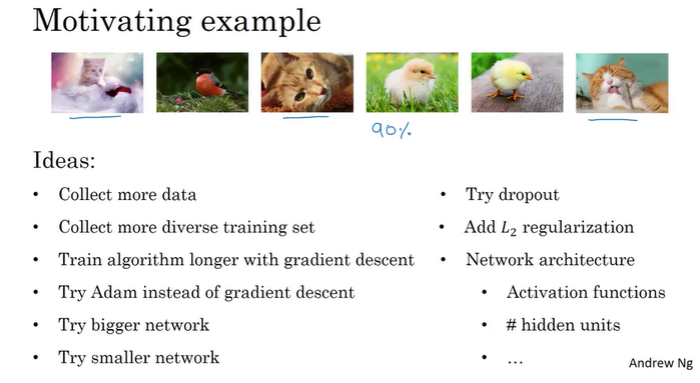
# ML Strategy

**Objective:**

How to diagnose what exactly is the bottleneck to your system's performance. As well as identify the specific set of knobs you could use to tune your system to improve that aspect of its performance.

You've gotten your system to have 90% accuracy, but this isn't good enough for your application



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.

## Orthogonalization

The art of knowing what parameter to tune to get what effect, is called orthogonalisation. Parameter tuning is one of the area where we have to picking right parameter to tune first from many possible ones.

**Orthogonalization is about what to tune to achieve one effect** - knowingly it is trait of successful machine learning practitioners.

**That means we need to separate tuning knob(step) for required effect, rather than a working on combined knob for multiple aspects.** For instance, in machine learning algorithm, we need perform well on following fronts i.e. for 4 different effects;

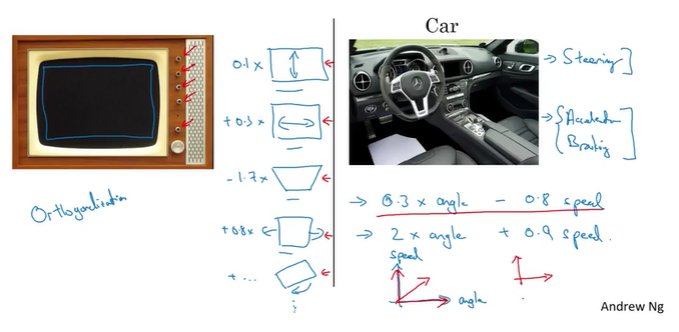
Training set (on cost function) ~ human level performance

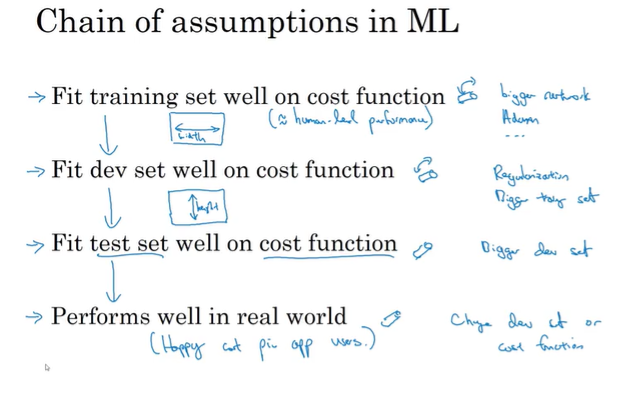
Dev set

Test set

Real world

According to orthogonalization, **we must achieve one effect from each at a time** - training set first and real world at the end.





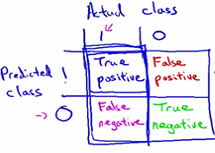
## Single number evaluation metric

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

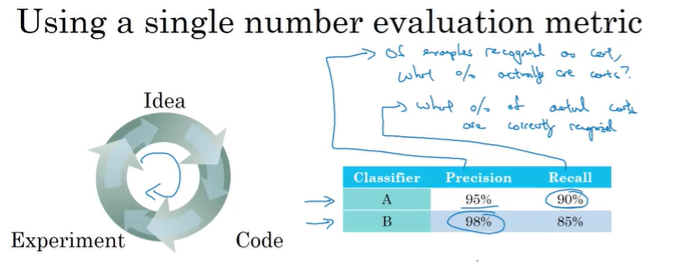
Have a single real number metric for your project as an evaluation metric so that you know if tuning a knob is helping your algorithm or not

While testing multiple scenarios, this metric can help you choose the most efficient algorithm quickly. Sometimes, you might need two metrics to evaluate your algorithm, say precision and recall. But with two evaluation metrics, it gets difficult to mark which algorithm is performing better.

### Precision/Recall



* Recall is TP/ Actual positive
* Precision is  **TP/Predicted positive.**

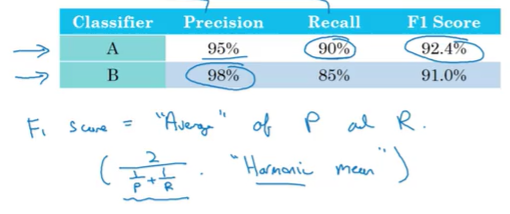


The problem with using precision recall as your evaluation metric is that if classifier A does better on recall, which it does here, the classifier B does better on precision, then you're not sure which classifier is better.

**So what I recommend 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.**

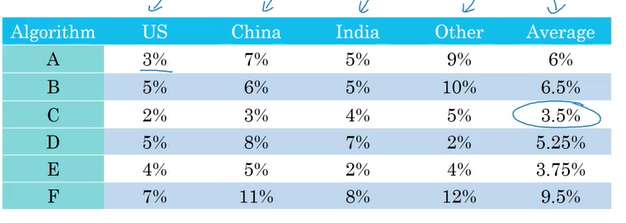
### F1 Score

Which classifier is better?



Evaluation metric allows you to quickly tell if classifier A or classifier B is better, and therefore having a **dev set plus single number evaluation metric distance to speed up iterating.**

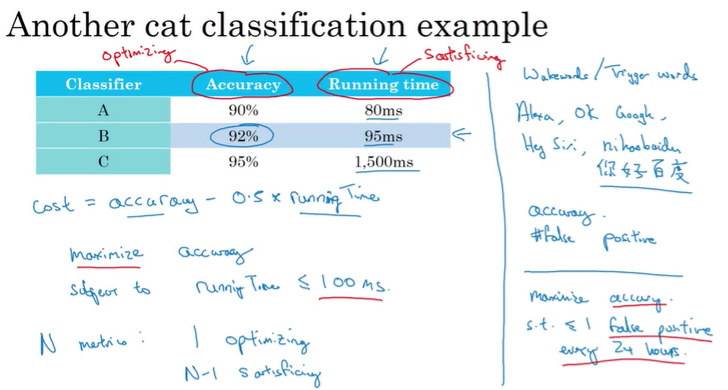
Ex: cat app with errors for A &B algorithms/classifiers



Average error might be used to choose the classifier.

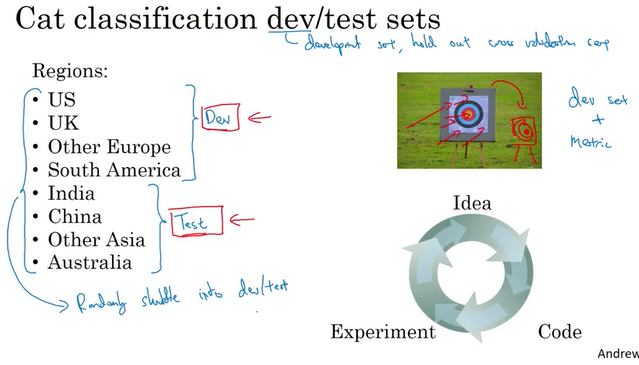
## Satisficing/Optimizing Metric

There might be cases where you don’t have just 1 (or 2) but n metrics that you care about for your system. Suppose you are asked to create a classifier with maximum accuracy, with minimum time and space complexity. You build the following 4 classifiers, which one do you deliver?



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



* Randomly shuffle into dev and test data
* Choose a dev set and test set**(same distribution)** to reflect data you expect to get in the future and consider important to do well on

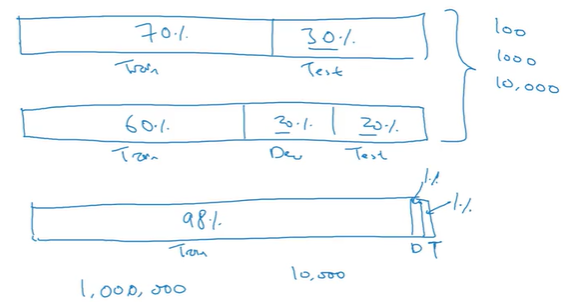
## Size of the dev and test sets

**Training set**: a set of examples used for learning: to fit the parameters of the classifier In the MLP case, we would use the training set to find the “optimal” weights with the back-prop rule

**Validation set:** a set of examples used to tune the parameters of a classifier In the MLP case, we would use the validation **set to find the “optimal” number of hidden units or determine a stopping point for the back-propagation algorithm**

**Test set:** a set of examples used only to assess the performance of a fully-trained classifier In the MLP case, we would use the test to estimate the error rate after we have chosen the final model (MLP size and actual weights). After assessing the final model on the test set, **YOU MUST NOT tune the model any further!**

Why separate test and validation sets? The error rate estimate of the final model on validation data will be biased (smaller than the true error rate) since the validation set is used to select the final model After assessing the final model on the test set, YOU MUST NOT tune the model any further!



## Why separate dev and test set?

Validation set is used to tune your hyperparameters. eg. number of nodes,layers, amount of regularization you want in your archiṭecture etc.This is also used to make sure if you're not overfitting the data.

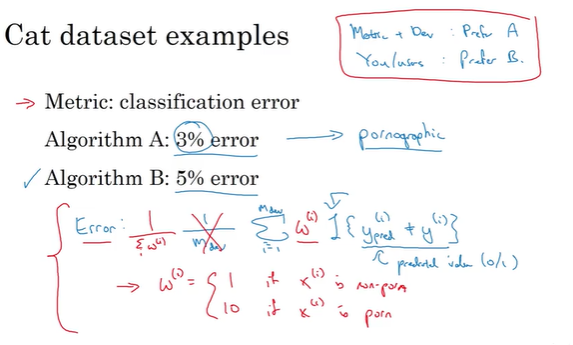
Now, since you used this set to tune your architecture, it is natural that your architecture will work great for the set.

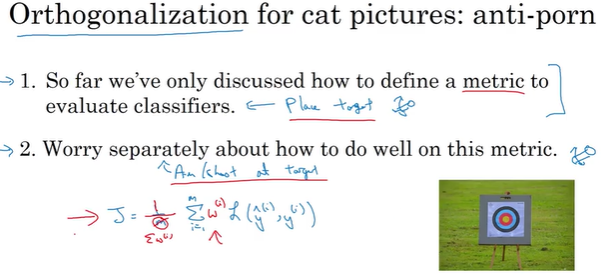
But at the end, the data that your model will be taking as input won't be a part of this validation set. It would rather be something you haven't used to make the tweaks in your model.

So, to make sure that your tuning wasn't biased to the validation set as well, one last step, just to make sure of the quality of your model. Test set is used.

## When to change dev/test sets and metrics

While algorithm A has a better error metric, it contains porno; so another metric is added where pornographic pics included algorithm is classified as W=0.

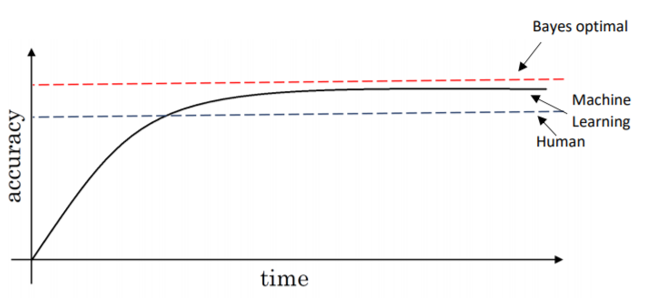




Why human-level performance?

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.(purple line)

## Bayes optimal



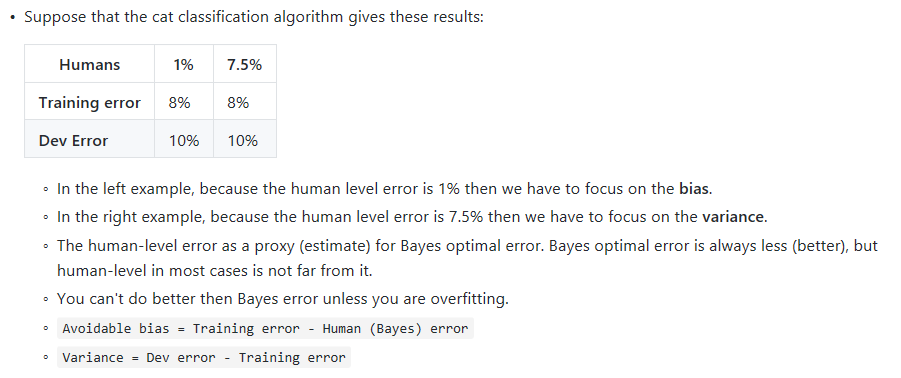
Usually, human and Bayes error are quite close, especially for natural perception problems, and there is little scope for improvement after surpassing human-level performance and thus, learning slows down considerably.

So long as your algorithm is doing worse than humans, following methods can be used to improve performance -

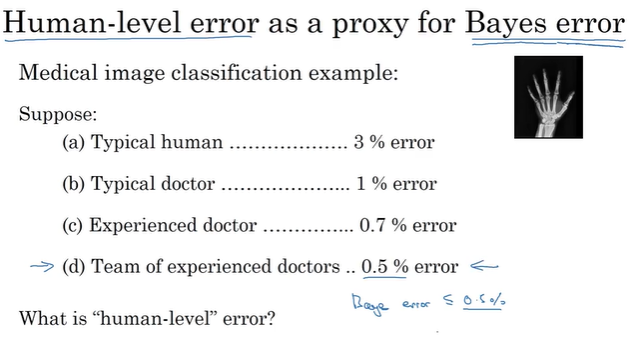
* Get labelled data from humans
* Gain insights from manual error analysis, e.g. understand why a human got this right
* Better analysis of Bias/Variance

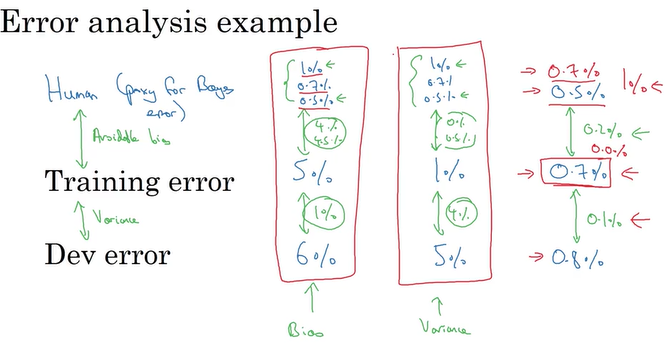
## Avoidable bias

* If your algorithm is doing much worse than human..



## Understanding human-level performance



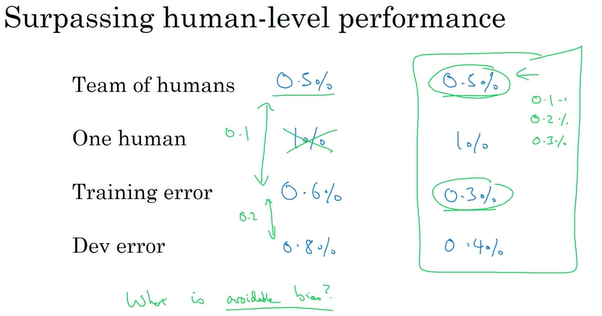


And the difference between training error and dev error, that tells you how much variance is a problem, whether your algorithm's able to generalize from the training set to the dev set.

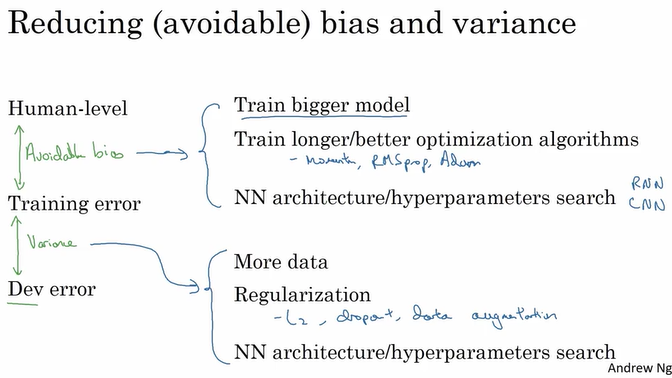
So to recap, having an estimate of human-level performance gives you an estimate of Bayes error. And this allows you to more quickly make decisions as to whether you should focus on trying to reduce a bias or trying to reduce the variance of your algorithm.

And these techniques will tend to **work well until you surpass human-level performance**, whereupon you might no longer have a good estimate of Bayes error that still helps you make this decision really clearly

## Surpassing human level performance



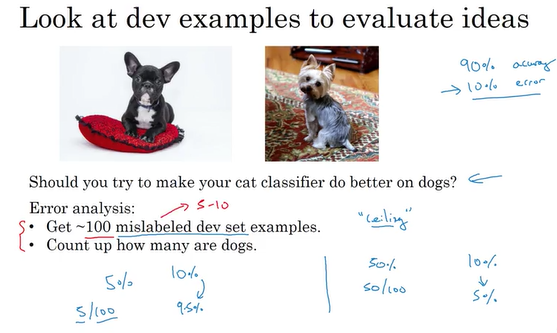
Now, what is the avoidable bias? (the example on the right) It's now actually much harder to answer that. Is the fact that your training error, 0.3%, does this mean you've over-fitted by 0.2%, or is base error, actually 0.1%, or maybe is base error 0.2%, or maybe base error is 0.3%? You don't really know, but based on the information given in this example,



# W2

## 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.



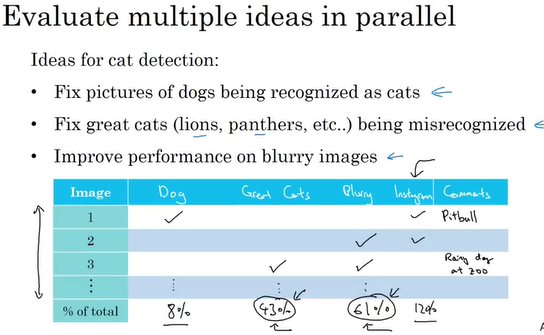
* Cat classifier with 10% error;
* Misclassifying problem; mainly some dogs pics as cats.
* To make algorithm better: focus effort to collect more dog pics etc.
* **So the question is**, should you go ahead and start a project focus on the dog problem?
  + Rather than spending a few months doing this, 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.
* Take 100 mislabeled pictures and figure out how many of them are actually dog.
* Let’s say 5 of them dog. So if you focus on dog problem then you could decrease error from 10% to 9.5% only.

## Ceiling on performance

What's in the best case? (How well could working on the dog problem help you?)

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.

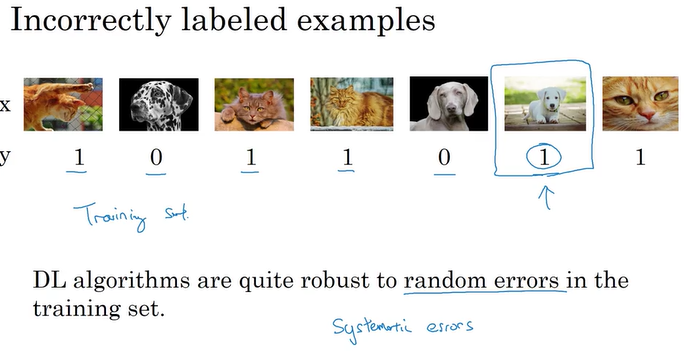


So to summarize,

* 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. So what do you do about that? Let's discuss that in the next video.

## Cleaning up incorrectly labeled data

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

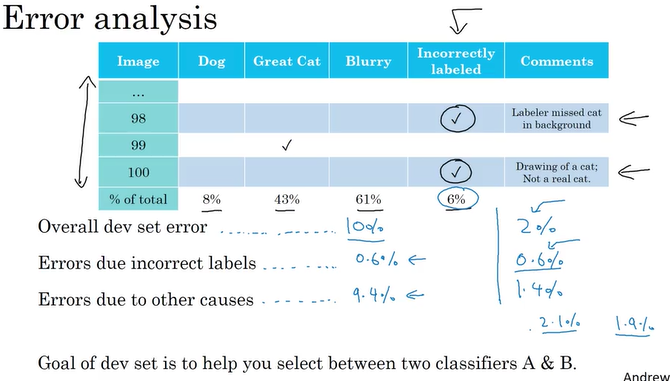


Deep learning algorithms are robust to random errors. They are less robust to systematic errors

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

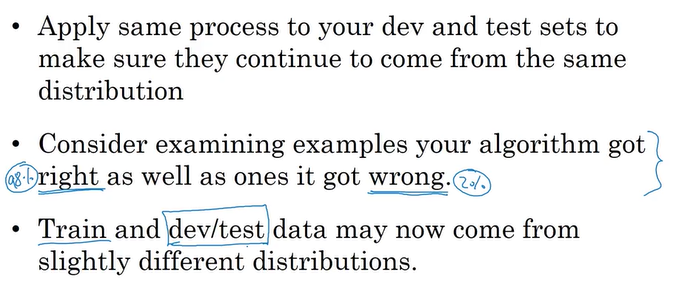
* How about incorrectly labeled examples in your dev set or test set?

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



* 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.
* If the errors(incorrect labels) is only 6 percent of all errors, no need to fix the incorrect labels.(above pic left side)
* But if the error ratio of the same label is 30%(0.6 in 2 percent), then you try to fix that.(right side above without looking at the dev set error performance)

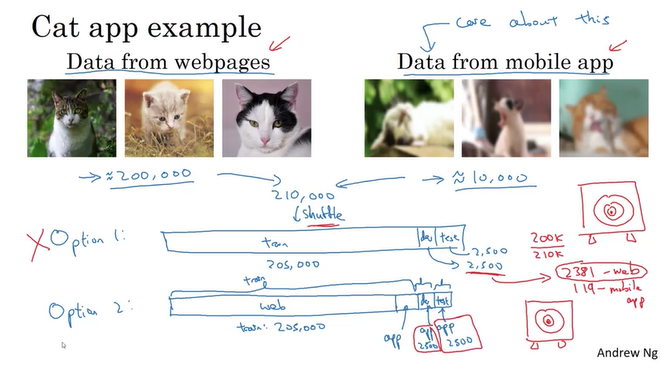
## Correcting Incorrect dev/test set examples



## Build your first system quickly, then iterate

* Set up dev/test set and metric
* Build initial system quickly
* Use Bias/Variance analysis & Error analysis to prioritize next steps
* 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

## Training and testing on different distributions



In the example above, you care more about mobile app pictures, but you don’t have enough mobile pics yet. Then you will get data from web pages.

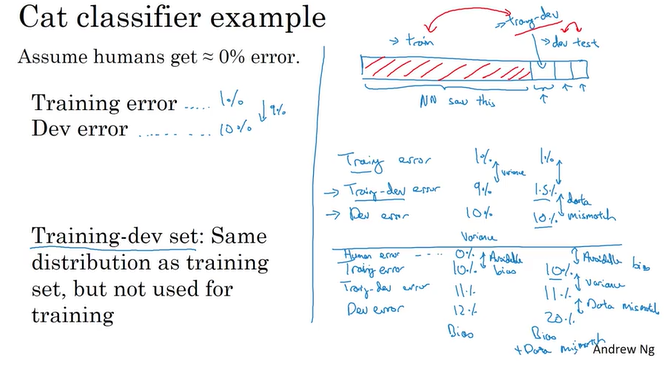
There are two options to select train/dev/test sets

* First you can take 200K webpage pics and 10K mobile pics; shuffle them and split to training, dev and test sets.
  + 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
  + The disadvantage is you will use only 119 mobile app pics for dev set which is important for you to
  + 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.**
* Second you can take 200K webpage pics, 5K mobile pics and put them to training data,; then you can put all dev and test set from mobile app pics.
  + 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, 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

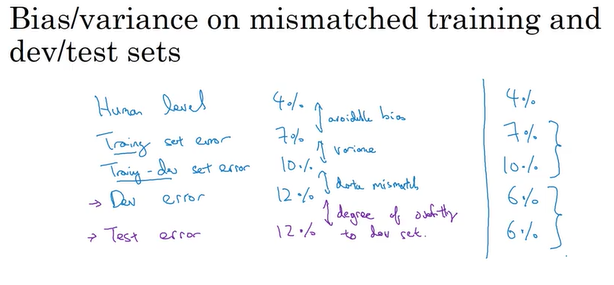
## Bias and Variance with mismatched data distributions

Estimating the bias and variance of your learning algorithm really helps you prioritize what to work on next

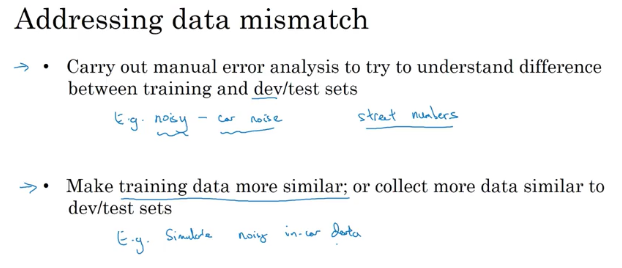
But the way you analyze bias and variance changes when your training set comes from a different distribution than your dev and test sets.



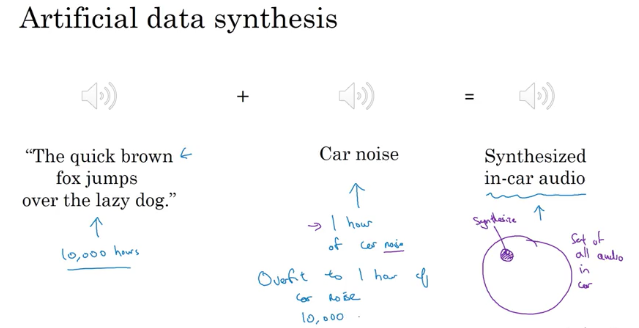
* You would say first there is a large variance problem, that your algorithm's just not generalizing well. But in the setting where your training data and your dev data comes from a different distribution, you can no longer safely draw this conclusion.
* In particular, maybe it's doing just fine on the dev set, it's just that the training set was really easy because it was high res, very clear images, and maybe the dev set is just much harder.
* So maybe there isn't a variance problem and this just reflects that the dev set contains images that are much more difficult to classify accurately.
* So the problem with this analysis is that when you went from the training error to the dev error, two things changed at a time. One is that the algorithm saw data in the training set but not in the dev set. Two, the distribution of data in the dev set is different. And because you changed two things at the same time, it's difficult to know of this 9% increase in error, how much of it is because the algorithm didn't see the data in the dev set, so that's some of the variance part of the problem. And how much of it, is because the dev set data is just different.
* So we create a new subset (training dev set) which is composed of training data; but NN doesn’t train from this data. To catch if there is a variance problem.



## Addressing data mismatch



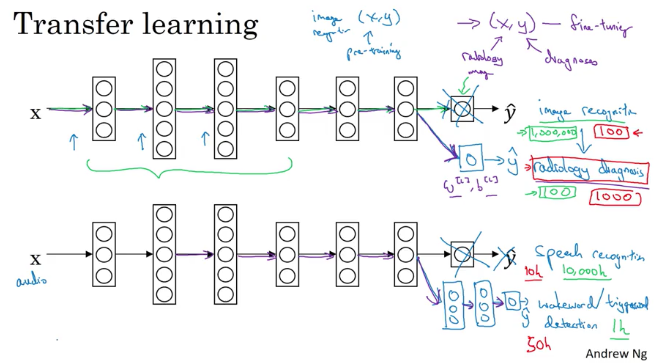
### Artificial Data Synthesis



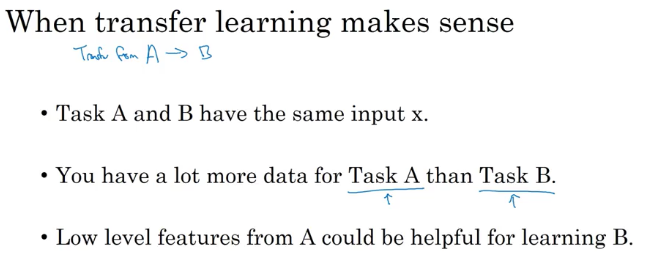
* Little data of car noise, much more data of clean voice; you synthesize these two.
* Overfitting is possible
* If you think you have a data mismatch problem,
  + I recommend you do error analysis, or look at the training set, or look at the dev set to try this figure out, to try to gain insight into 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.

## Transfer Learning

* 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.

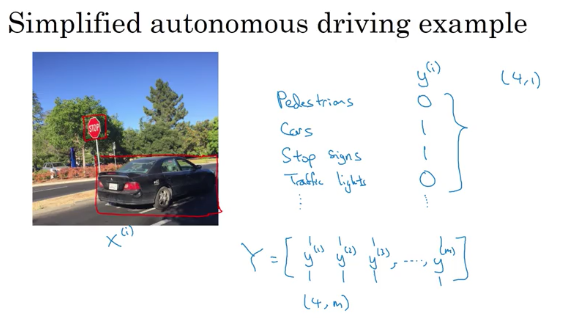


* So you first take a neural network and train it on X Y pairs, where X is an image and Y is some object. An image is a cat or a dog or a bird or something else. If you want to take this neural network and adapt, or we say transfer, what is learned to a different task, such as radiology diagnosis

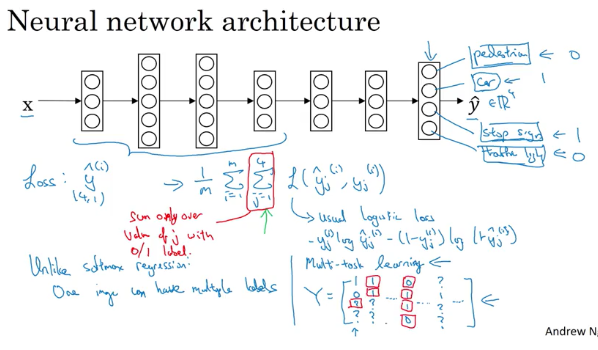


* If Task A actually has less data than Task B and in those cases, you kind of don't expect to see much of a gain

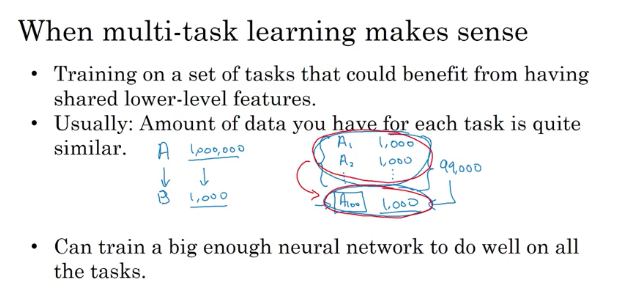
## Multitask learning



* One image can have multiple label so y is 4,1 dimensional
* One other thing you could have done is just train four separate neural networks, instead of train one network to do four N.
* But if some of the earlier features in neural network can be shared between these different types of objects, then you find that training one neural network to do four things results in better performance than training four completely separate neural networks to do the four tasks separately.
* Some labels may be 1 0 , some maybe unknown(?)



* So whenever there's a question mark, you just omit that term from summation but just sum over only the values where there is a label.



* In practice, multi-task learning is used much less often than transfer learning.(one exception computer vision object detection)
* 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
* you can train in your bigger neural network and then transfer it to the problem where we have very low data
* you can compute the cost such that it is not influenced by the fact that some entries haven’t been labeled.

## End-to-end deep learning

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

It maps directly the input (x) to the output