Course 3: Structuring Machine Learning Projects

Author:	Pradeep K. Pant
URL:	https://www.coursera.org/learn/machine-learning-projects/home/welcome

****

Course 3: Structuring Machine Learning Projects

****

In this course we'll learn how to build a successful machine learning project. If you aspire to be a technical leader in AI, and know how to set direction for your team's work, this course will show you how.

Much of this content has never been taught elsewhere, and is drawn from my experience building and shipping many deep learning products. This course also has two "flight simulators" that let you practice decision-making as a machine learning project leader. This provides "industry experience" that you might otherwise get only after years of ML work experience.
After 2 weeks, you will:
- Understand how to diagnose errors in a machine learning system, and
- Be able to prioritize the most promising directions for reducing error
- Understand complex ML settings, such as mismatched training/test sets, and comparing to and/or surpassing human-level performance
- Know how to apply end-to-end learning, transfer learning, and multi-task learning

Week 1: ML Strategy (1)

Learning Objectives

- Understand why Machine Learning strategy is important
- Apply satisfying and optimizing metrics to set up your goal for ML projects
- Choose a correct train/dev/test split of your dataset
- Understand how to define human-level performance
- Use human-level perform to define your key priorities in ML projects
- Take the correct ML Strategic decision based on observations of performances and dataset

Introduction to ML Strategy

Why ML Strategy

In this course we'll learn how to much more quickly and efficiently get your machine learning systems working. So, what is machine learning strategy. Let's start with a motivating example. Let's say you are working on your cat classifier. And after working it for some time, you've gotten your system to have 90% accuracy, but this isn't good enough for your application. You might then have a lot of ideas as to how to improve your system. For example, you might think well let's **collect more data**, more training data. Or you might say, maybe your training set isn't diverse enough yet, you should collect images of cats in more diverse poses, or maybe a more diverse set of negative examples. Well maybe you want to train the algorithm longer with **gradient descent**. Or maybe you want to try a different optimization algorithm, like the **Adam optimization algorithm**. Or maybe trying a **bigger network** or a smaller network or maybe you want to try to dropout or maybe **L2 regularization**. Or maybe you want to change the network architecture such as changing activation functions, changing the number of hidden units and so on and so on. When trying to improve a deep learning system, you often have a lot of ideas or things you could try. And the problem is that if you choose poorly, it is entirely possible that you end up spending six months charging in some direction only to realize after six months that that didn't do any good. For example, I've seen some teams spend literally six months collecting more data only to realize after six months that it barely improved the performance of their system. So, assuming you don't have six months to waste on your problem, won't it be nice if you had quick and effective ways to figure out which of all of these ideas and maybe even other ideas, are worth pursuing and which ones you can safely discard. So what I hope to do in this course is teach you a number of strategies, that is, ways of analyzing a machine learning problem that will point you in the direction of the most promising things to try. What I will do in this course also is share with you a number of lessons I've learned through building and shipping large number of deep learning products. And I think these materials are actually quite unique to this course. I don't see a lot of these ideas being taught in universities' deep learning courses for example. It turns out also that machine learning strategy is changing in the era of deep learning because the things you could do are now different with deep learning algorithms than with previous generation of machine learning algorithms. I hope that these ideas will help you become much more effective at getting your deep learning systems to work.

Orthogonalization

One of the challenges with building machine learning systems is that there's so many things you could try, so many things you could change. Including, for example, so many hyperparameters you could tune. One of the things I've noticed is about the most effective machine learning people is they're very clear-eyed about what to tune in order to try to achieve one effect. This is a process we call **orthogonalization**. Let me tell you what I mean.

For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true. First, is that you usually have to make sure that you're at **least doing well on the training set**. So performance on the training set needs to pass some acceptability assessment. For some applications, this might mean doing comparably to human level performance. But this will depend on your application, and we'll talk more about comparing to human level performance next week.

But after doing well on the training sets, you then hope that this leads to also doing well on the **dev set**. And you then hope that this also does well on the **test set**. And finally, you hope that doing well on the test set on the cost function results in your system performing in the real world. So you hope that this resolves in happy cat picture app users, for example. So to relate back to the TV tuning example, if the picture of your TV was either too wide or too narrow, you wanted one knob to tune in order to adjust that. You don't want to have to carefully adjust five different knobs, which also affect different things. You want one knob to just affect the width of your TV image. So in a similar way, if your algorithm is not fitting the training set well on the cost function, you want one knob, yes, that's my attempt to draw a knob. Or maybe one specific set of knobs that you can use, to make sure you can tune your algorithm to make it fit well on the training set. So the knobs you use to tune this are, you might train a bigger network or you might switch to a better optimization algorithm, like the **Adam optimization algorithm**, and so on. In contrast, if you find that the algorithm is not fitting the dev set well, then there's a separate set of knobs. Yes, that's my not very artistic rendering of another knob, you want to have a distinct set of knobs to try. So for example, if your algorithm is not doing well on the dev set, it's doing well on the training set but not on the dev set, then you have a set of knobs around regularization that you can use to try to make it satisfy the second criteria. So the analogy is, now that you've tuned the width of your TV set, if the height of the image isn't quite right, then you want a different knob in order to tune the height of the TV image. And you want to do this hopefully without affecting the width of your TV image too much. And getting a bigger training set would be another knob you could use, that helps your learning algorithm generalize better to the dev set. Now, having adjusted the width and height of your TV image, well, what if it doesn't meet the third criteria? What if you do well on the dev set but not on the test set? If that happens, then the knob you tune is, you probably want to get a bigger dev set. Because if it does well **on the dev set but not the test set, it probably means you've overtuned to your dev set, and you need to go back and find a bigger dev set** And finally, if it does well on the test set, but it isn't delivering to you a happy cat picture app user, then what that means is that you want to go back and change either the dev set or the cost function. Because if doing well on the test set according to some cost function doesn't correspond to your algorithm doing what you need it to do in the real world, then it means that either your dev test set distribution isn't set correctly, or your cost function isn't measuring the right thing.

So Orthogonalization or orthogonality is a system design property that assures that modifying an instruction or a component of an algorithm will not create or propagate side effects to other components of the system. It becomes easier to verify the algorithms independently from one another, it reduces testing and development time. When a supervised learning system is design, these are the 4 assumptions that needs to be true and orthogonal.

1. Fit training set well in cost function
- If it doesn’t fit well, the use of a bigger neural network or switching to a better optimization algorithm might help.
2. Fit development set well on cost function
- If it doesn’t fit well, regularization or using bigger training set might help.
3. Fit test set well on cost function
- If it doesn’t fit well, the use of a bigger development set might help
4. Performs well in real world
- If it doesn’t perform well, the development test set is not set correctly or the cost function is not evaluating the right thing

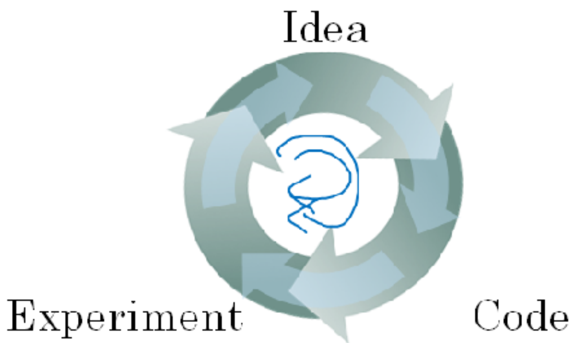
Setting up your goal

Single number evaluation metric

Whether you're tuning hyperparameters, or trying out different ideas for learning algorithms, or just trying out different options for building your machine learning system. You'll find that your progress will be much faster if you have a **single real number evaluation metric** that lets you quickly tell if the new thing you just tried is working better or worse than your last idea. So when teams are starting on a machine learning project, I often recommend that you set up a single real number evaluation metric for your problem. Let's look at an example.

You've heard me say before that applied machine learning is a very empirical process. We often have an idea, code it up, run the experiment to see how it did, and then use the outcome of the experiment to refine your ideas. And then keep going around this loop as you keep on improving your algorithm. So let's say for your classifier, you had previously built some classifier A. And by changing the hyperparameters and the training sets or some other thing, you've now trained a new classifier, B. So one reasonable way to evaluate the performance of your classifiers is to look at its precision and recall. The exact details of what's precision and recall don't matter too much for this example. But briefly, the definition of precision is, of the examples that **your classifier recognizes as cats**, What percentage actually are cats? So if classifier A has 95% precision, this means that when classifier A says something is a cat, there's a 95% chance it really is a cat. And recall is, **of all the images that really are cats**, what percentage were correctly recognized by your classifier? So what percentage of actual cats, Are correctly recognized?

Using a single number evaluation metric



→ Of examples recognized as cat, what % actually are cats?

→ what % of actual cats are correctly recognized

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

F_1 score = "Average" of P and R.

$\left(\frac{2}{\frac{1}{P}+\frac{1}{R}} \right.$. "Harmonic mean" $\left. \right)$

Dev set + Single number evaluation metric

real

Speed up iterating

So if classifier A is 90% recall, this means that of all of the images in, say, your dev sets that really are cats, classifier A accurately pulled out 90% of them. So don't worry too much about the definitions of precision and recall. **It turns out that there's often a tradeoff between precision and recall, and you care about both**. You want that, when the classifier says something is a cat, there's a high chance it really is a cat. But of all the images that are cats, you also want it to pull a large fraction of them as cats. So it might be reasonable to try to evaluate the classifiers in terms of its precision and its recall. The problem with using precision recall as your evaluation metric is that if classifier A does better on recall, which it does in the diagram above, the classifier B does better on precision, then you're not sure which classifier is better and if you're trying out a lot of different ideas, a lot of different hyperparameters, you want to rather quickly try out not just two classifiers, but maybe a dozen classifiers and quickly pick out the, quote, best ones, so you can keep on iterating from there and with two evaluation metrics, it is difficult to know how to quickly pick one of the two or quickly pick one of the ten.

So what is recommended is rather than using two numbers, precision and recall, to pick a classifier, you just have to find a new evaluation metric that combines precision and recall. In the machine learning literature, the standard way to combine precision and recall is something called an F1 score. And the details of F1 score aren't too important, but informally, you can think of this as the average of precision, P, and recall, R.
Check below diagram for details on F1 score.

Single number evaluation metric

To choose a classifier, a well-defined development set and an evaluation metric speed up the iteration process.

Example : Cat vs Non- cat

y = 1, cat image detected

Predict class $\hat{y}$ \ Actual class y	1	0
1	True positive	False positive
0	False negative	True negative

Precision

Of all the images we predicted y=1, what fraction of it have cats?

Precision (%) = $\frac{True\ positive}{Number\ of\ predicted\ positive} x\ 100 = \frac{True\ positive}{(True\ positive+False\ positive)}\ x\ 100$

Recall

Of all the images that actually have cats, what fraction of it did we correctly identifying have cats?

Recall (%) = $\frac{True\ positive}{Number\ of\ predicted\ actually\ positive} x\ 100 = \frac{True\ positive}{(True\ positive+False\ negative)}\ x\ 100$

Let's compare 2 classifiers A and B used to evaluate if there are cat images:

Classifier	Precision (p)	Recall (r)
A	95%	90%
B	98%	85%

In this case the evaluation metrics are precision and recall.

For classifier A, there is a 95% chance that there is a cat in the image and a 90% chance that it has correctly detected a cat. Whereas for classifier B there is a 98% chance that there is a cat in the image and a 85% chance that it has correctly detected a cat.

The problem with using precision/recall as the evaluation metric is that you are not sure which one is better since in this case, both of them have a good precision et recall. F1-score, a harmonic mean, combine both precision and recall.

$$\text{F1-Score}= \frac{2}{\frac{1}{p}+\frac{1}{r}}$$

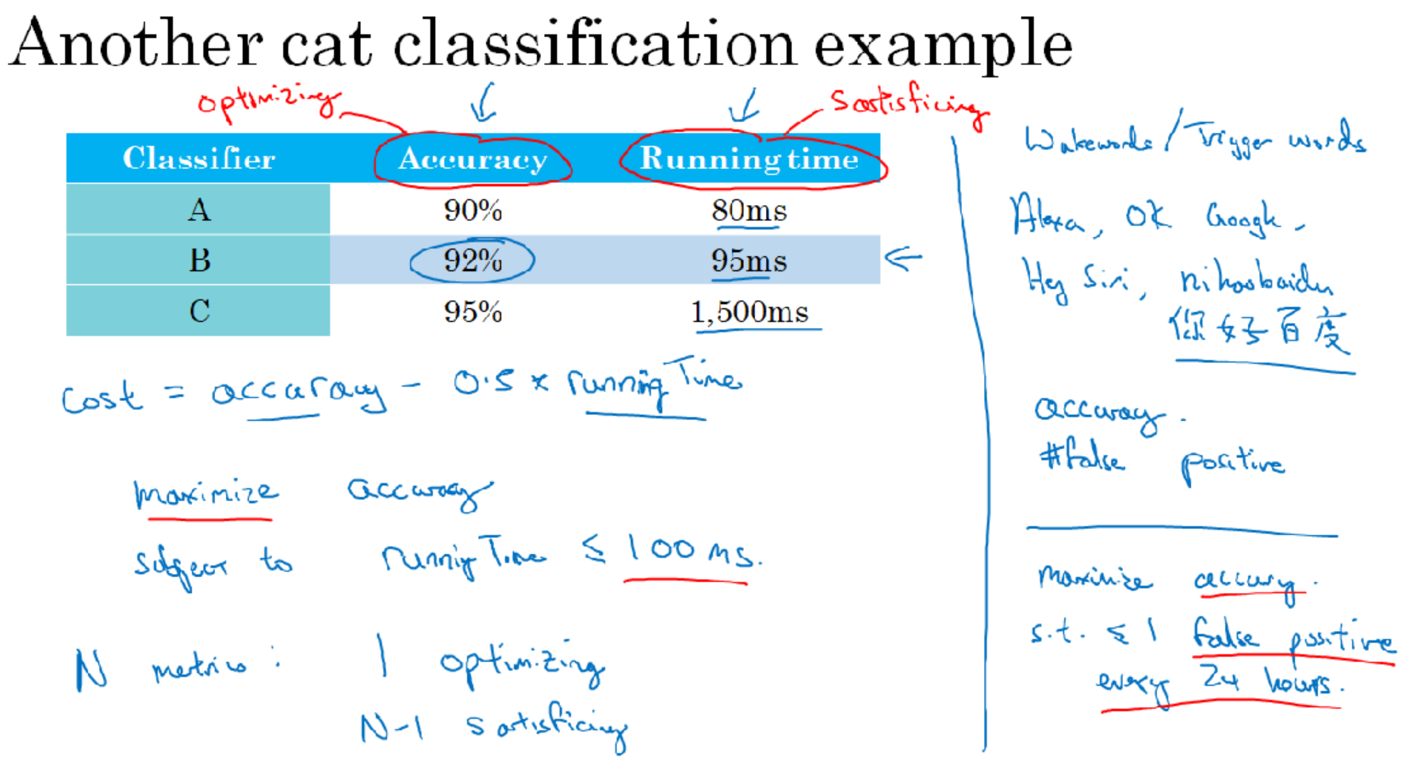
Classifier	Precision (p)	Recall (r)	F1-Score
A	95%	90%	92.4 %
B	98%	85%	91.0%

Classifier A is a better choice. F1-Score is not the only evaluation metric that can be use, the average, for example, could also be an indicator of which classifier to use.

So what we have learned in this section is that having a single number evaluation metric can really improve your efficiency or the efficiency of your tea, in making decisions.

Satisficing and optimistic metric

It's not always easy to combine all the things you care about into a single row number evaluation metric. In those cases I've found it sometimes useful to set up **satisficing as well as optimizing matrix**. Let me show you what I mean. Let's say that you've decided you care about the classification accuracy of your cat's classifier, this could have been **F1 score** or some other measure of accuracy, but let's say that in addition to accuracy you also care about the **running time**.



So how long it takes to classify an image and classifier A takes 80 milliseconds, B takes 95 milliseconds, and C takes 1,500 milliseconds, that's 1.5 seconds to classify an image. So one thing you could do is combine accuracy and running time into an overall evaluation metric. And so the costs such as maybe the overall cost is accuracy minus 0.5 times running time. But maybe it seems a bit artificial to combine accuracy and running time using a formula like this, like a linear weighted sum of these two things. **So here's something else you could do instead which is that you might want to choose a classifier that maximizes accuracy but subject to that the running time, that is the time it takes to classify an image, that that has to be less than or equal to 100 milliseconds. So in this case we would say that accuracy is an optimizing metric because you want to maximize accuracy**. You want to do as well as possible on accuracy but that running time is what we call a **satisficing metric**. Meaning that it just has to be good enough, it just needs to be **less than 100 milliseconds** and beyond that you don't really care, or at least you don't care that much. So this will be a pretty reasonable way to trade off or to put together accuracy as well as running time. And it may be the case that so long as the running time is less that 100 milliseconds, your users won't care that much whether it's 100 milliseconds or 50 milliseconds or even faster. And by defining optimizing as well as satisficing matrix, this gives you a clear way to pick the, quote, best classifier, which in this case would be classifier B because of all the ones with a running time better than 100 milliseconds it has the best accuracy.

Satisficing and optimizing metric

There are different metrics to evaluate the performance of a classifier, they are called evaluation matrices. They can be categorized as satisficing and optimizing matrices. It is important to note that these evaluation matrices must be evaluated on a training set, a development set or on the test set.

Example: Cat vs Non-cat

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

In this case, accuracy and running time are the evaluation matrices. Accuracy is the optimizing metric, because you want the classifier to correctly detect a cat image as accurately as possible. The running time which is set to be under 100 ms in this example, is the satisficing metric which mean that the metric has to meet expectation set.

The general rule is:

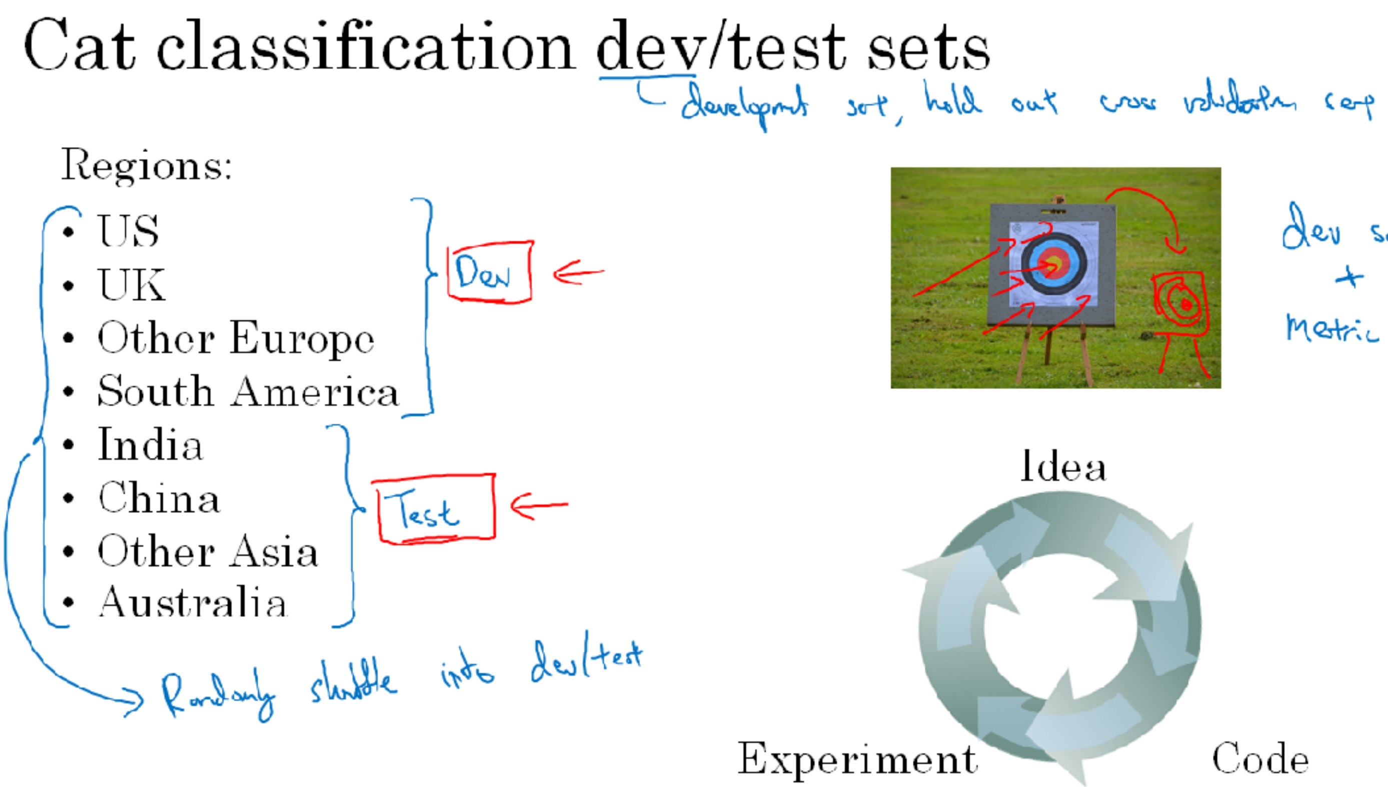
$$N_{metric}: \begin{cases} 1 & Optimizing\ metric \\ N_{metric} - 1 & Satisficing\ metric \end{cases}$$

Here's another example. Let's say you're building a system to detect wake words, also called trigger words. So this refers to the voice control devices like the Amazon Echo where you wake up by saying Alexa or some Google devices which you wake up by saying okay Google or some Apple devices which you wake up by saying Hey Siri. So these are the wake words you use to tell one of these voice control devices to wake up and listen to something you want to say so you might care about the accuracy of your trigger word detection system. So when someone says one of these trigger words, how likely are you to actually wake up your device, and you might also care about the number of false positives. So when no one actually said this trigger word, how often does it randomly wake up? So in this case maybe one reasonable way of combining these two evaluation matrix might be to maximize accuracy, so when someone says one of the trigger words, maximize the chance that your device wakes up. And subject to that, you have at most one false positive every 24 hours of operation, right? So that your device randomly wakes up only once per day on average when no one is actually talking to it. So in this case accuracy is the optimizing metric and a number of false positives every 24 hours is the **satisficing metric** where you'd be satisfied so long as there is at most one false positive every 24 hours.

To summarize, if there are multiple things you care about by say there's one as the optimizing metric that you want to do as well as possible on and one or more as satisficing metrics were you'll be satisfice. Almost it does better than some threshold you can now have an almost automatic way of quickly looking at multiple core size and picking the, quote, best one. Now these evaluation matrix must be evaluated or calculated on a training set or a development set or maybe on the test set. So one of the things you also need to do is set up training, dev or development, as well as test sets.

Train/dev/test distributions

The way you set up your training dev, or development sets and test sets, can have a huge impact on how rapidly you or your team can make progress on building machine learning application. The same teams, even teams in very large companies, set up these data sets in ways that really slows down, rather than speeds up, the progress of the team. Let's take a look at how you can set up these data sets to maximize your team's efficiency. In this section, I want to focus on how you set up your dev and test sets. So, that dev set is also called the development set, or sometimes called the hold out cross validation set. And, workflow in machine learning is that you try a lot of ideas, train up different models on the training set, and then use the dev set to evaluate the different ideas and pick one. And, keep innovating to improve dev set performance until, finally, you have one clause that you're happy with that you then evaluate on your test set. Now, let's say, by way of example, that you're building a cat classifier, and you are operating in these regions: in the U.S, U.K, other European countries, South America, India, China, other Asian countries, and Australia. So, how do you set up your dev set and your test set? Well, one way you could do so is to pick four of these regions (check diagram below). I'm going to use these four but it could be four randomly chosen regions. And say, that data from these four regions will go into the dev set. And, the other four regions, I'm going to use these four, could be randomly chosen four as well, that those will go into the test set.



It turns out, this is a very bad idea because in this example, your dev and test sets come from different distributions. I would, instead, recommend that you find a **way to make your dev and test sets come from the same distribution**. So, here's what I mean. One picture to keep in mind is that, I think, setting up your dev set, plus, your single role number evaluation metric, that's like placing a target and telling your team where you think is the bull's eye you want to aim at. Because, what happen once you've established that dev set and the metric is that, the team can innovate very quickly, try different ideas, run experiments and very quickly use the dev set and the metric to evaluate crossfires and try to pick the best one. So, machine learning teams are often very good at shooting different arrows into targets and innovating to get closer and closer to hitting the bullseye. So, doing well on your metric on your dev sets. And, the problem with how we've set up the dev and test sets in the example on the left is that, your team might spend months innovating to do well on the dev set only to realize that, when you finally go to test them on the test set, that data from these four countries or these four regions at the bottom, might be very different than the regions in your dev set. So, you might have a nasty surprise and realize that, all the months of work you spent optimizing to the dev set, is not giving you good performance on the test set. **So, having dev and test sets from different distributions is like**

setting a target, having your team spend months trying to aim closer and closer to bull's eye, only to realize after months of work that, you'll say, "Oh wait, to test it, I'm going to move target over here." And, the team might say, "Well, why did you make us spend months optimizing for a different bull's eye when suddenly, you can move the bull's eye to a different location somewhere else?" So, to avoid this, what **I recommend instead is that, you take all this randomly shuffled data into the dev and test set**. **So that, both the dev and test sets have data from all eight regions and that the dev and test sets really come from the same distribution, which is the distribution of all of your data mixed together.** Here's another example. This is a, actually, true story but with some details changed. So, I know a machine learning team that actually spent several months optimizing on a dev set which was comprised of loan approvals for medium income zip codes. So, the specific machine learning problem was, "Given an input X about a loan application, can you predict why and which is, whether or not, they'll repay the loan?" So, this helps you decide whether or not to approve a loan. And so, the dev set came from loan applications. They came from medium income zip codes. Zip codes is what we call postal codes in the United States. But, after working on this for a few months, the team then, suddenly decided to test this on data from low income zip codes or low income postal codes. And, of course, the distributional data for medium income and low income zip codes is very different. And, the classifier, that they spend so much time optimizing in the former case, just didn't work well at all on the latter case. And so, this particular team actually wasted about three months of time and had to go back and really re-do a lot of work. And, what happened here was, the team spent three months aiming for one target, and then, after three months, the manager asked, "Oh, how are you doing on hitting this other target?" This is a totally different location. And, it just was a very frustrating experience for the team. So, what I recommand for setting up a dev set and test set is, **choose a dev set and test set to reflect data you expect to get in future and consider important to do well on.** And, in particular, the dev set and the test set here, should come from the same distribution. So, whatever type of data you expect to get in the future, and once you do well on, try to get data that looks like that. And, whatever that data is, put it into both your dev set and your test set. Because that way, you're putting the target where you actually want to hit and you're having the team innovate very efficiently to hitting that same target, hopefully, the same targets well.

Important take away from this section is that, setting up the dev set, as well as the validation metric, is really defining what target you want to aim at. And hopefully, by setting the dev set and the test set to the same distribution, you're really aiming at whatever target you hope your machine learning team will hit. **The way you choose your training set will affect how well you can actually hit that target** but, we can talk about that separately in upcoming section.

Size of the dev and test sets

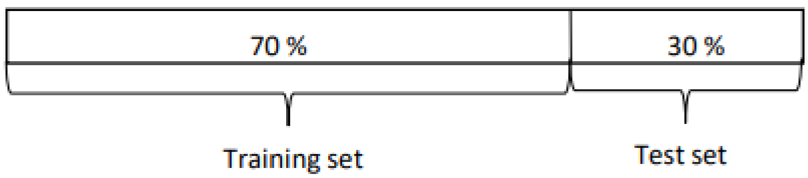
In the last section, we saw how our dev and test sets should come from the same distribution, but how long should they be? The guidelines to help set up your dev and test sets are changing in the Deep Learning era. Let's take a look at some best practices. You might have heard of the rule of thumb in machine learning of taking all the data you have and using a 70/30 split into a train and test set, or have you had to set up train dev and test sets maybe, you would use a 60% training and 20% dev and 20% tests. In earlier eras of machine learning, this was pretty reasonable, especially back when data set sizes were just smaller. So if you had a hundred examples in total, these 70/30 or 60/20/20 rule of thumb would be pretty reasonable. If you had thousand examples, maybe if you had ten thousand examples, these things are not unreasonable. But in the modern machine learning era, we are now used to working with much larger data set sizes. So let's say you have a million training examples, it might be quite reasonable to set up your data so that you have 98% in the training set, 1% dev, and 1% test because if you have a million examples, then 1% of that, is 10,000 examples, and that might be plenty enough for a dev set or for a test set. So, in the modern Deep Learning era where sometimes we have much larger data sets, It's quite reasonable to use a much smaller than 20 or 30% of your data for a dev set or a test set. And because Deep Learning algorithms have such a huge hunger for data, I'm seeing that, the problems we have large data sets that have much larger fraction of it goes into the training set. So, how about the test set? Remember the purpose of your test set is that, after you finish developing a system, the **test set helps evaluate how good your final system is**. The guideline is, to set your test set to big enough to give high confidence in the overall performance of your system. So, unless you need to have a very accurate measure of how well your final system is performing, maybe you don't need millions and millions of examples in a test set, and maybe for your application if you think that having 10,000 examples gives you enough confidence to find the performance on maybe 100,000 or whatever it is, that might be enough. And this could be much less than, say 30% of the overall data set, depend on how much data you have. For some applications, maybe you don't need a high confidence in the overall performance of your final system. Maybe all you need is a train and dev set, And I think, not having a test set might be okay. In fact, what sometimes happened was, people were talking about using train test splits but what they were actually doing was iterating on the test set. So rather than test set, what they had was a train dev split and no test set. If you're actually tuning to this set, to this dev set and this test set, It's better to call the dev set. Although I think in the history of machine learning, not everyone has been completely clean and completely records of about calling the dev set when it really should be treated as test set. But, if all you care about is having some data that you train on, and having some data to tune to, and you're just going to shake the final system and not worry too much about how it was actually doing, I think it will be healthy and just call the train dev set and acknowledge that you have no test set. This a bit unusual? I'm definitely not recommending not having a test set when building a system. I do find it reassuring to have a separate test set you can use to get an unbiased estimate of how I was doing before you shift it, but if you have a very large dev set so that you think you won't overfit the dev set too badly. Maybe it's not totally unreasonable to just have a train dev set, although it's not what I usually recommend.

So to summarize, in the era of big data, I think the old rule of thumb of a 70/30 is that, that no longer applies. And the trend has been to use more data for training and less for dev and test, especially when you have a very large data sets. And the rule of thumb is really to try to set the dev set to big enough for its purpose, which helps you evaluate different ideas and the purpose of test set is to help you evaluate your final cost. You just have to set your test set big enough for that purpose, and that could be much less than 30% of the data. Check below diagram for summary:

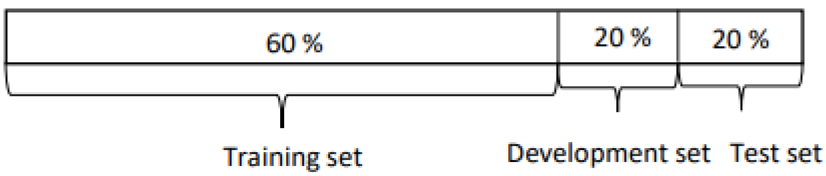
Size of the development and test sets

Old way of splitting data

We had smaller data set therefore we had to use a greater percentage of data to develop and test ideas and models.

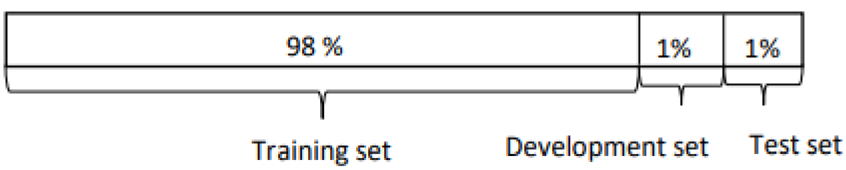


Or



Modern era – Big data

Now, because a large amount of data is available, we don't have to compromised as much and can use a greater portion to train the model.



Guidelines

- Set up the size of the test set to give a high confidence in the overall performance of the system.
- Test set helps evaluate the performance of the final classifier which could be less 30% of the whole data set.
- The development set has to be big enough to evaluate different ideas.

When to change dev/test sets and metrics

You've seen how set to have a dev set and evaluation metric is like placing a target somewhere for your team to aim at. But sometimes partway through a project you might realize you put your target in the wrong place. In that case you should move your target. Let's take a look at an example. Let's say you build a cat classifier to try to find lots of pictures of cats to show to your cat loving users and the metric that you decided to use is classification error. So algorithms A and B have, respectively, 3 percent error and 5 percent error, so it seems like Algorithm A is doing better. But let's say you try out these algorithms, you look at these algorithms and Algorithm A, for some reason, is letting through a lot of the pornographic images. So if you shift Algorithm A the users would see more cat images because you'll see 3 percent error and identify cats, but it also shows the users some pornographic images which is totally unacceptable both for your company, as well as for your users. In contrast, Algorithm B has 5 percent error so this classifies fewer images but it doesn't have pornographic images. So from your company's point of view, as well as from a user acceptance point of view, Algorithm B is actually a much better algorithm because it's not letting through any pornographic images. So, what has happened in this example is that Algorithm A is doing better on evaluation metric. It's getting 3 percent error but it is actually a worse algorithm. In this case, the evaluation metric plus the dev set prefers Algorithm A because they're saying, look, Algorithm A has lower error which is the metric you're using but you and your users prefer Algorithm B because it's not letting through pornographic images. So when this happens, when your evaluation metric is no longer correctly rank ordering preferences between algorithms, in this case is mispredicting that Algorithm A is a better algorithm, then that's a sign that you should change your evaluation metric or perhaps your development set or test set.

Cat dataset examples

Metric + Dev : Prefer A
You/users : Prefer B.

→ Metric: classification error

Algorithm A: 3% error → pornographic

✓ Algorithm B: 5% error

Error: $\frac{1}{\sum w^{(i)}} \cancel{\frac{1}{m_{dev}}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathcal{I}\{ y_{pred}^{(i)} \neq y^{(i)} \}$

predicted value (0/1)

→ $w^{(i)} = \begin{cases} 1 & \text{if } x^{(i)} \text{ is non-porn} \\ 10 & \text{if } x^{(i)} \text{ is porn} \end{cases}$

The problem with this evaluation metric is that they treat pornographic and non-pornographic images equally but you really want your classifier to not mislabel pornographic images, like maybe you recognize a pornographic image in cat image and therefore show it to unsuspecting user, therefore very unhappy with unexpectedly seeing porn. One way to change this evaluation metric would be if you add the weight term.

Here, we call this $w^{(i)}$ where $w^{(i)}$ is going to be equal to 1 if $x^{(i)}$ is non-porn and maybe 10 or maybe even large number like a 100 if $x^{(i)}$ is porn. So this way you're giving a much larger weight to examples that are pornographic so that the error term goes up much more if the algorithm makes a mistake on classifying a pornographic image as a cat image. In this example you giving 10 times bigger weights to classify pornographic images correctly. If you want this normalization constant, technically this becomes sum over i of $w^{(i)}$, so then this error would still be between zero and one.

Take away is, if you find that evaluation metric is not giving the correct rank order preference for what is actually better algorithm, then there's a time to think about defining a new evaluation metric. And this is just one possible way that you could define an evaluation metric. The goal of the **evaluation metric is accurately tell you, given two classifiers, which one is better for your application**.

One thing you might notice is that so far we've only talked about how to define a metric to evaluate classifiers. That is, we've defined an evaluation metric that helps us better rank order classifiers when they are performing at varying levels in terms of streaming of porn. And this is actually an example of an **orthogonalization** where I think you should take a machine learning problem and break it into distinct steps.

The overall guideline is if your current metric and data you are evaluating on doesn't correspond to doing well on what you actually care about, then change your metrics and/or your dev/test set to better capture what you need your algorithm to actually do well on. Having an evaluation metric and the dev set allows you to much more quickly make decisions about is Algorithm A or Algorithm B better. It really speeds up how quickly you and your team can iterate. So my recommendation is, even if you can't define the perfect evaluation metric and dev set, just set something up quickly and use that to drive the speed of your team iterating. Here is the summary diagram:

When to change development/test sets and metrics

Example: Cat vs Non-cat

A cat classifier tries to find a great amount of cat images to show to cat loving users. The evaluation metric used is a classification error.

Algorithm	Classification error [%]
A	3%
B	5%

It seems that Algorithm A is better than Algorithm B since there is only a 3% error, however for some reason, Algorithm A is letting through a lot of the pornographic images.

Algorithm B has 5% error thus it classifies fewer images but it doesn't have pornographic images. From a company's point of view, as well as from a user acceptance point of view, Algorithm B is actually a better algorithm. The evaluation metric fails to correctly rank order preferences between algorithms. The evaluation metric or the development set or test set should be changed.

The misclassification error metric can be written as a function as follow:

$$Error: \frac{1}{m_{dev}} \sum_{i=1}^{m_{dev}} \mathcal{L}\{(\hat{y}^{(i)} \neq y^{(i)}\}$$

This function counts up the number of misclassified examples.

The problem with this evaluation metric is that it treats pornographic vs non-pornographic images equally. On way to change this evaluation metric is to add the weight term $w^{(i)}$.

$$w^{(i)} = \begin{cases} 1 & if\ x^{(i)}\ is\ non-pornographic \\ 10 & if\ x^{(i)}\ is\ pornographic \end{cases}$$

The function becomes:

$$Error: \frac{1}{\sum w^{(i)}} \sum_{i=1}^{m_{dev}} w^{(i)} \mathcal{L}\{(\hat{y}^{(i)} \neq y^{(i)}\}$$

Guideline

1. Define correctly an evaluation metric that helps better rank order classifiers
2. Optimize the evaluation metric

Comparing to human level performance

Why human-level performance?

In the last few years, a lot more machine learning teams have been talking about comparing the machine learning systems to human level performance. Why is this? I think there are two main reasons. First is that because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance. Second, it turns out that the workflow of designing and building a machine learning system, the workflow is much more efficient when you're trying to do something that humans can also do. So in those settings, it becomes natural to talk about comparing, or trying to mimic human-level performance. Let's see a couple examples of what this means. I've seen on a lot of machine learning tasks that as you work on a problem over time, so the x-axis, time, this could be many months or even many years over which some team or some research community is working on a problem. Progress tends to be relatively rapid as you approach human level performance. But then after a while, the algorithm surpasses human-level performance and then progress and accuracy actually slows down. And maybe it keeps getting better but after surpassing human level performance it can still get better, but performance, the slope

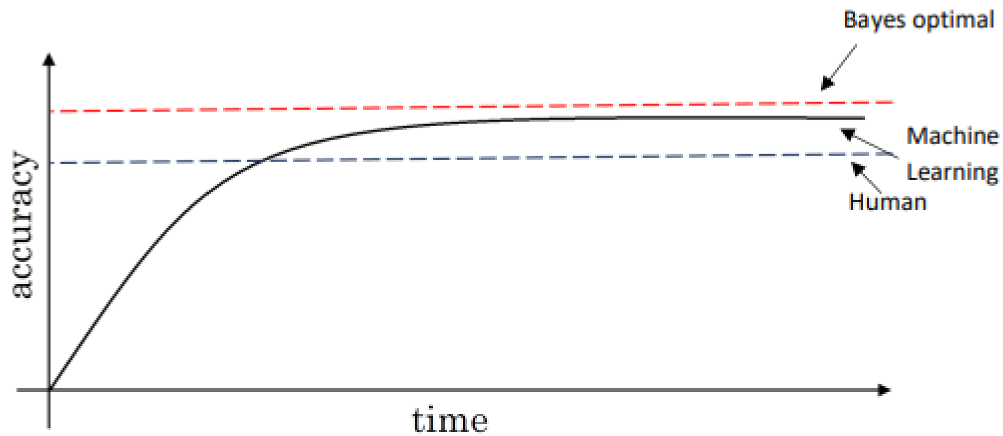
of how rapid the accuracy's going up, often that slows down. And the hope is it achieves some theoretical optimum level of performance.
and over time, as you keep training the algorithm, maybe bigger and bigger models on more and more data, the performance approaches but never surpasses some theoretical limit, which is called the **Bayes optimal erro**r. So **Bayes optimal error, think of this as the best possible error**.

Why human-level performance?

Today, machine learning algorithms can compete with human-level performance since they are more productive and more feasible in a lot of application. Also, the workflow of designing and building a machine learning system, is much more efficient than before.

Moreover, some of the tasks that humans do are close to ''perfection'', which is why machine learning tries to mimic human-level performance.

The graph below shows the performance of humans and machine learning over time.



The

Machine learning progresses slowly when it surpasses human-level performance. One of the reason is that human-level performance can be close to Bayes optimal error, especially for natural perception problem.

Bayes optimal error is defined as the best possible error. In other words, it means that any functions mapping from x to y can't surpass a certain level of accuracy.

Also, when the performance of machine learning is worse than the performance of humans, you can improve it with different tools. They are harder to use once its surpasses human-level performance.

These tools are:

- Get labeled data from humans
- Gain insight from manual error analysis: Why did a person get this right?
- Better analysis of bias/variance.

and that's just the way for any function mapping from x to y to surpass a certain level of accuracy.

So for example, for speech recognition, if x is audio clips, some audio is just so noisy it is impossible to tell what is in the correct transcription. So the perfect error may not be 100%. Or for cat recognition. Maybe some images are so blurry, that it is just impossible for anyone or anything to tell whether or not there's a cat in that picture. So, the perfect level of accuracy may not be 100%. And Bayes optimal error, or Bayesian optimal error, or sometimes Bayes error for short, is the very best theoretical function for mapping from x to y. That can never be surpassed.

So it should be no surprise that this purple line, no matter how many years you work on a problem you can never surpass Bayes error, Bayes optimal error. And it turns out that progress is often quite fast until you surpass human level performance and it sometimes slows down after you surpass human level performance. And I think there are two reasons for that, for why progress often slows down when you surpass human level performance. One reason is that human level performance is for many tasks not that far from Bayes' optimal error. People are very good at looking at images and telling if there's a cat or listening to audio and transcribing

it. So, by the time you surpass human level performance maybe there's not that much head room to still improve.
But the second reason is that so long as your performance is worse than human level performance, then there are actually certain tools you could use to improve performance that are harder to use once you've surpassed human level performance. So here's what I mean. For tasks that humans are quite good at, and this includes looking at pictures and recognizing things, or listening to audio, or reading language, really natural data tasks humans tend to be very good at. For tasks that humans are good at, so long as your machine learning algorithm is still worse than the human, you can get labeled data from humans. That is you can ask people, ask higher humans, to label examples for you so that you can have more data to feed your learning algorithm. Something we'll talk about next week is manual error analysis. But so long as humans are still performing better than any other algorithm, you can ask people to look at examples that your algorithm's getting wrong, and try to gain insight in terms of why a person got it right but the algorithm got it wrong.

Avoidable bias

We talked about how you want your learning algorithm to do well on the training set but sometimes you don't actually want to do too well and knowing what human level performance is, can tell you exactly how well but not too well you want your algorithm to do on the training set. Let me show you what I mean. We have used Cat classification a lot and given a picture, let's say humans have near-perfect accuracy so the human level error is one percent. In that case, if your learning algorithm achieves 8 percent training error and 10 percent dev error, then maybe you wanted to do better on the training set. So the fact that there's a huge gap between how well your algorithm does on your training set versus how humans do shows that your algorithm isn't even fitting the training set well.

So in terms of tools to reduce bias or variance, in this case I would say focus on reducing bias. So you want to do things like train a bigger neural network or run training set longer, just try to do better on the training set. But now let's look at the same training error and dev error and imagine that human level performance was not 1%. So this copy is over but you know in a different application or maybe on a different data set, let's say that human level error is actually 7.5%. Maybe the images in your data set are so blurry that even humans can't tell whether there's a cat in this picture. This example is maybe slightly contrived because humans are actually very good at looking at pictures and telling if there's a cat in it or not. But for the sake of this example, let's say your data sets images are so blurry or so low resolution that even humans get 7.5% error. In this case, even though your training error and dev error are the same as the other example, you see that maybe you're actually doing just fine on the training set. It's doing only a little bit worse than human level performance. And in this second example, you would maybe want to focus on reducing this component, reducing the variance in your learning algorithm. So you might try regularization to try to bring your dev error closer to your training error for example. So in the earlier courses discussion on bias and variance, we were mainly assuming that there were tasks where Bayes error is nearly zero. So to explain what just happened here, for our Cat classification example, think of human level error as a proxy or as a estimate for Bayes error or for Bayes optimal error. And for computer vision tasks, this is a pretty reasonable proxy because humans are actually very good at computer vision and so whatever a human can do is maybe not too far from Bayes error. By definition, human level error is worse than Bayes error because nothing could be better than Bayes error but human level error might not be too far from Bayes error. So the surprising thing we saw here is that depending on what human level error is or really this is really approximately Bayes error or so we assume it to be, but depending on what we think is achievable, with the same training error and dev error in these two cases, we decided to focus on bias reduction tactics or on variance reduction tactics. And what happened is in the example on the left, 8% training error is really high when you think you could get it down to 1% and so bias reduction tactics could help you do that. Whereas in the example on the right, if you think that Bayes error is 7.5% and here we're using human level error as an estimate or as a proxy for Bayes error, but you think that Bayes error is close to seven point five percent then you know there's not that much headroom for reducing your training error further down. You don't really want it to be that much better than 7.5% because you could achieve that only by maybe starting to offer further training so, and instead, there's much more room for improvement in terms of taking this 2% gap and trying to reduce that by using variance reduction techniques such as regularization or maybe getting more training data. So to give these things a couple of names, this is not widely used terminology but I found this useful terminology and a useful way of thinking about it, which is I'm going to call the difference between Bayes error or approximation of Bayes error and the training error to be the avoidable bias. So what you want is maybe keep improving your training performance until you get down to Bayes error but you don't actually want to do better than Bayes error. You can't actually do better than Bayes error unless you're overfitting. And this, the difference between your training area and the dev error, there's a measure still of the variance problem of your algorithm. And the term avoidable bias acknowledges that there's some bias or some minimum level of error that you just cannot get below which is that if Bayes error is 7.5%, you don't actually want to get below that level of error. So rather than saying that

if you're training error is 8%, then the 8% is a measure of bias in this example, you're saying that the avoidable bias is maybe 0.5% or 0.5% is a measure of the avoidable bias whereas 2% is a measure of the variance and so there's much more room in reducing this 2% than in reducing this 0.5%. Whereas in contrast in the example on the left, this 7% is a measure of the avoidable bias, whereas 2% is a measure of how much variance you have. And so in this example on the left, there's much more potential in focusing on reducing that avoidable bias. So in this example, understanding human level error, understanding your estimate of Bayes error really causes you in different scenarios to focus on different tactics, whether bias avoidance tactics or variance avoidance tactics. There's quite a lot more nuance in how you factor in human level performance into how you make decisions in choosing what to focus on.

Avoidable bias

By knowing what the human-level performance is, it is possible to tell when a training set is performing well or not.

Example: Cat vs Non-Cat

	Classification error (%)	
	Scenario A	Scenario B
Humans	1	7.5
Training error	8	8
Development error	10	10

In this case, the human level error as a proxy for Bayes error since humans are good to identify images. If you want to improve the performance of the training set but you can't do better than the Bayes error otherwise the training set is overfitting. By knowing the Bayes error, it is easier to focus on whether bias or variance avoidance tactics will improve the performance of the model.

Scenario A

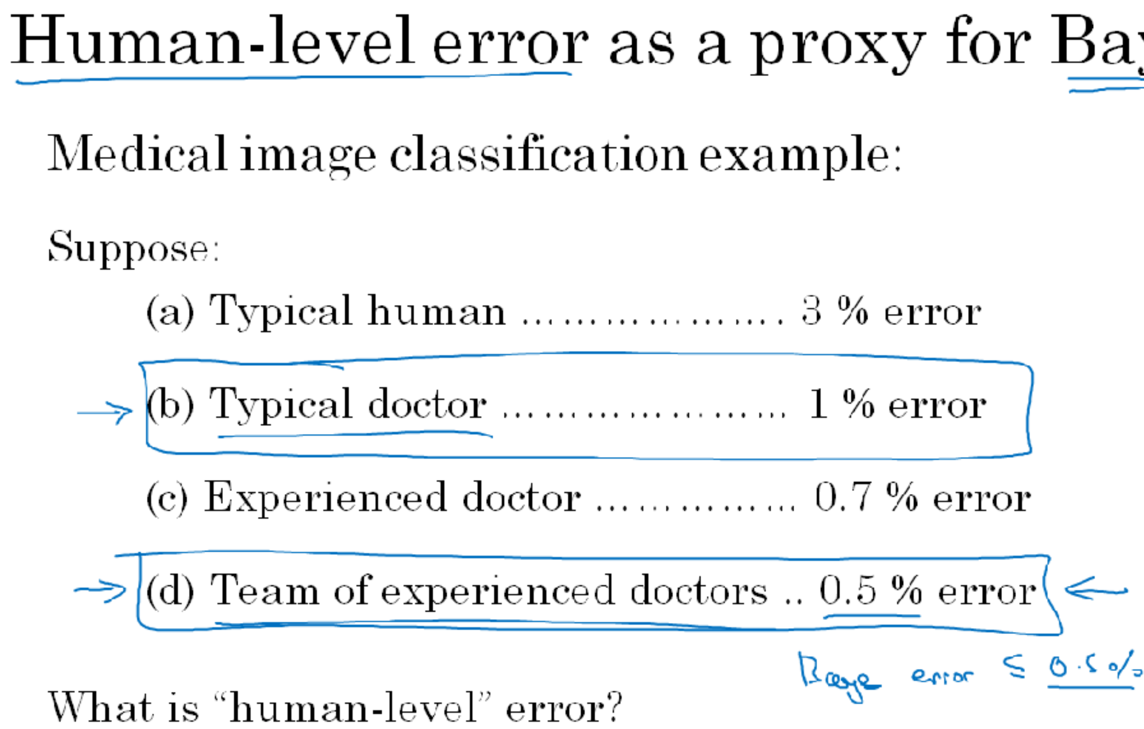
There is a 7% gap between the performance of the training set and the human level error. It means that the algorithm isn't fitting well with the training set since the target is around 1%. To resolve the issue, we use bias reduction technique such as training a bigger neural network or running the training set longer.

Scenario B

The training set is doing good since there is only a 0.5% difference with the human level error. The difference between the training set and the human level error is called avoidable bias. The focus here is to reduce the variance since the difference between the training error and the development error is 2%. To resolve the issue, we use variance reduction technique such as regularization or have a bigger training set.

Understanding human-level performance

The term human-level performance is sometimes used casually in research articles. But let me show you how we can define it a bit more precisely. And in particular, use the definition of the phrase, human-level performance, that is most useful for helping you drive progress in your machine learning project. So remember from our last section that one of the uses of this phrase, human-level error, is that it gives us a way of estimating Bayes error. What is the best possible error any function could, either now or in the future, ever, ever achieve? So bearing that in mind, let's look at a medical image classification example. Let's say that you want to look at a radiology image like this, and make a diagnosis classification decision and suppose that a typical human, untrained human, achieves 3% error on this task. A typical doctor, maybe a typical radiologist doctor, achieves 1% error. An experienced doctor does even better, 0.7% error. And a team of experienced doctors, that is if you get a team of experienced doctors and have them all look at the image and discuss and debate the image, together their consensus opinion achieves 0.5% error. So the question I want to pose to you is, how should you define human-level error? Is human-level error 3%, 1%, 0.7% or 0.5%?



One of the most useful ways to think of human error is as a proxy or an estimate for Bayes error. Which is if you want a proxy or an estimate for Bayes error, then given that a team of experienced doctors discussing and debating can achieve 0.5% error, we know that Bayes error is less than equal to 0.5%. So because some system, team of these doctors can achieve 0.5% error, so by definition, this directly, optimal error has got to be 0.5% or lower. We don't know how much better it is, maybe there's a even larger team of even more experienced doctors who could do even better, so maybe it's even a little bit better than 0.5%. But we know the optimal error cannot be higher than 0.5%. So what I would do in this setting is use 0.5% as our estimate for Bayes error. So I would define human-level performance as 0.5%. Now, for the purpose of publishing a research paper or for the purpose of deploying a system, maybe there's a different definition of human-level error that you can use which is so long as you surpass the performance of a typical doctor. That seems like maybe a very useful result if accomplished, and maybe surpassing a single radiologist, a single doctor's performance might mean the system is good enough to deploy in some context. So maybe the takeaway from this is to be clear about what your purpose is in defining the term human-level error. And if it is to show that you can surpass a single human and therefore argue for deploying your system in some context, maybe this is the appropriate definition. But if your goal is the proxy for Bayes error, then this is the appropriate definition. Let's look at an error analysis example.

Understanding human-level performance

Human-level error gives an estimate of Bayes error.

Example 1: Medical image classification

This is an example of a medical image classification in which the input is a radiology image and the output is a diagnosis classification decision.

	Classification error (%)
Typical human	3.0
Typical doctor	1.0
Experienced doctor	0.7
Team of experienced doctors	0.5

The definition of human-level error depends on the purpose of the analysis, in this case, by definition the Bayes error is lower or equal to 0.5%.

Example 2: Error analysis

<table>
<tr><th rowspan="2"></th><th colspan="3">Classification error (%)</th></tr>
<tr><th>Scenario A</th><th>Scenario B</th><th>Scenario C</th></tr>
<tr><td rowspan="3">Human (proxy for Bayes error)</td><td>1</td><td>1</td><td rowspan="3">0.5</td></tr>
<tr><td>0.7</td><td>0.7</td></tr>
<tr><td>0.5</td><td>0.5</td></tr>
<tr><td>Training error</td><td>5</td><td>1</td><td>0.7</td></tr>
<tr><td>Development error</td><td>6</td><td>5</td><td>0.8</td></tr>
</table>

Scenario A

In this case, the choice of human-level performance doesn't have an impact. The avoidable bias is between 4%-4.5% and the variance is 1%. Therefore, the focus should be on bias reduction technique.

Scenario B

In this case, the choice of human-level performance doesn't have an impact. The avoidable bias is between 0%-0.5% and the variance is 4%. Therefore, the focus should be on variance reduction technique.

Scenario C

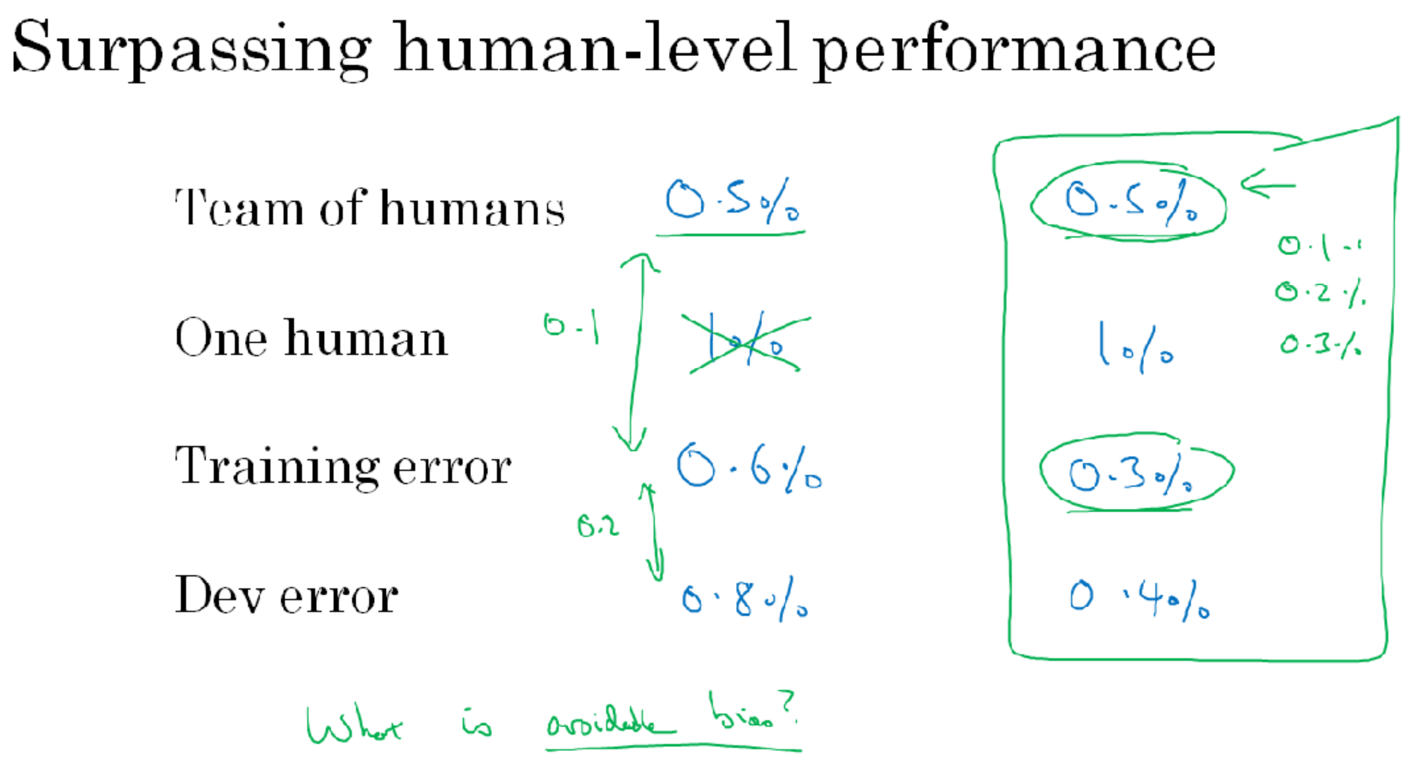
In this case, the estimate for Bayes error has to be 0.5% since you can't go lower than the human-level performance otherwise the training set is overfitting. Also, the avoidable bias is 0.2% and the variance is 0.1%. Therefore, the focus should be on bias reduction technique.

Summary of bias/variance with human-level performance

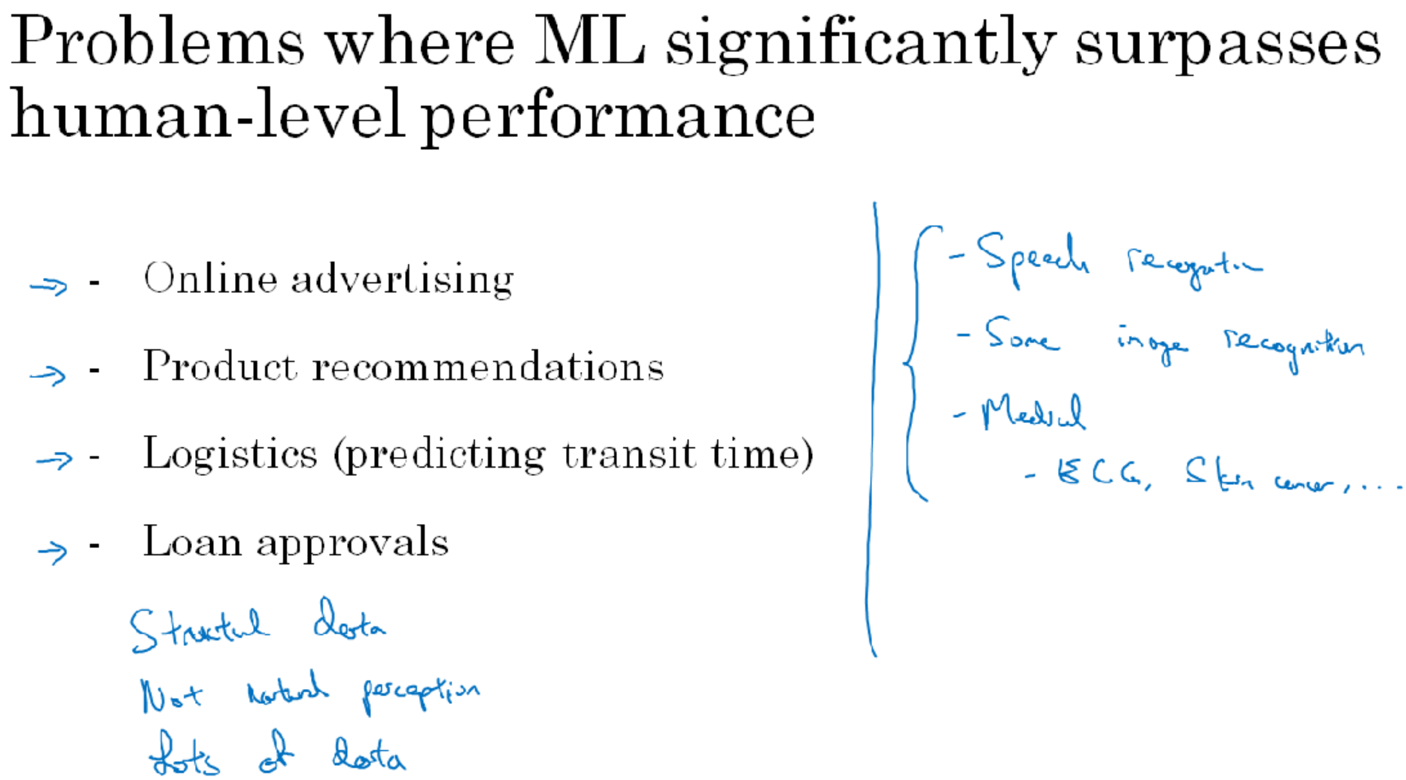
- Human - level error – proxy for Bayes error
- If the difference between human-level error and the training error is bigger than the difference between the training error and the development error. The focus should be on bias reduction technique
- If the difference between training error and the development error is bigger than the difference between the human-level error and the training error. The focus should be on variance reduction technique

Surpassing human-level performance

Lots of teams often find it exciting to surpass human-level performance on the specific recreational classification task. Let's talk over some of the things you see if you try to accomplish this yourself. We've discussed before how machine learning progress gets harder as you approach or even surpass human-level performance. Let's talk over one more example of why that's the case. Let's say you have a problem where a team of humans discussing and debating achieves 0.5% error, a single human 1% error, and you have an algorithm of 0.6% training error and 0.8% dev error. So in this case, what is the **avoidable bias**? So this one is relatively easier to answer, 0.5% is your estimate of base error, so your avoidable bias is, you're not going to use this 1% number as reference, you can use this difference, so maybe you estimate your avoidable bias is at least 0.1% and your variance as 0.2%. So there's maybe more to do to reduce your variance than your avoidable bias perhaps. But now let's take a harder example, let's say, a team of humans and single human performance, the same as before, but your algorithm gets 0.3% training error, and 0.4% dev error. Now, what is the avoidable bias? It's now actually much harder to answer that. Is the fact that your training error, 0.3%, does this mean you've over-fitted by 0.2%, or is base error, actually 0.1%, or maybe is base error 0.2%, or maybe base error is 0.3%? You don't really know, but based on the information given in this example, you actually don't have enough information to tell if you should focus on reducing bias or reducing variance in your algorithm. So that slows down the efficiency where you should make progress. Moreover, if your error is already better than even a team of humans looking at and discussing and debating the right label, for an example, then it's just also harder to rely on human intuition to tell your algorithm what are ways that your algorithm could still improve the performance? So in this example, once you've surpassed this 0.5% threshold, your options, your ways of making progress on the machine learning problem are just less clear. It doesn't mean you can't make progress, you might still be able to make significant progress, but some of the tools you have for pointing you in a clear direction just don't work as well.



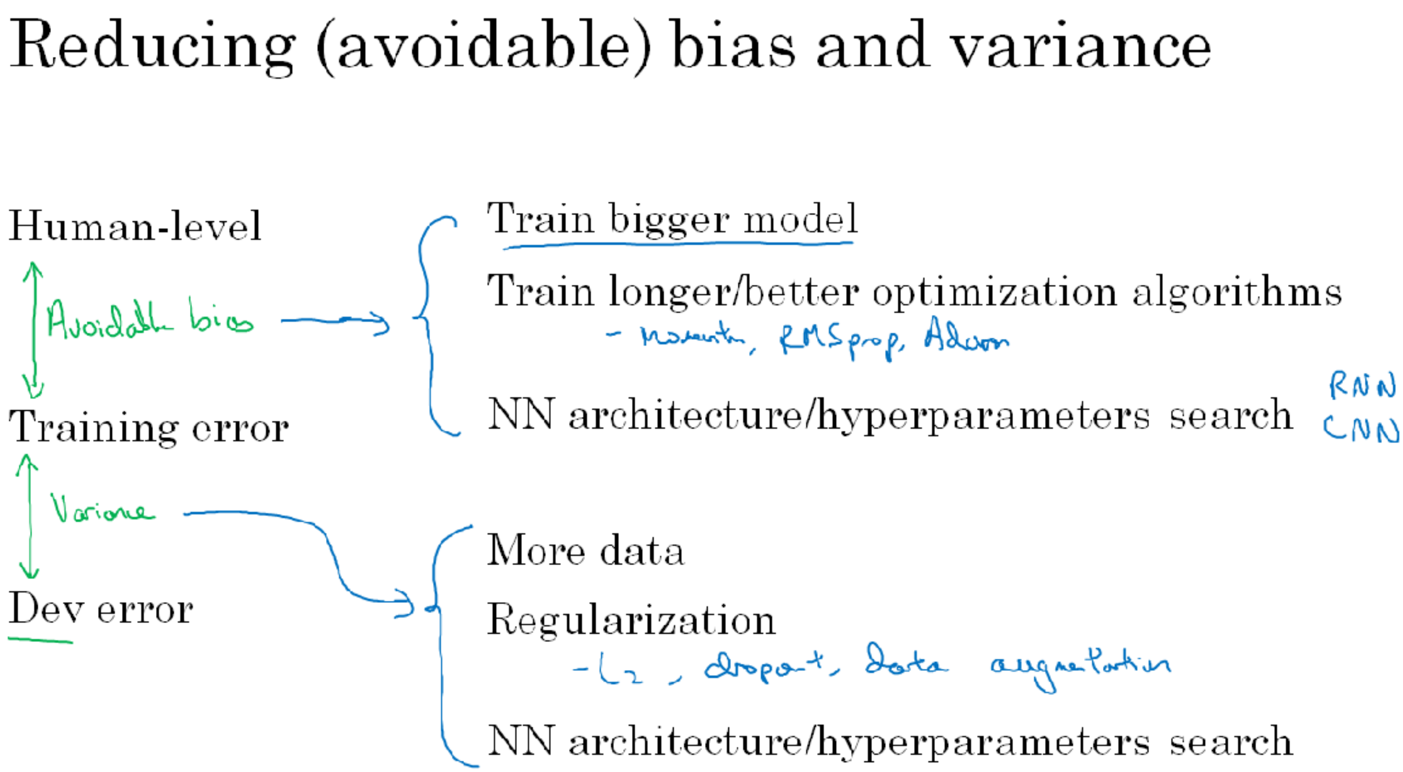
Now, there are many problems where machine learning significantly surpasses human-level performance. For example, I think, online advertising, estimating how likely someone is to click on that. Probably, learning algorithms do that much better today than any human could, or making product recommendations, recommending movies or books to you. I think that web sites today can do that much better than maybe even your closest friends can. All logistics predicting how long will take you to drive from A to B or predicting how long to take a delivery vehicle to drive from A to B, or trying to predict whether someone will repay a loan, and therefore, whether or not you should approve a loan offer. All of these are problems where I think today machine learning far surpasses a single human's performance. Notice something about these four examples. All four of these examples are actually learning from structured data, where you might have a database of what has users clicked on, database of proper support for, databases of how long it takes to get from A to B, database of previous loan applications and their outcomes. And these are not natural perception problems, so these are not computer vision, or speech recognition, or natural language processing task. Humans tend to be very good in natural perception task. So it is possible, but it's just a bit harder for computers to surpass human-level performance on natural perception task.



And finally, all of these are problems where there are teams that have access to huge amounts of data. So for example, the best systems for all four of these applications have probably looked at far more data of that application than any human could possibly look at. And so, that's also made it relatively easy for a computer to surpass human-level performance. Now, the fact that there's so much data that computer could examine, so it can petrifies that's called patterns than even the human mind. Other than these problems, today there are speech recognition systems that can surpass human-level performance. And there are also some computer vision, some image recognition tasks, where computers have surpassed human-level performance. But because humans are very good at this natural perception task, I think it was harder for computers to get there. And then there are some medical tasks, for example, reading ECGs or diagnosing skin cancer, or certain narrow radiology task, where computers are getting really good and maybe surpassing a single human-level performance. And I guess one of the exciting things about recent advances in deep learning is that even fo**r these tasks we can now surpass human-level performance in some cases, but it has been a bit harder because humans tend to be very good at this natural perception task**. So surpassing human-level performance is often not easy, but given enough data there've been lots of deep learning systems have surpassed human-level performance on a single supervisory problem. So that makes sense for an application you're working on.

Improving your model performance

In previous sections we have learned about You **orthogonalization**. How to set up your dev and test sets, human level performance as a proxy for Bayes's error and how to estimate your avoidable bias and variance. Let's pull it all together into a set of guidelines for how to improve the performance of your learning algorithm. So, I think getting a supervised learning algorithm to work well means fundamentally hoping or assuming that you can do two things. **First is that you can fit the training set pretty well and you can think of this as roughly saying that you can achieve low avoidable bias**. And the second **thing you're assuming can do well is that doing well in the training set generalizes pretty well to the dev set or the test set and this is sort of saying that variance is not too bad**.



How much better do you think you should be trying to do on your training set and then look at the difference between your dev error and your training error as an estimate. So, it's how much of a variance problem you have. In other words, how much harder you should be working to make your performance generalize from the training set to the desk set, that it wasn't trained on explicitly?

So to whatever extent you want to try to reduce avoidable bias, I would try to apply tactics like train a bigger model. So, you can just do better on your training sets or train longer. Use a better optimization algorithm such as momentum or RMS prop, Adam. One of the thing you could try is to just find a better new NN architecture or hyperparameters and this could include everything from changing the activation functions or changing the number of layers or hidden do this or you can try other NN models architectures, such as the recurrent neural network and convolution neural networks. See below diagram with summary.

Improving your model performance

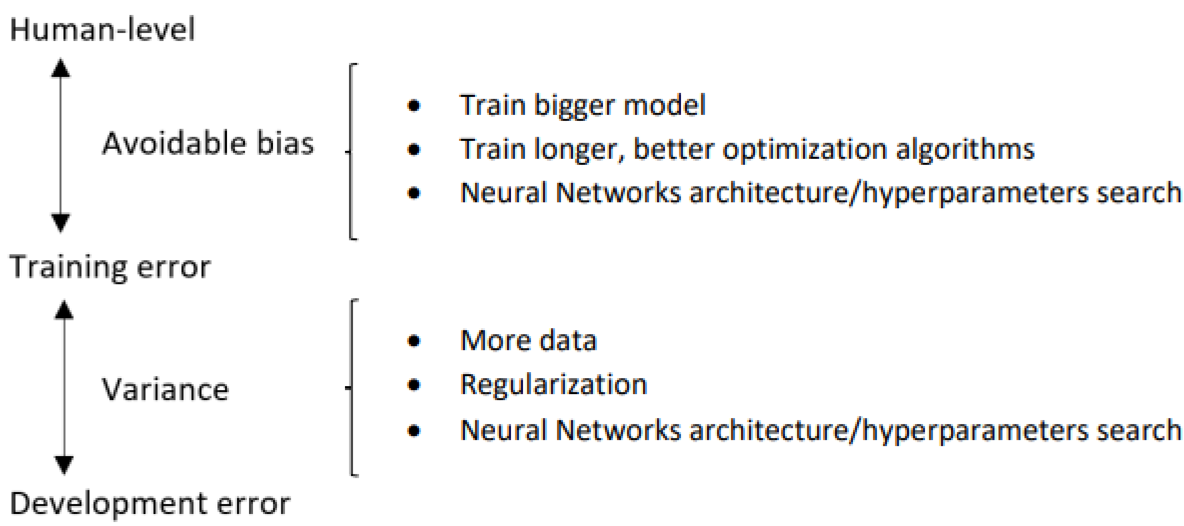
The two fundamental assumptions of supervised learning

There are 2 fundamental assumptions of supervised learning. The first one is to have a low avoidable bias which means that the training set fits well. The second one is to have a low or acceptable variance which means that the training set performance generalizes well to the development set and test set.

If the difference between human-level error and the training error is bigger than the difference between the training error and the development error, the focus should be on bias reduction technique which are training a bigger model, training longer or change the neural networks architecture or try various hyperparameters search.

If the difference between training error and the development error is bigger than the difference between the human-level error and the training error, the focus should be on variance reduction technique which are bigger data set, regularization or change the neural networks architecture or try various hyperparameters search.

Summary



Week 2: ML Strategy (2)

Learning Objectives

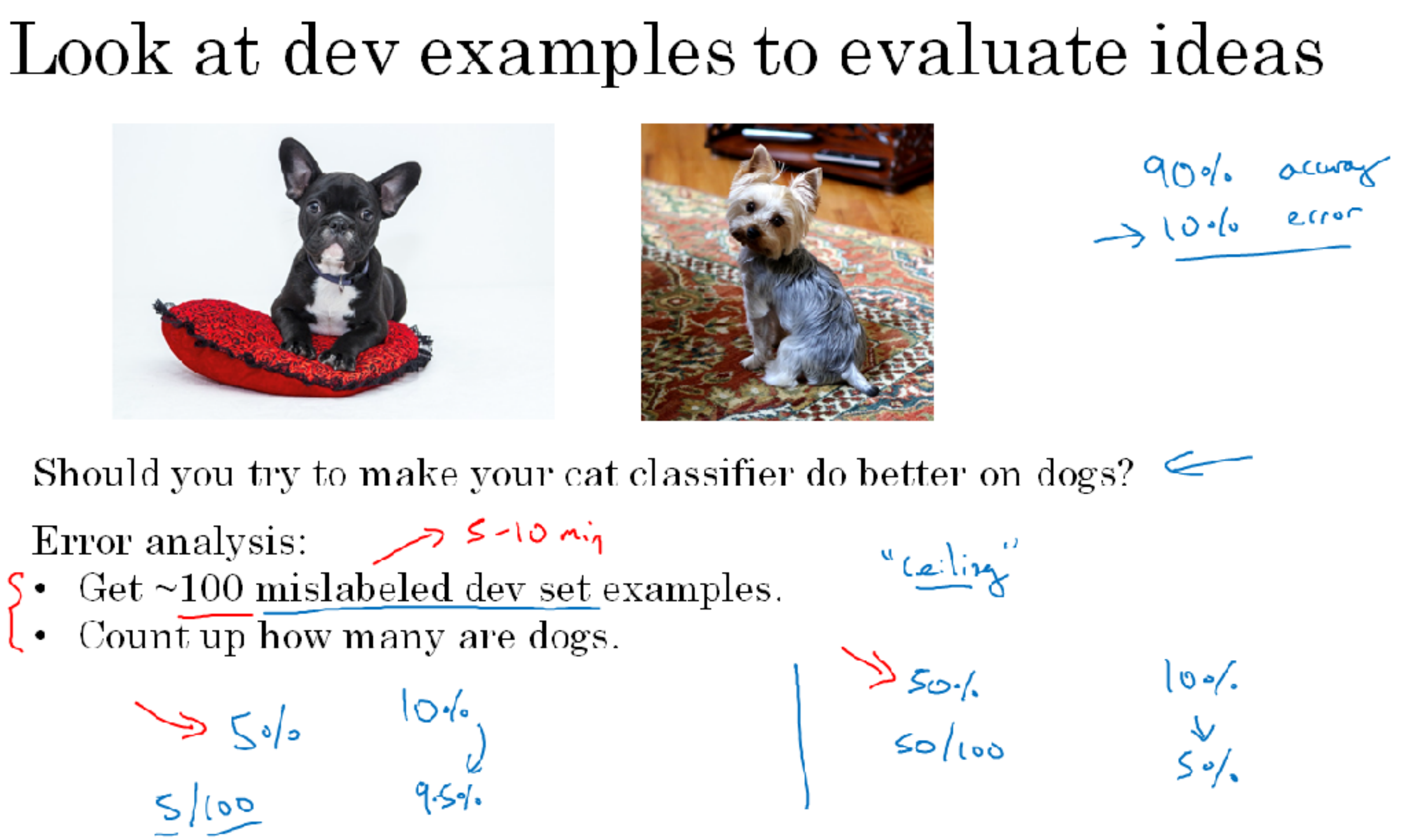
- Understand what multi-task learning and transfer learning are
- Recognize bias, variance and data-mismatch by looking at the performances of your algorithm on train/dev/test sets

Error Analysis

Carrying out error analysis

If you're trying to get a learning algorithm to do a task that humans can do and if your learning algorithm is not yet at the performance of a human. Then manually examining mistakes that your

algorithm is making, can give you insights into what to do next. This process is called **error analysis**. Let's start with an example. Let's say you're working on your cat classifier, and you've achieved 90% accuracy, or equivalently 10% error, on your dev set. And let's say this is much worse than you're hoping to do. Maybe one of your teammates looks at some of the examples that the algorithm is misclassifying, and notices that it is miscategorizing some dogs as cats. And if you look at these two dogs, maybe they look a little bit like a cat, at least at first glance. So maybe your teammate comes to you with a proposal for how to make the algorithm do better, specifically on dogs, right? You can imagine building a focus effort, maybe to collect more dog pictures, or maybe to design features specific to dogs, or something. In order to make your cat classifier do better on dogs, so it stops misrecognizing these dogs as cats. So the question is, **should you go ahead and start a project focus on the dog problem**?
There could be several months of works you could do in order to make your algorithm make few mistakes on dog pictures. So is that worth your effort? Well, rather than spending a few months doing this, only to risk finding out at the end that it wasn't that helpful. Here's an **error analysis procedure that can let you very quickly tell whether or not this could be worth your effort**. Here's what I recommend you do. First, get about, say 100 mislabeled dev set examples, then examine them manually. Just count them up one at a time, to see how many of these mislabeled examples in your dev set are actually pictures of dogs. Now, suppose that it turns out that 5% of your 100 mislabeled dev set examples are pictures of dogs. So, that is, if 5 out of 100 of these mislabeled dev set examples are dogs, what this means is that of the 100 examples. Of a typical set of 100 examples you're getting wrong, even if you completely solve the dog problem, you only get 5 out of 100 more correct. Or in other words, if only 5% of your errors are dog pictures, then the best you could easily hope to do, if you spend a lot of time on the dog problem. Is that your error might go down from 10% error, down to 9.5% error, right? So this a 5% relative decrease in error, from 10% down to 9.5%. And so you might reasonably decide that this is not the best use of your time. Or maybe it is, but at least this gives you a ceiling, a upper bound on how much you could improve performance by working on the dog problem. In machine learning, sometimes we call this the ceiling on performance. Which just means, what's in the best case? How well could working on the dog problem help you?



But now, suppose something else happens. Suppose that we look at your 100 mislabeled dev set examples, you find that 50 of them are actually dog images. So 50% of them are dog pictures. Now you could be much more optimistic about spending time on the dog problem. In this case, if you actually solve the dog problem, your error would go down from this 10%, down to potentially 5% error. And you might decide that halving your error could be worth a lot of effort. Focus on reducing the problem of mislabeled dogs. I know that in machine learning, sometimes we speak disparagingly of hand engineering things, or using too much value insight. But if you're building applied systems, then this simple counting procedure, error analysis, can save you a lot of time. In terms of deciding what's the most important, or what's the most promising direction to focus on. In fact, if you're looking at 100 mislabeled dev set examples, maybe this is a 5 to 10 minute effort. To manually go through 100 examples, and manually count up how many of them are dogs. And depending on the outcome, whether there's more like 5%, or 50%, or something else. This, in just 5 to 10 minutes, gives you an estimate of how worthwhile this direction is and could help you make a much better decision, whether or not to spend the next few months focused on trying to find solutions to solve the problem of mislabeled dogs. In this section, we'll describe using error analysis to evaluate whether or not a single idea, dogs in this case, is worth working on. Sometimes you can also evaluate multiple ideas in parallel doing error analysis.

For example, let's say you have several ideas in improving your cat detector. Maybe you can improve performance on dogs? Or maybe you notice that sometimes, what are called great cats, such as lions, panthers, cheetahs, and so on. That they are being recognized as small cats, or house cats. So you could maybe find a way to work on that. Or maybe you find that some of your images are blurry, and it would be nice if you could design something that just works better on blurry images and maybe you have some ideas on how to do that. So if carrying out error analysis to evaluate these three ideas, what I would do is create a table like this and I usually do this in a spreadsheet, but using an ordinary text file will also be okay and on the left side, this goes through the set of images you plan to look at manually. So this maybe goes from 1 to 100, if you look at 100 pictures. And the columns of this table, of the spreadsheet, will correspond to the ideas you're evaluating. So the dog problem, the problem of great cats, and blurry images. And I usually also leave space in the spreadsheet to write comments. So remember, during error analysis, you're just looking at dev set examples that your algorithm has misrecognized. So if you find that the first misrecognized image is a picture of a dog, then I'd put a check mark there. And to help myself remember these images, sometimes I'll make a note in the comments. So maybe that was a pit bull picture. If the second picture was blurry, then make a note there. If the third one was a lion, on a rainy day, in the zoo that was misrecognized. Then that's a great cat, and the blurry data. Make a note in the comment section, rainy day at zoo, and it was the rain that made it blurry, and so on. Then finally, having gone through some set of images, I would count up what percentage of these algorithms. Or what percentage of each of these error categories were attributed to the dog, or great cat, blurry categories. So maybe 8% of these images you examine turn out be dogs, and maybe 43% great cats, and 61% were blurry. So this just means going down each column, and counting up what percentage of images have a check mark in that column. As you're part way through this process, sometimes you notice other categories of mistakes. So, for example, you might find that Instagram style filter, those fancy image filters, are also messing up your classifier. In that case, it's actually okay, part way through the process, to add another column like that. For the multi-colored filters, the Instagram filters, and the Snapchat filters. And then go through and count up those as well, and figure out what percentage comes from that new error category.

Evaluate multiple ideas in parallel

Ideas for cat detection:

- Fix pictures of dogs being recognized as cats
- Fix great cats (lions, panthers, etc..) being misrecognized
- Improve performance on blurry images

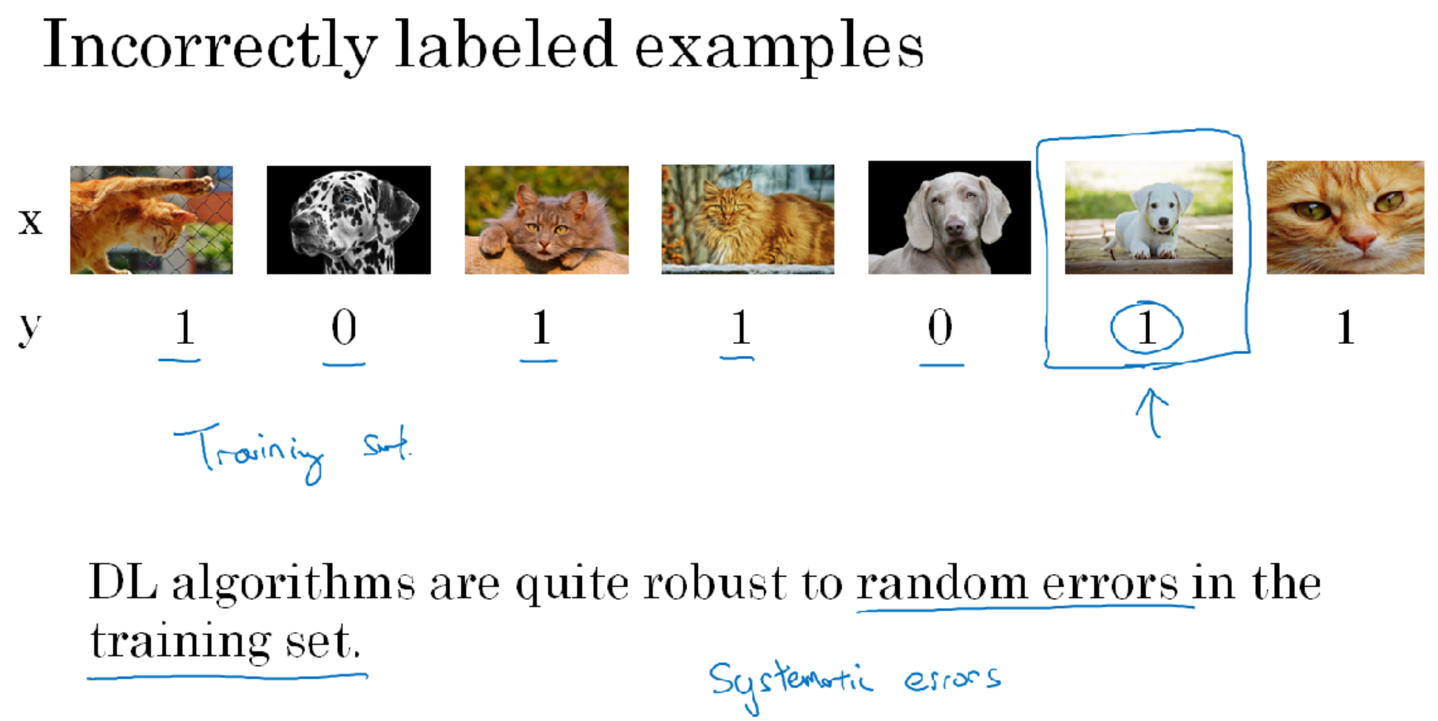
Image	Dog	Great Cats	Blurry	Instagram	Comments
1	✓			✓	Pitbull
2			✓	✓	
3		✓	✓		Rainy day at zoo
⋮	⋮	⋮	⋮		
% of total	8%	43%	61%	12%	

The conclusion of this process gives you an estimate of how worthwhile it might be to work on each of these different categories of errors. For example, clearly in the example (see diagram above), a lot of the mistakes we made on blurry images, and quite a lot on were made on great cat images. And so the outcome of this analysis is not that you must work on blurry images. This doesn't give you a rigid mathematical formula that tells you what to do, but it gives you a sense of the best options to pursue. It also tells you, for example, that no matter how much better you do on dog images, or on Instagram images. You at most improve performance by maybe 8%, or 12%, in these examples. Whereas you can to better on great cat images, or blurry images, the potential improvement. Now there's a ceiling in terms of how much you could improve performance, is much higher. So depending on how many ideas you have for improving performance on great cats, on blurry images. Maybe you could pick one of the two, or if you have enough personnel on your team, maybe you can have two different teams. Have one work on improving errors on great cats, and a different team work on improving errors on blurry images. But this quick counting procedure, which you can often do in, at most, small numbers of hours. Can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.

So to summarize, to carry out error analysis, you should find a set of mislabeled examples, either in your dev set, or in your development set. And look at the mislabeled examples for false positives and false negatives. And just count up the number of errors that fall into various different categories. During this process, you might be inspired to generate new categories of errors, like we saw. If you're looking through the examples and you say gee, there are a lot of Instagram filters, or Snapchat filters, they're also messing up my classifier. You can create new categories during that process. But by counting up the fraction of examples that are mislabeled in different ways, often this will help you prioritize. Or give you inspiration for new directions to go in. Now as you're doing error analysis, sometimes you notice that some of your examples in your dev sets are mislabeled, we'll discuss that in next section.

Cleaning up incorrectly labeled data

The data for your supervised learning problem comprises input X and output labels Y. What if you going through your data and you find that some of these output labels Y are incorrect, you have data which is incorrectly labeled? Is it worth your while to go in to fix up some of these labels? Let's take a look. In the cat classification problem, Y equals one for cats and zero for non cats.



So if you find that your data has some incorrectly labeled examples, what should you do? Well, first, let's consider the training set. It turns out that deep learning algorithms are quite robust to random errors in the training set. So as long as your errors or your incorrectly labeled examples, so as long as those errors are not too far from random, maybe sometimes the labeler just wasn't paying attention or they accidentally, randomly hit the wrong key on the keyboard. **If the errors are reasonably random, then it's probably okay to just leave the errors as they are and not spend too much time fixing them**. There's certainly no harm to going into your training set and be examining the labels and fixing them. Sometimes that is worth doing but your effort might be okay even if you don't. So as long as the total data set size is big enough and the actual percentage of errors is maybe not too high. So I see a lot of machine learning algorithms that trained even when we know that there are few X mistakes in the training set labels and usually works okay. There is one caveat to this which is that deep learning algorithms are robust to random errors. They are less robust to systematic errors. So for example, if your labeler consistently labels white dogs as cats, then that is a problem because your classifier will learn to classify all white colored dogs as cats. But random errors or near random errors are usually not too bad for most deep learning algorithms. Now, this discussion has focused on what to do about incorrectly labeled examples in your training set. How about incorrectly labeled examples in your dev set or test set? If you're worried about the impact of incorrectly labeled examples on your dev set or test set, what they recommend you do is during error analysis to add one extra column so that you can also count up the number of examples where the label Y was incorrect. So for example, maybe when you count up the impact on a 100 mislabeled dev set examples, so you're going to find a 100 examples where your classifier's output disagrees with the label in your dev set. And sometimes for a few of those examples, your classifier disagrees with the label because the label was wrong, rather than because your classifier was wrong.

Error analysis

Image	Dog	Great Cat	Blurry	Incorrectly labeled	Comments
...					
98				✓	Labeler missed cat in background
99		✓			
100				✓	Drawing of a cat; Not a real cat.
% of total	8%	43%	61%	6%	

Overall dev set error 10% 2%

Errors due incorrect labels 0.6% 0.6%

Errors due to other causes 9.4% 1.4%

2.1% 1.9%

Goal of dev set is to help you select between two classifiers A & B.

So maybe in this example, you find that the labeler missed a cat in the background. So put the check mark there to signify that example 98 had an incorrect label. And maybe for this one, the picture is actually a picture of a drawing of a cat rather than a real cat. Maybe you want the labeler to have labeled that Y equals zero rather than Y equals one. And so put another check mark there. And just as you count up the percent of errors due to other categories like we saw in the previous video, you'd also count up the fraction of percentage of errors due to incorrect labels. Where the Y value in your dev set was wrong and that accounted for why your learning algorithm made a prediction that differed from what the label on your data says. So the question now is, is it worthwhile going in to try to fix up this 6% of incorrectly labeled examples. My advice is, if it makes a significant difference to your **ability to evaluate algorithms** on your dev set, then go ahead and spend the time to fix incorrect labels. But if it doesn't make a significant difference to your ability to use the dev set to evaluate cost buyers, then it might not be the best use of your time.

Correcting incorrect dev/test set examples

- Apply same process to your dev and test sets to make sure they continue to come from the same distribution
- Consider examining examples your algorithm got 98% right as well as ones it got wrong. 2%
- Train and dev/test data may now come from slightly different distributions.

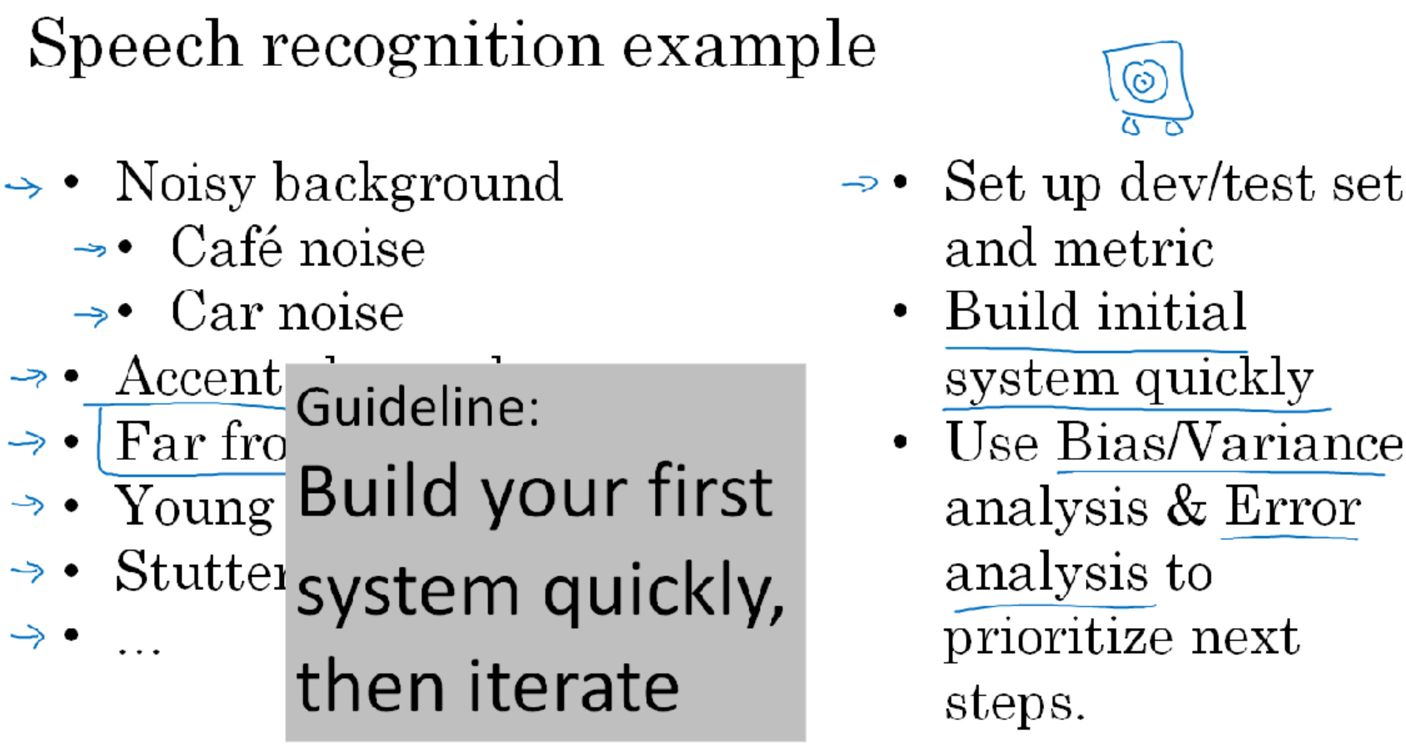
Let me show you an example that illustrates what I mean by this. So, three numbers I recommend you look at to try to decide if it's worth going in and reducing the number of mislabeled examples are the following. I recommend you look at the overall dev set error. And so in the example we had from the previous video, we said that maybe our system has 90% overall accuracy. So 10% error. Then you should look at the number of errors or the percentage of errors that are due to incorrect labels. So it looks like in this case, 6% of the errors are due to incorrect labels. So 6% of 10% is 0.6%. And then you should look at errors due to all other causes. So if you made 10% error on your dev set and 0.6% of those are because the labels is wrong, then the remainder, 9.4% of them, are due to other causes such as misrecognizing dogs being cats, great cats and their images. So in this case, I would say there's 9.4% worth of error that you could focus on fixing, whereas the errors due to incorrect labels is a relatively small fraction of the overall set of errors. So by all means, go in and fix these incorrect labels if you want but it's maybe not the most important thing to do right now. Now, let's take another example. Suppose you've made a lot more progress on your learning problem. So instead of 10% error, let's say you brought the errors down to 2%, but still 0.6% of your overall errors are due to incorrect labels. So now, if you want to examine a set of mislabeled dev set images, set that comes from just 2% of dev set data you're mislabeling, then a very large fraction of them, 0.6 divided by 2%, so that is actually 30% rather than 6% of your labels. Your incorrect examples are actually due to incorrectly label examples. And so errors due to other causes are now 1.4%. When such a high fraction of your mistakes as measured on your dev set due to incorrect labels, then it maybe seems much more worthwhile to fix up the incorrect labels in your dev set. And if you remember the goal of the dev set, the main purpose of the dev set is, you want to really use it to help you select between two classifiers A and B. So you're trying out two classifiers A and B, and one has 2.1% error and the other has 1.9% error on your dev set. But you don't trust your dev set anymore to be correctly telling you whether this classifier is actually better than this because your 0.6% of these mistakes are due to incorrect labels. Then there's a good reason to go in and fix the incorrect labels in your dev set. Because in this example on the right is just having a very large impact on the overall assessment of the errors of the algorithm, whereas example on the left, the percentage impact is having on your algorithm is still smaller. Now, if you decide to go into a dev set and manually re-examine the labels and try to fix up some of the labels, here are a few additional guidelines or principles to consider. First, I would encourage you to apply whatever process you apply to both your dev and test sets at the same time. We've talk previously about why you want to dev and test sets to come from the same distribution. The dev set is tagging you into target and when you hit it, you want that to generalize to the test set. So your team really works more efficiently to dev and test sets come from the same distribution. So if you're going in to fix something on the dev set, I would apply the same process to the test set to make sure that they continue to come from the same distribution. So we hire someone to examine the labels more carefully.

Do that for both your dev and test sets. Second, I would urge you to consider examining examples your algorithm got right as well as ones it got wrong. It is easy to look at the examples your algorithm got wrong and just see if any of those need to be fixed. But it's possible that there are some examples that you haven't got right, that should also be fixed. And if you only fix ones that your algorithms got wrong, you end up with more bias estimates of the error of your algorithm. It gives your algorithm a little bit of an unfair advantage. We just try to double check what it got wrong but you don't also double check what it got right because it might have gotten something right, that it was just lucky on fixing the label would cause it to go from being right to being wrong, on that example. The second bullet isn't always easy to do, so it's not always done. The reason it's not always done is because if you classifier's very accurate, then it's getting fewer things wrong than right. So if your classifier has 98% accuracy, then it's getting 2% of things wrong and 98% of things right. So it's much easier to examine and validate the labels on 2% of the data and it takes much longer to validate labels on 98% of the data, so this isn't always done. That's just something to consider.

Finally, if you go into a dev and test data to correct some of the labels there, you may or may not decide to go and apply the same process for the training set. Remember we said that at this other section that it's actually **less important to correct the labels in your training set**. And it's quite possible you decide to just **correct the labels in your dev and test set** which are also often smaller than a training set and you might not invest all that extra effort needed to correct the labels in a much larger training set. This is actually okay. Learning algorithms are quite robust to that. It's super important that your dev and test sets come from the same distribution. But if your training set comes from a slightly different distribution, often that's a pretty reasonable thing to do.So couple of advice, First, deep learning researchers sometimes like to say things like, "I just fed the data to the algorithm. I trained in and it worked." There is a lot of truth to that in the deep learning error. There is more of feeding data in algorithm and just training it and doing less hand engineering and using less human insight. But I think that in building practical systems, often there's also more manual error analysis and more human insight that goes into the systems than sometimes deep learning researchers like to acknowledge. Second is that somehow I've seen some engineers and researchers be reluctant to manually look at the examples. Maybe it's not the most interesting thing to do, to sit down and look at a 100 or a couple hundred examples to counter the number of errors. But this is something that I so do myself. When I'm leading a machine learning team and I want to understand what mistakes it is making, I would actually go in and look at the data myself and try to counter the fraction of errors. And I think that because these minutes or maybe a small number of hours of counting data can really help you prioritize where to go next. I find this a very good use of your time and I urge you to consider doing it if those machines are in your system and you're trying to decide what ideas or what directions to prioritize things.

Build your first system, quickly, then iterate

If you're working on a brand new machine learning application, one of the piece of advice I often give people is that, I think you should build your first system quickly and then iterate. Let me show you what I mean. I've worked on speech recognition for many years. And if you're thinking of building a new speech recognition system, there's actually a lot of directions you could go and a lot of things you could prioritize. For example, there are specific techniques for making speech recognition systems more robust to noisy background.



And noisy background could mean cafe noise, like a lot of people talking in the background or car noise, the sounds of cars and highways or other types of noise. There are ways to make a speech recognition system more robust to accented speech. There are specific problems associated with speakers that are far from the microphone, this is called far-field speech recognition. Young children speech poses special challenges, both in terms of how they pronounce individual words as well as their choice of words and the vocabulary they tend to use. And if sometimes the speaker stutters or if they use nonsensical phrases like oh, ah, um, there are different choices and different techniques for making the transcript that you output, still read more fluently. So, there are these and many other things you could do to improve a speech recognition system. And more generally, for almost any machine learning application, there could be 50 different directions you could go in and each of these directions is reasonable and would make your system better. But the challenge is, how do you pick which of these to focus on. And even though I've worked in speech recognition for many years, if I'm building a new system for a new application domain, I would still find it maybe a little bit difficult to pick without spending some time thinking about the problem. So what we recommend you do, if you're starting on building a brand new machine learning application, **is to build your first system quickly and then iterate**. What I mean by that is I recommend that **you first quickly set up a dev/test set and metric. So this is really deciding where to place your target. And if you get it wrong, you can always move it later, but just set up a target somewhere. And then I recommend you build an initial machine learning system quickly. Find the training set, train it and see**. Start to see and understand how well you're doing against your dev/test set and your values and metric. **When you build your initial system, you then be able to use bias/variance analysis which we talked about earlier as well as error analysis** which we talked about just in the last several sections, to prioritize the next steps. In particular, if error analysis causes you to realize that a lot of the errors are from the speaker being very far from the microphone, which causes special challenges to speech recognition, then that will give you a good reason to focus on techniques to address this called **far-field speech recognition** which basically means handling when the speaker is very far from the microphone. Of all the value of building this initial system, it can be a quick and dirty implementation, you know, don't overthink it, but all the value of the initial system is having some learned system, having some trained system allows you to localize bias/variance, to try to prioritize what to do next, allows you to do error analysis, look at some mistakes, to figure out all the different directions you can go in, which ones are actually the most worthwhile.

So to recap, what I recommend you do is build your first system quickly, then iterate. This advice applies less strongly if you're working on an application area in which you have significant prior experience. It also implies to build less strongly if there's a significant body of academic literature that you can draw on for pretty much the exact same problem you're building. So, for example, there's a large academic literature on face recognition. And if you're trying to build a face recognizer, it might be okay to build a more complex system from the get-go by building on this large body of academic literature. But if you are tackling a new problem for the first time, then I would encourage you to really not overthink or not make your first system too complicated. Well, just build something quick and dirty and then use that to help you prioritize how to improve your system. So I've seen a lot of machine learning projects and I've seen some teams over-think the solution and build something too complicated. I've also seen some teams under-think and then build something maybe too simple. Well on average, I've seen a lot more teams over-think and build something too complicated.

Build system quickly, then iterate

Depending on the area of application, the guideline below will help you prioritize when you build your system.

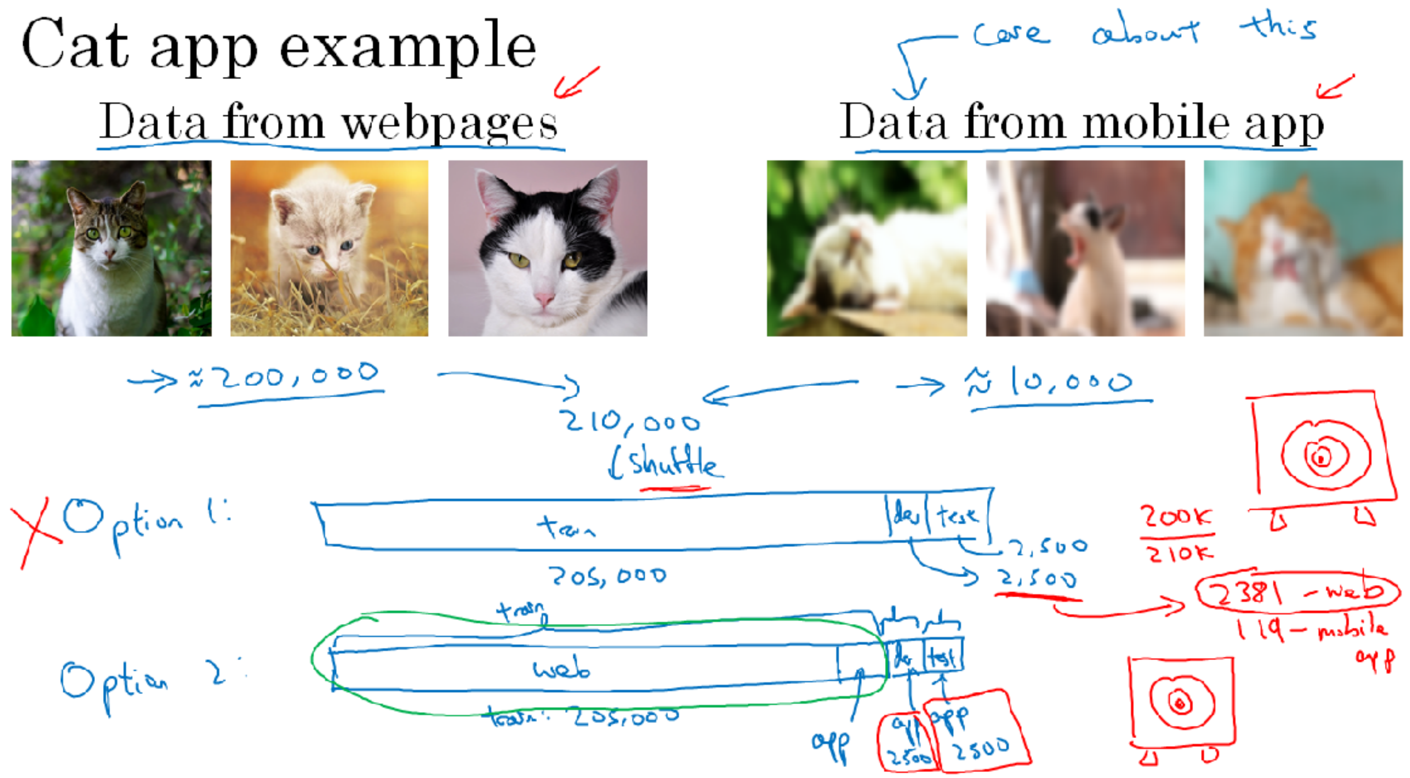
Guideline

1. Set up development/ test set and metrics
   - Set up a target
2. Build an initial system quickly
   - Train training set quickly: Fit the parameters
   - Development set: Tune the parameters
   - Test set: Assess the performance
3. Use Bias/Variance analysis & Error analysis to prioritize next steps

Mismatched training and dev/test set

Training and testing on different distributions

Deep learning algorithms have a huge hunger for training data. They just often work best when you can find enough label training data to put into the training set. This has resulted in many teams sometimes taking whatever data you can find and just shoving it into the training set just to get it more training data. Even if some of this data, or even maybe a lot of this data, doesn't come from the same distribution as your dev and test data. So in a deep learning era, more and more teams are now training on data that comes from a different distribution than your dev and test sets. And there's some subtleties and some best practices for dealing with when you're training and test distributions differ from each other. Let's take a look. Let's say that you're building a mobile app where users will upload pictures taken from their cell phones, and you want to recognize whether the pictures that your users upload from the mobile app is a cat or not. So you can now get two sources of data. One which is the distribution of data you really care about, this data from a mobile app like that on the right, which tends to be less professionally shot, less well framed, maybe even blurrier because it's shot by amateur users. The other source of data you can get is you can crawl the web and just download a lot of, for the sake of this example, let's say you can download a lot of very professionally framed, high resolution, professionally taken images of cats. And let's say you don't have a lot of users yet for your mobile app. So maybe you've gotten 10,000 pictures uploaded from the mobile app. But by crawling the web you can download huge numbers of cat pictures, and maybe you have 200,000 pictures of cats downloaded off the Internet.



So what you really care about is that your final system does well on the mobile app distribution of images, right? Because in the end, your users will be uploading pictures like those on the right and you need your classifier to do well on that. But you now have a bit of a dilemma because you have a relatively small dataset, just 10,000 examples drawn from that distribution. And you have a much bigger dataset that's drawn from a different distribution. There's a different appearance of image than the one you actually want. So you don't want to use just those 10,000 images because it ends up giving you a relatively small training set and using those 200,000 images seems helpful, but the dilemma is this 200,000 images isn't from exactly the distribution you want. So what can you do? Well, here's one option. One thing you can do is put both of these data sets together so you now have 210,000 images. And you can then take the 210,000 images and randomly shuffle them into a train, dev, and test set. And let's say for the sake of argument that you've decided that your dev and test sets will be 2,500 examples each. So your training set will be 205,000 examples.

Now so set up your data this way has some advantages but also disadvantages. The advantage is that now you're training, dev and test sets will all come from the same distribution, so that makes it easier to manage. But the disadvantage, and this is a huge disadvantage, is that if you look at your dev set, of these 2,500 examples, a lot of it will come from the web page distribution of images, rather than what you actually care about, which is the mobile app distribution of images. So it turns out that of your total amount of data, 200,000 or 200k, out of 210,000 or 210k, that comes from web pages. So all of these 2,500 examples on expectation, I think 2,381 of them will come from web pages. This is on expectation, the exact number will vary around depending on how the random shuttle operation went. But on average, only 119 will come from mobile app uploads.

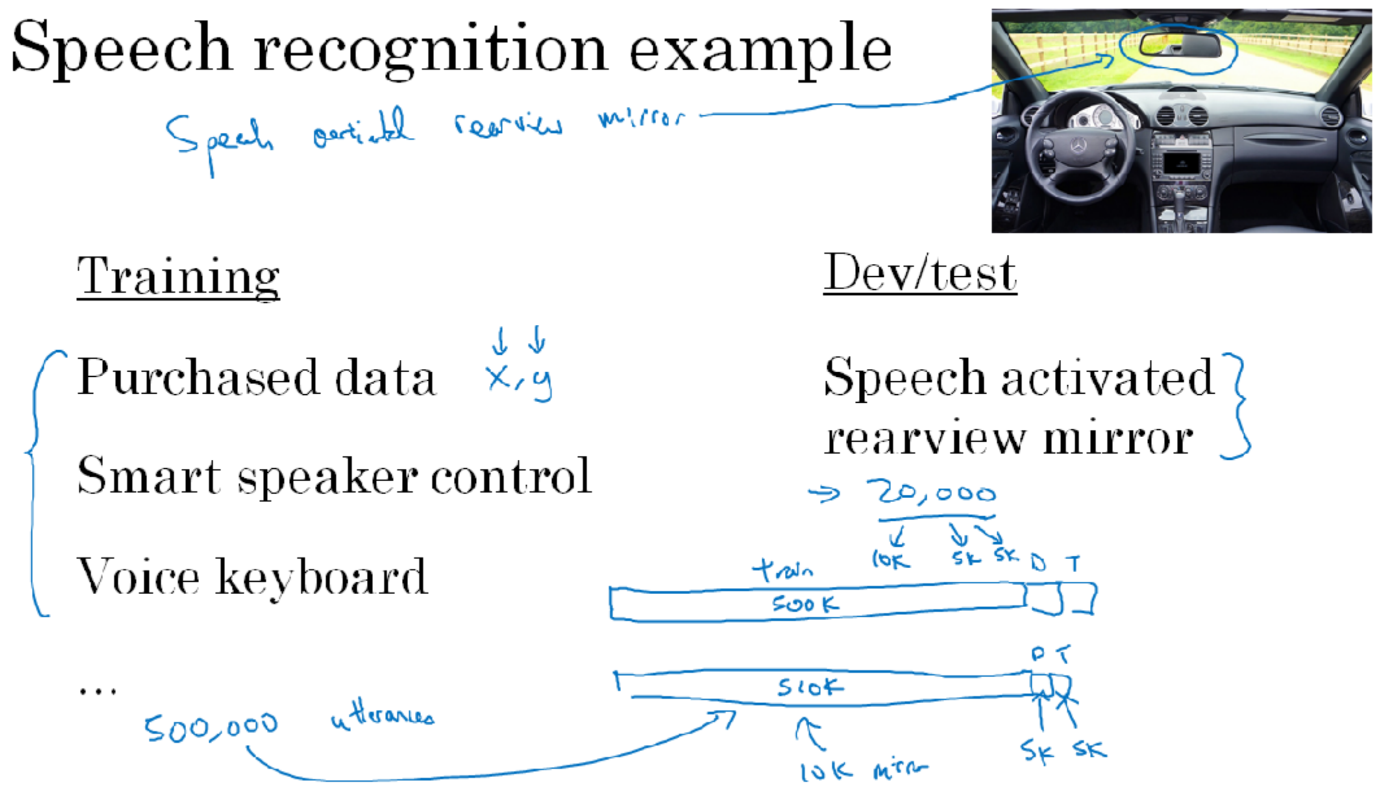
So remember that setting up your dev set is telling your team where to aim the target and the way you're aiming your target, you're saying spend most of the time optimizing for the web page distribution of images, which is really not what you want.

So I would recommend against option one, because this is setting up the dev set to tell your team to optimize for a different distribution of data than what you actually care about.

So instead of doing this, I would recommend that you instead take another option, which is the following. The training set, let's say it's still 205,000 images, I would have the training set have all 200,000 images from the web. And then you can, if you want, add in 5,000 images from the

mobile app. And then for your dev and test sets, I guess my data sets size aren't drawn to scale. Your dev and test sets would be all mobile app images.
So the training set will include 200,000 images from the web and 5,000 from the mobile app. The dev set will be 2,500 images from the mobile app, and the test set will be 2,500 images also from the mobile app. The advantage of this way of splitting up your data into train, dev, and test, is that you're now aiming the target where you want it to be. You're telling your team, my dev set has data uploaded from the mobile app and that's the distribution of images you really care about, so let's try to build a machine learning system that does really well on the mobile app distribution of images. The disadvantage, of course, is that now your training distribution is different from your dev and test set distributions. But it turns out that this split of your data into train, dev and test will get you better performance over the long term. And we'll discuss later some specific techniques for dealing with your training sets coming from different distribution than your dev and test sets.

Let's look at another example. Let's say you're building a brand new product, a speech activated rearview mirror for a car. So this is a real product in China. It's making its way into other countries but you can build a rearview mirror to replace this little thing there, so that you can now talk to the rearview mirror and basically say, dear rearview mirror, please help me find navigational directions to the nearest gas station and it'll deal with it.



So how can you get data to train up a speech recognition system for this product? Well, maybe you've worked on speech recognition for a long time so you have a lot of data from other speech recognition applications, just not from a speech activated rearview mirror. Here's how you could split up your training and your dev and test sets. So for your training, you can take all the speech data you have that you've accumulated from working on other speech problems, such as data you purchased over the years from various speech recognition data vendors. And today you can actually buy data from vendors of x, y pairs, where x is an audio clip and y is a transcript. Or maybe you've worked on smart speakers, smart voice activated speakers, so you have some data from that. Maybe you've worked on voice activated keyboards and so on. And for the sake of argument, maybe you have 500,000 utterences from all of these sources. And for your dev and test set, maybe you have a much smaller data set that actually came from a speech activated rearview mirror because users are asking for navigational queries or trying to find directions to various places. This data set will maybe have a lot more street addresses, right? Please help me

navigate to this street address, or please help me navigate to this gas station. So this distribution of data will be very different than these on the left but this is really the data you care about, because this is what you need your product to do well on, so this is what you set your dev and test set to be. So what you do in this example is set your training set to be the 500,000 utterances on the left, and then your dev and test sets which I'll abbreviate D and T, these could be maybe 10,000 utterances each. That's drawn from actual the speech activated rearview mirror. Or alternatively, if you think you don't need to put all 20,000 examples from your speech activated rearview mirror into the dev and test sets, maybe you can take half of that and put that in the training set. So then the training set could be 510,000 utterances, including all 500 from there and 10,000 from the rearview mirror and then the dev and test sets could maybe be 5,000 utterances each. So of the 20,000 utterances, maybe 10k goes into the training set and 5k into the dev set and 5,000 into the test set. So this would be another reasonable way of splitting your data into train, dev, and test and this gives you a much bigger training set, over 500,000 utterances, than if you were to only use **speech activated rearview mirror data** for your training set. So in this video, you've seen a couple examples of when allowing your training set data to come from a different distribution than your dev and test set allows you to have much more training data and in these examples, it will cause your learning algorithm to perform better. Now one question you might ask is, should you always use all the data you have? The answer is subtle, it is not always yes in next section will see a counter example.
Summary diagram:

Training and testing on different distributions

Example: Cat vs Non-cat

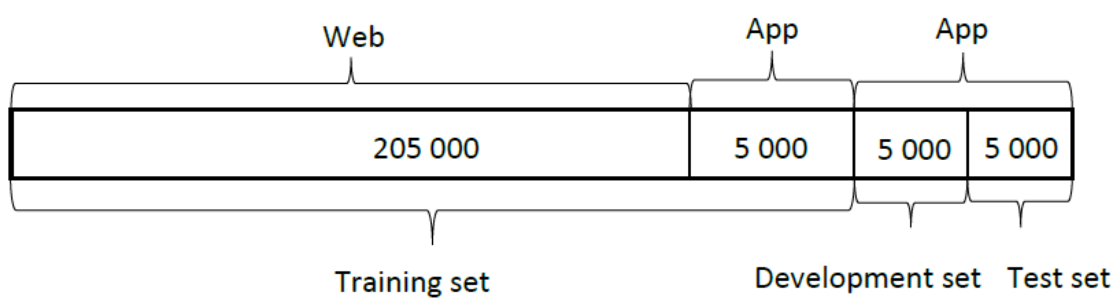
In this example, we want to create a mobile application that will classify and recognize pictures of cats taken and uploaded by users.

There are two sources of data used to develop the mobile app. The first data distribution is small, 10 000 pictures uploaded from the mobile application. Since they are from amateur users, the pictures are not professionally shot, not well framed and blurrier. The second source is from the web, you downloaded 200 000 pictures where cat's pictures are professionally framed and in high resolution.

The problem is that you have a different distribution:

1- small data set from pictures uploaded by users. This distribution is important for the mobile app.
2- bigger data set from the web.

The guideline used is that you have to choose a development set and test set to reflect data you expect to get in the future and consider important to do well.



The advantage of this way of splitting up is that the target is well defined.

The disadvantage is that the training distribution is different from the development and test set distributions. However, this way of splitting the data has a better performance in long term.

Bias and Variance with mismatched data distributions

Estimating the bias and variance of your learning algorithm really helps you prioritize what to work on next. But the way you analyze bias and variance changes when your training set comes from a different distribution than your dev and test sets. Let's see how.

Bias and variance with mismatched data distributions

Example: Cat classifier with mismatch data distribution

When the training set is from a different distribution than the development and test sets, the method to analyze bias and variance changes.

	Classification error (%)					
	Scenario A	Scenario B	Scenario C	Scenario D	Scenario E	Scenario F
Human (proxy for Bayes error)	0	0	0	0	0	4
Training error	1	1	1	10	10	7
Training-development error	-	9	1.5	11	11	10
Development error	10	10	10	12	20	6
Test error	-	-	-	-	-	6

Scenario A

If the development data comes from the same distribution as the training set, then there is a large variance problem and the algorithm is not generalizing well from the training set.

However, since the training data and the development data come from a different distribution, this conclusion cannot be drawn. There isn't necessarily a variance problem. The problem might be that the development set contains images that are more difficult to classify accurately.

When the training set, development and test sets distributions are different, two things change at the same time. First of all, the algorithm trained in the training set but not in the development set. Second of all, the distribution of data in the development set is different.

It's difficult to know which of these two changes what produces this 9% increase in error between the training set and the development set. To resolve this issue, we define a new subset called training-development set. This new subset has the same distribution as the training set, but it is not used for training the neural network.

Scenario B

The error between the training set and the training- development set is 8%. In this case, since the training set and training-development set come from the same distribution, the only difference between them is the neural network sorted the data in the training and not in the training development. The neural network is not generalizing well to data from the same distribution that it hadn't seen before

Therefore, we have really a variance problem.

Scenario C

In this case, we have a mismatch data problem since the 2 data sets come from different distribution.

Scenario D

In this case, the avoidable bias is high since the difference between Bayes error and training error is 10 %.

Scenario D

In this case, the avoidable bias is high since the difference between Bayes error and training error is 10 %.

Scenario E

In this case, there are 2 problems. The first one is that the avoidable bias is high since the difference between Bayes error and training error is 10 % and the second one is a data mismatched problem.

Scenario F

Development should never be done on the test set. However, the difference between the development set and the test set gives the degree of overfitting to the development set.

General formulation

Bayes error

$\updownarrow$ Avoidable Bias

Training set error

$\updownarrow$ Variance

Development - Training set error

$\updownarrow$ Data mismatch

Development set error

$\updownarrow$ Degree of overfitting to the development set

Test set error

Addressing data mismatch

If your training set comes from a different distribution, than your dev and test set, and if error analysis shows you that you have a data mismatch problem, what can you do? There are completely systematic solutions to this, but let's look at some things you could try. If I find that I have a large data mismatch problem, what I usually do is carry out manual error analysis and try to understand the differences between the training set and the dev/test sets. To avoid overfitting the test set, technically for error analysis, you should manually only look at a dev set and not at the test set. But as a concrete example, if you're building the speech-activated rear-view mirror application, you might look or, I guess if it's speech, listen to examples in your dev set to try to figure out how your dev set is different than your training set. So, for example, you might find that a lot of dev set examples are very noisy and there's a lot of car noise. And this is one way that your dev set differs from your training set. And maybe you find other categories of errors. For example, in the speech-activated rear-view mirror in your car, you might find that it's often mis-recognizing street numbers because there are a lot more navigational queries which will have street address. So, getting street numbers right is really important. When you have insight into the nature of the dev set errors, or you have insight into how the dev set may be different or harder than your training set, what you can do is then try to find ways to make the training data more similar. Or, alternatively, try to collect more data similar to your dev and test sets. So, for example, if you find that car noise in the background is a major source of error, one thing you could do is simulate noisy in-car data. So a little bit more about how to do this on the next slide. Or you find that you're having a hard time recognizing street numbers, maybe you can go and deliberately try to get more data of people speaking out numbers and add that to your training set. So, if your goal is to make the training data more similar to your dev set, what are some things you can do? One of the techniques you can use is artificial data synthesis.

So, to summarize, if you think you have a data mismatch problem, I recommend you do error analysis, or look at the training set, or look at the dev set to try this figure out, to try to gain insight into how these two distributions of data might differ. And then see if you can find some ways to get more training data that looks a bit more like your dev set. One of the ways we talked about is **artificial data synthesis** and artificial data synthesis does work. In speech recognition, I've seen artificial data synthesis significantly boost the performance of what were already very good speech recognition system. So, it can work very well. But, if you're using artificial data synthesis, just be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

Addressing data mismatch

This is a general guideline to address data mismatch:

- Perform manual error analysis to understand the error differences between training, development/test sets. Development should never be done on test set to avoid overfitting.

- Make training data or collect data similar to development and test sets. To make the training data more similar to your development set, you can use is artificial data synthesis. However, it is possible that if you might be accidentally simulating data only from a tiny subset of the space of all possible examples.

Learning from multiple tasks

Transfer learning

One of the most powerful ideas in deep learning is that sometimes you can take knowledge the neural network has learned from one task and apply that knowledge to a separate task. So for example, maybe you could have the neural network learn to recognize objects like cats and then use that knowledge or use part of that knowledge to help you do a better job reading x-ray scans. This is called **transfer learning**.

Transfer Learning

Transfer learning refers to using the neural network knowledge for another application.

When to use transfer learning

- Task A and B have the same input x
- A lot more data for Task A than Task B
- Low level features from Task A could be helpful for Task B

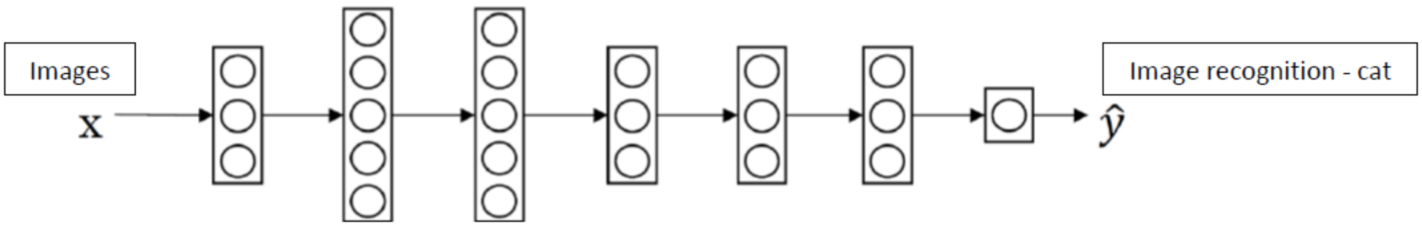
Example 1: Cat recognition - radiology diagnosis

The following neural network is trained for cat recognition, but we want to adapt it for radiology diagnosis. The neural network will learn about the structure and the nature of images. This initial phase of training on image recognition is called pre-training, since it will pre-initialize the weights of the neural network. Updating all the weights afterwards is called fine-tuning.

For cat recognition

Input x: image

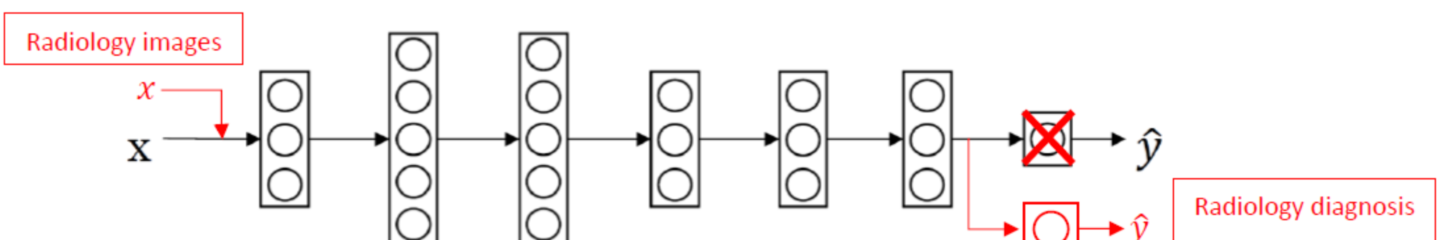
Output y – 1: cat, 0: no cat



Radiology diagnosis

Input x: Radiology images – CT Scan, X-rays

Output y :Radiology diagnosis – 1: tumor malign, 0: tumor benign



Guideline

- Delete last layer of neural network
- Delete weights feeding into the last output layer of the neural network
- Create a new set of randomly initialized weights for the last layer only
- New data set (x, y)

Multi-task learning

So whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B. In multi-task learning, you start off simultaneously, trying to have one neural network do several things at the same time and then each of these task helps hopefully all of the other task. Let's look at an example. Let's say you're building an autonomous vehicle, building a self driving car. Then your self driving car would need to detect several different things such as pedestrians, detect other cars, detect stop signs and also detect traffic lights and also other things.

Multi-task learning

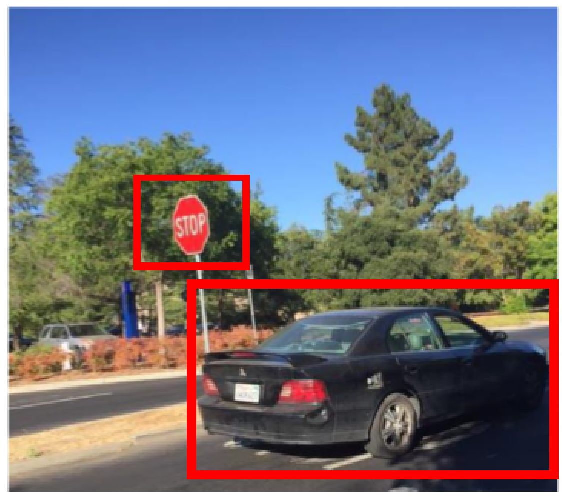
Multi-task learning refers to having one neural network do simultaneously several tasks.

When to use multi-task learning

- Training on a set of tasks that could benefit from having shared lower-level features
- Usually: Amount of data you have for each task is quite similar
- Can train a big enough neural network to do well on all tasks

Example: Simplified autonomous vehicle

The vehicle has to detect simultaneously several things: pedestrians, cars, road signs, traffic lights, cyclists, etc. We could have trained four separate neural networks, instead of train one to do four tasks. However, in this case, the performance of the system is better when one neural network is trained to do four tasks than training four separate neural networks since some of the earlier features in the neural network could be shared between the different types of objects.



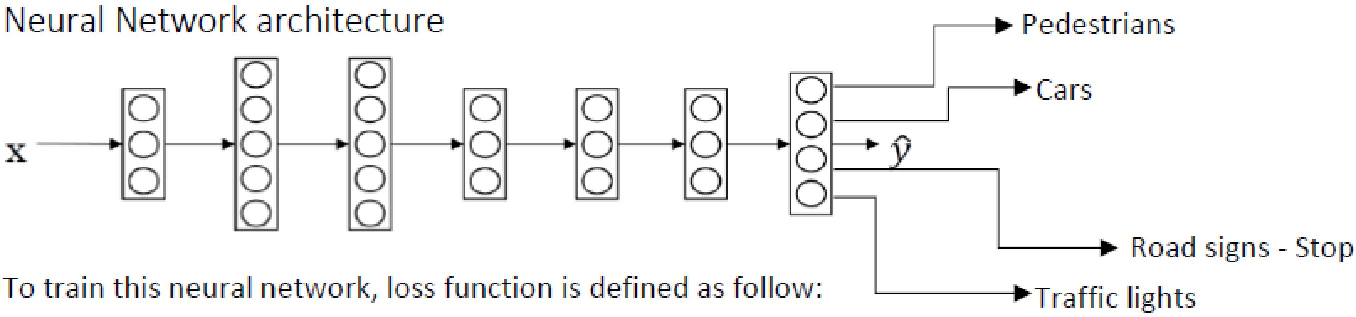
The input $x^{(i)}$ is the image with multiple labels
The output $y^{(i)}$ has 4 labels which are represents:

$$y^{(i)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \begin{matrix} \text{Pedestrians} \\ \text{Cars} \\ \text{Road signs - Stop} \\ \text{Traffic lights} \end{matrix}$$

$$\downarrow$$

$$Y = \begin{bmatrix} | & | & | & | \\ y^{(1)} & y^{(2)} & y^{(3)} & y^{(4)} \\ | & | & | & | \end{bmatrix} \qquad Y = (4, m) \qquad Y = (4,1)$$

Neural Network architecture



To train this neural network, loss function is defined as follow:

$$-\frac{1}{m}\sum_{i=1}^{m}\boxed{\sum_{j=1}^{4}}\left(y_j^{(i)}\log\left(\hat{y}_j^{(i)}\right) + \left(1 - y_j^{(i)}\right)\log\left(1 - \hat{y}_j^{(i)}\right)\right)$$

Also, the cost can be compute such as it is not influenced by the fact that some entries are not labeled.
Example:

$$Y = \begin{bmatrix} 1 & 0 & ? & ? \\ 0 & 1 & ? & 0 \\ 0 & 1 & ? & 1 \\ ? & 0 & 1 & 0 \end{bmatrix}$$

So what a researcher, Rich Carona, found many years ago was that the only times multi-task learning hurts performance compared to training separate neural networks is if your neural network isn't big enough. But if you can train a big enough neural network, then **multi-task learning** certainly should not or should very rarely hurt performance. And hopefully it will actually help performance compared to if you were training neural networks to do these different tasks in isolation. So that's it for multi-task learning. In practice, multi-task learning is used much less often than transfer learning. I see a lot of applications of transfer learning where you have a problem you want to solve with a small amount of data. So you find a related problem with a lot of data to learn something and transfer that to this new problem. But multi-task learning is just more rare that you have a huge set of tasks you want to use that you want to do well on, you can train all of those tasks at the same time. Maybe the one example is computer vision. In object detection I see more applications of multi-task any where one neural network trying to detect a whole bunch of objects at the same time works better than different neural networks trained separately to detect objects. But I would say that on average transfer learning is used much more today than multi-task learning, but both are useful tools to have in your arsenal.

So to summarize, multi-task learning enables you to train one neural network to do many tasks and this can give you better performance than if you were to do the tasks in isolation. Now one note of caution, in practice I see that transfer learning is used much more often than multi-task learning. So I do see a lot of tasks where if you want to solve a machine learning problem but you have a relatively small data set, then transfer learning can really help. Where if you find a related problem but you have a much bigger data set, you can train in your neural network from there and then transfer it to the problem where we have very low data. So transfer learning is used a lot today. There are some applications of transfer multi-task learning as well, but multi-task learning I think is used much less often than transfer learning. And maybe the one exception is computer vision object detection, where I do see a lot of applications of training a neural network to detect lots of different objects. And that works better than training separate neural networks and detecting the visual objects. But on average I think that even though transfer learning and multi-task learning often you're presented in a similar way, in practice I've seen a lot more applications of transfer learning than of multi-task learning. I think because often it's just difficult to set up or to find so many different tasks that you would actually want to train a single neural network for. Again, with some sort of computer vision, object detection examples being the most notable exception. So that's it for multi-task learning. Multi-task learning and transfer learning are both important tools to have in your tool bag.

End-to-end deep learning

What is end-to-end deep learning?

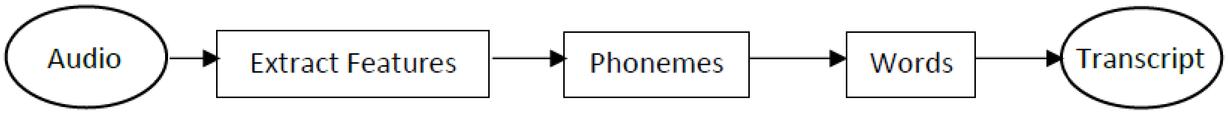
One of the most exciting recent developments in deep learning, has been the rise of end-to-end deep learning. So what is the end-to-end learning? Briefly, there have been some data processing systems, or learning systems that require multiple stages of processing. And what end-to-end deep learning does, is it can take all those multiple stages, and replace it usually with just a single neural network.

What is end-to-end deep learning

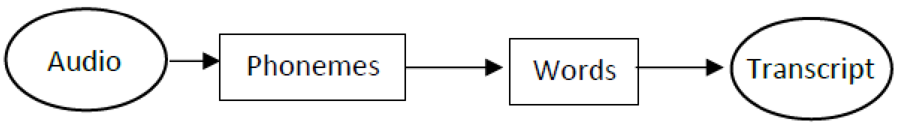
End-to-end deep learning is the simplification of a processing or learning systems into one neural network.

Example - Speech recognition model

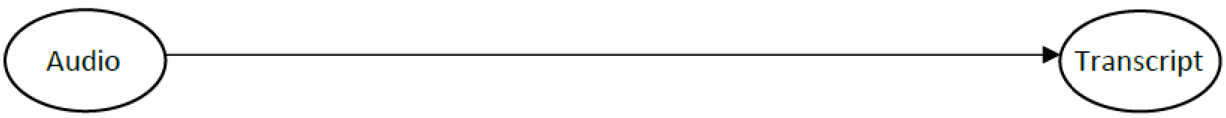
The traditional way - small data set



The hybrid way - medium data set



The End-to-End deep learning way – large data set



End-to-end deep learning cannot be used for every problem since it needs a lot of labeled data. It is used mainly in audio transcripts, image captures, image synthesis, machine translation, steering in self-driving

Whether to use end-to-end deep learning

Let's say in building a machine learning system you're trying to decide whether or not to use an end-to-end approach. Let's take a look at some of the pros and cons of end-to-end deep learning so that you can come away with some guidelines on whether or not an end-to-end approach seems promising for your application. Here are some of the benefits of applying end-to-end learning. First is that end-to-end learning really just lets the data speak. So if you have enough X,Y data then whatever is the most appropriate function mapping from X to Y, if you train a big enough neural network, hopefully the neural network will figure it out. And by having a pure machine learning approach, your neural network learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.

Whether to use end-to-end deep learning

Before applying end-to-end deep learning, you need to ask yourself the following question: Do you have enough data to learn a function of the complexity needed to map x and y?

Pro:

- Let the data speak
  - By having a pure machine learning approach, the neural network will learn from x to y. It will be able to find which statistics are in the data, rather than being forced to reflect human preconceptions.

- Less hand-designing of components needed
  - It simplifies the design work flow.

Cons:

- Large amount of labeled data
  - It cannot be used for every problem as it needs a lot of labeled data.

- Excludes potentially useful hand-designed component
  - Data and any hand-design's components or features are the 2 main sources of knowledge for a learning algorithm. If the data set is small than a hand-design system is a way to give manual knowledge into the algorithm.

END OF COURSE