more time on digital devices (laptops, mobile devices). Their digital activities generate huge amounts of data that we can feed to our learning algorithms.
- **Computational scale.** We started just a few years ago to be able to train neural networks that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

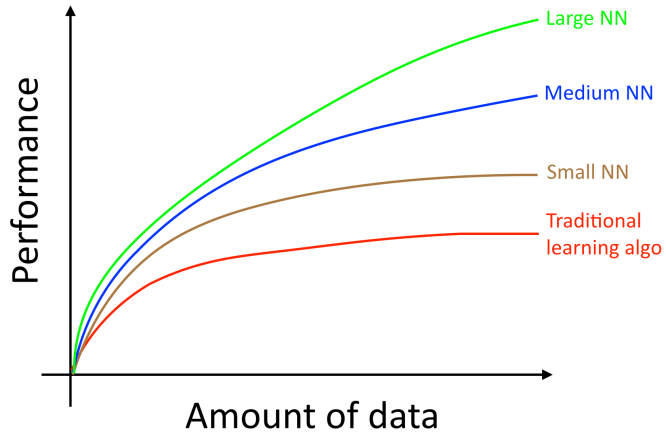


It was as if the older algorithms didn't know what to do with all the data we now have.

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.

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.

[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.

Setting up development and test sets

5

Your development and test sets

Lets 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 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 have less ideal lighting. Since your training/test sets were made of website images, your algorithm did not generalize well to the actual distribution you care about of smartphone 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 is 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 about regarding what learning algorithm or parameters to use.

One 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 order words, your test set should not simply be 30% of the available data, especially if you expect your future data (mobile app 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 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, we can randomly assign two of these segments to the dev set, and the other two to the test set, right? Say US and India in the dev set; China and Other in the test set.

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 most to improve on: 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 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're 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 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 "geometric 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 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).

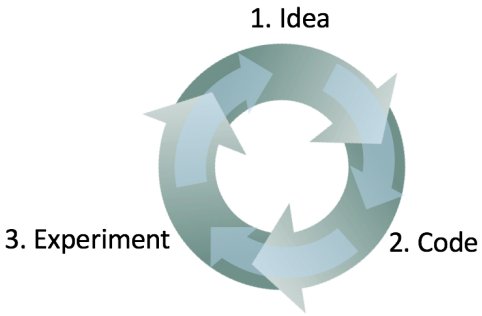
Once your team is aligned on the evaluation metric to optimize, they will be able to make faster progress.

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—almost never longer. It it 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're 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 ever 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.