Here we discuss several ways to improve the machine learning algorithm. There are so many things we can change for the algorithm, but the question is what are the most effective one!

Orthogonalization

Most data scientist have a clear eye, what to change in order to get specific effect. This is called orthogonalization. For example, in the TV remote, you know exactly pressing each bottom results in what action.

Chain of assumption in ML:

- Fit training set well on cost function → if fit not good, try using Adam, bigger network→ avoidable bias
- Fit dev set well→ if fit not good, use regularization, bigger training set (generalize better)→ avoidable variance
- Fit test set well → if fit not good, try bigger dev set
- Perform well in real word →if fit not good, change cost function or dev set

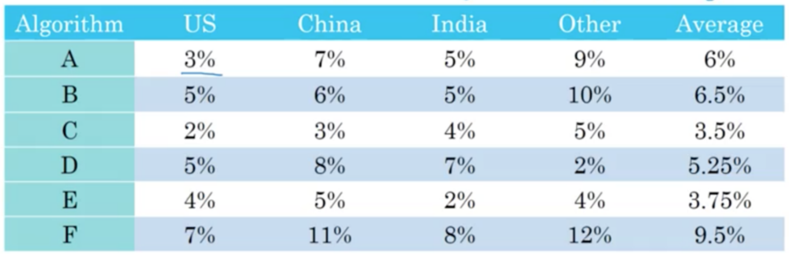
Try not to use early stopping. Because it effects the performance on the training set, also improve the dev performance. Instead you can use other adjustment methods.

Single number evaluation

Using a single number for evaluation helps deciding the direction to take in algorithm. For example, imaging you are working on classifier and using precision and recall for evaluation. It is quite hard to pick the best algorithms. Instead, use F1 score (harmonic mean of precision and recall).

Having a well-defined dev set, and single number evaluation metric allows you to see how well algorithm does.

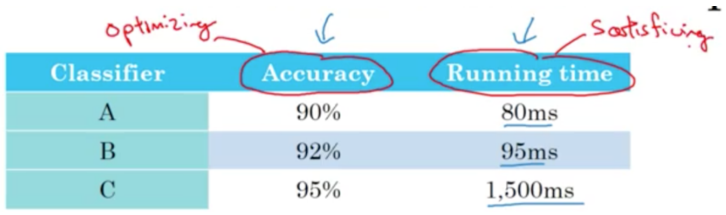
Another example: you are testing your algorithm for cat lovers around the world and for each nationality, you get different error. The best way to compare the result and picking the best algorithm is to average all of the performance and deciding based on that.



Algorithm	US	China	India	Other	Average
A	3%	7%	5%	9%	6%
B	5%	6%	5%	10%	6.5%
C	2%	3%	4%	5%	3.5%
D	5%	8%	7%	2%	5.25%
E	4%	5%	2%	4%	3.75%
F	7%	11%	8%	12%	9.5%

Satisfying and optimizing metric

Sometimes, you care about the running time beside the accuracy. you can have set a goal as follow: 1- maximize accuracy 2- running time<100 ms. Accuracy is the

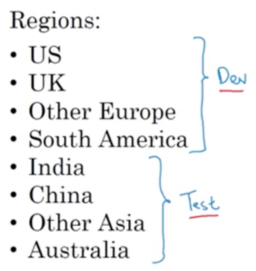

optimizing
Satisficing

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

optimizing metric (you would like to have it as high as possible) and running time is satisfying metric.
If you have n metric you care about, pick one metric to be the optimizing metric and the rest (n-1) satisfying metrics.

Train/dev/ test distribution

Dev set sometimes is called holdout cross validation set or development set. *Dev set, and test set must come from the same distribution*. The example on the right is the wrong way of splitting the set. Instead shuffle all of the nation and pick the test set and dev set.



One of the true example happened: the bank optimizes on dev set on loan approval for medium income zip codes in order to predict, if they get approve for the loan, what is the probability of them paying it. They test it on the low-income zip code and the result was very different. They had to redo the algorithm and wasted 3 months of work.

As a rule of thumb, get a dev set and test set to reflect the data you expect to get in future. Dev and test set must come from the same distribution.

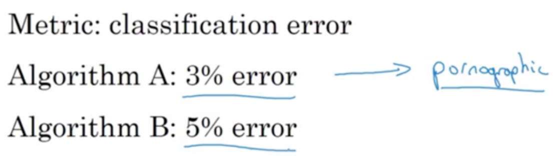
Size of dev/test sets

Old way of splitting data: 30% test, 70% train or 60% train, 20% dev, 20% test set → reasonable for small dataset.
Modern machine learning era: we have lots of data (like 1,000,000 set size) to have 98% train set, 1% test and 1% dev set.

Size of test set: set it to big enough to have high confidence in overall performance of system. Sometimes not having test set is ok. Instead only train and dev set.

When to change dev/test set and metrics?

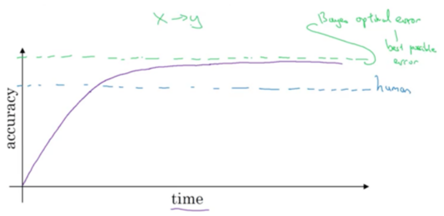
Imagine you are classifying the cat images, one algorithm is doing great, but sometimes shows the pornography images. In this case, the user prefers algorithm B with more error. In this case, we may want to change the evaluation metric. In this case, we add wi to the metric, to consider porn images to lower the metric if we classify them. Consider both rank and preference simultaneously to make one single evaluation metric.


Metric: classification error
Algorithm A: 3% error
pornographic
Algorithm B: 5% error

If the evaluation metric is doing well on the dev/test set, but not corresponds well to our application, we may want to change the metric. For example, if algorithm picks bunch of blurry cat images instead of high quality images of cat.

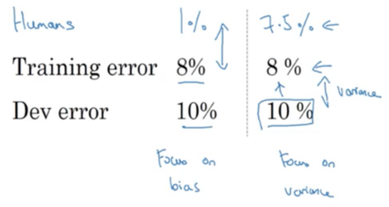
Comparing to human level performance

The accuracy over time increases, even it gets more accurate than the human level, until it reaches the Bayes optimal error (best possible error).



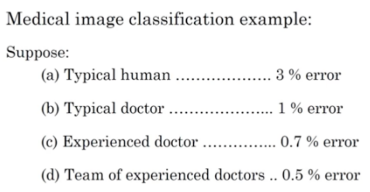
Avoidable bias

Imagine classifying the cat image. Human error is about 1% while your training error is 8%. Means that you can improve the model quiet a lot. But if human error is 7% and you get training error of 8%, your algorithm is doing pretty good. Depends on the condition, we may focus on reducing the bias or variance.
Use human level error, to estimate the Bays error.



Humans	1%	7.5%
Training error	8%	8 %
Dev error	10%	10 %
	Focus on bias	Focus on variance

Avoidable bias or avoidable variance is difference between training error and human error. Bays error must be lower than the experience team in the subject.



Medical image classification example:

Suppose:

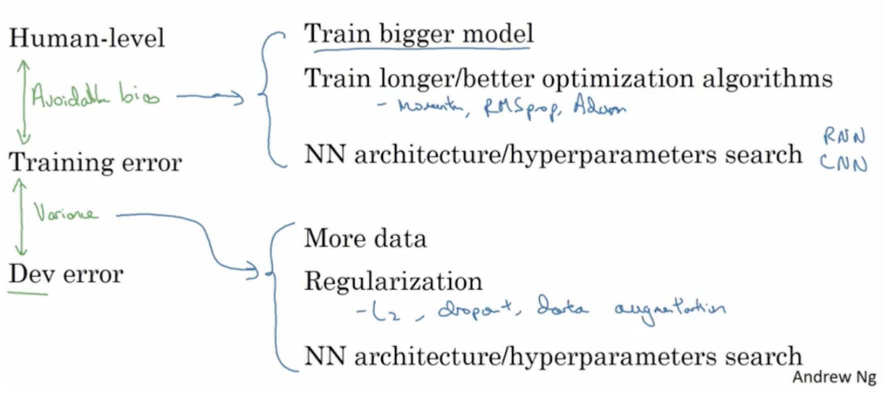
(a) Typical human	3 % error
(b) Typical doctor	1 % error
(c) Experienced doctor	0.7 % error
(d) Team of experienced doctors	0.5 % error

Big difference between human error and train error → Reduce the bias → train on more data
Big difference between training error and dev error → Reduce variance→ regularization, bigger training set

Problems with ML that suppress human level performance (not natural process tasks, huge amount of data):

- Online advertising
- Recommendation
- Loan approval
- Logistic (predicting transit time)
- Some image recognition
- Some speech recognition

Getting better estimate than human error is hard, but by increasing the training set we can overcome it. In summary:


Human-level
Avoidable bias
Training error
Variance
Dev error
Train bigger model
Train longer/better optimization algorithms
- momentum, RMSprop, Adam
NN architecture/hyperparameters search
RNN
CNN
More data
Regularization
- L2, dropout, data augmentation
NN architecture/hyperparameters search
Andrew Ng

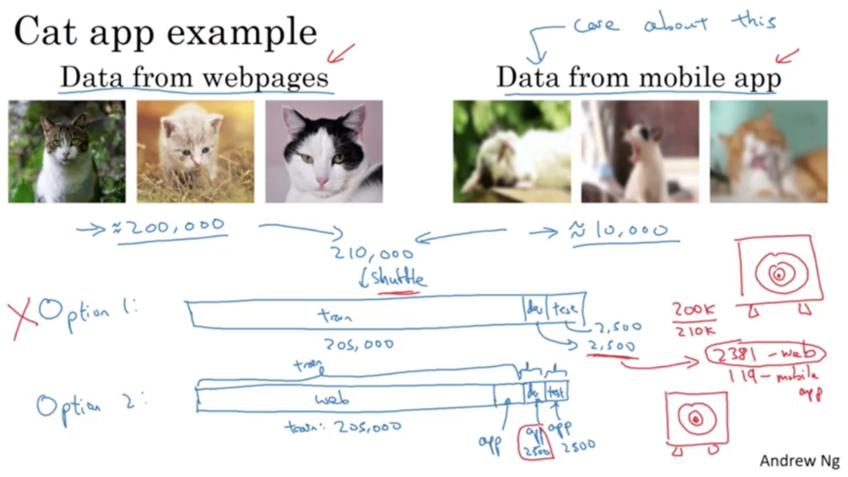
Error analysis

Imaging in cat classifier, you get bunch of dog pictures are mislabeled as cat. You may want to add a dog classifier too. *It's good to manually go over the wrong labeled images and study the reason for the wrong mislabeling*. Sometimes you may want to evaluate multiple ideas in parallel to increase the performance of you model.

Build your first system, then iterate and change the parameters as you are improving the model.

Training and testing on different distribution

Imagine, we have cat images from 2 different sources such as web and mobile. We mainly care about the mobile app images. The best way to split data into test and train is use all of the web images and few of the mobile app picture as training and dev/ test set to be chosen from the mobile app, since that's what we are aiming for. Even though the distribution of train and dev are different.



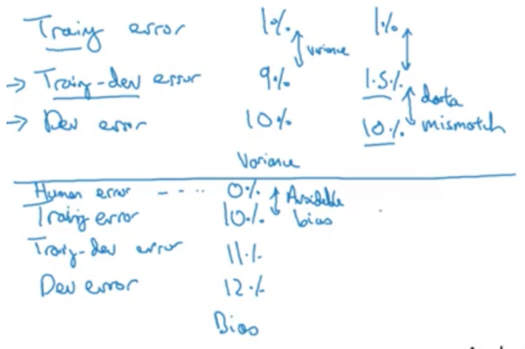
This way of splitting gives better performance in a long run.

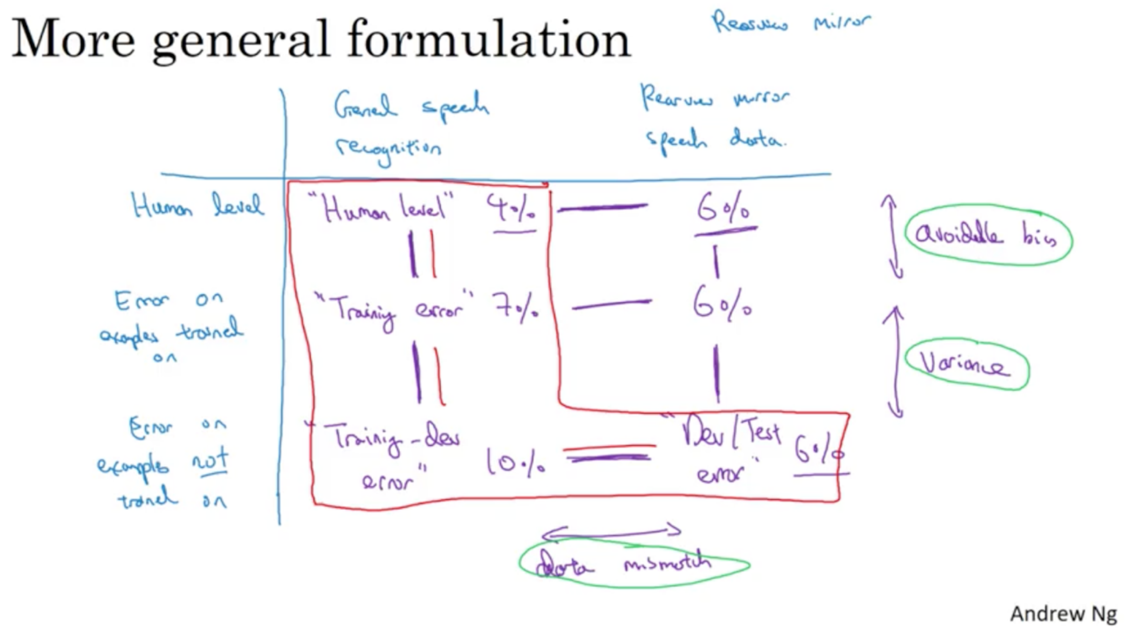
In cat classifier, if we get training error of 1% and dev error of 10%:

- If dev and train set, come from same distribution → we have a high variance problem
- If dev and train set come from different distribution → inclusive, maybe we have much harder set to predict from→ how can we do a comparison then? Here is few suggestions:

Have a train-dev set, that they come from the same distribution, but not used for training.

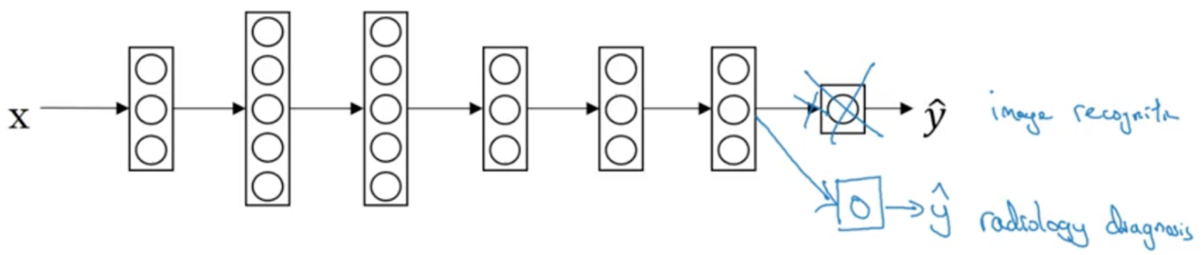
Basically, we split the data into trainset, train/dev set, dev set, and test set. Consider the training error, and training/dev. Train error and train/dev error should be comparable, if the error is high, it means we have variance problem. In summary:





Transfer learning

For example, we trained the system to recognize the cat images, now we are interested in dog images. We can transfer the learning. In NN, we take last layer of NN and delete the last weights feeding into it, initialize the weight for last output randomly and have the new problem diagnosis (dog image in this case). Basically we use the pretraining and fine tuning by manipulating the last layer.



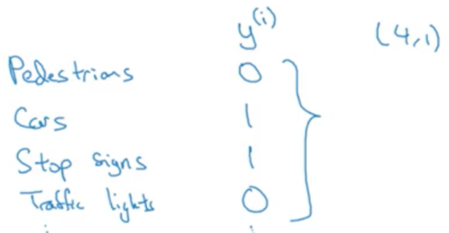
We may retrain the last layer or several layers back.

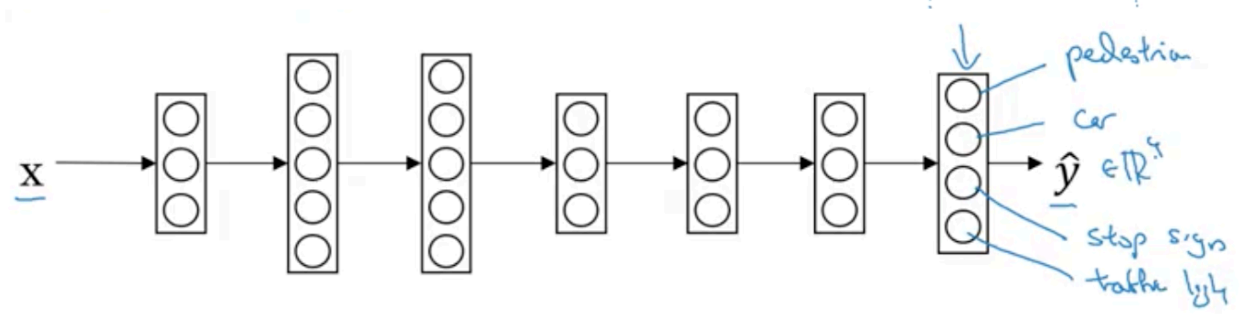
Transfer learning is mainly used when we have a big set of training examples for previous problem, but not many on the new problem. (In the opposite case, it doesn't make sense to do it). In summary it makes sense when:

- Task A and B have the same input x (like images)
- You have a lot more data for task A than B
- Low level features from A could be helpful for learning B

Multitask Learning

Here the NN simultaneously trying to do several things at the same time. Each of these task, help the other tasks. For example, self-driving cars need to predict pedestrians, cars, lights, stop sighs and etc. here is a 4 by 1 metrics.



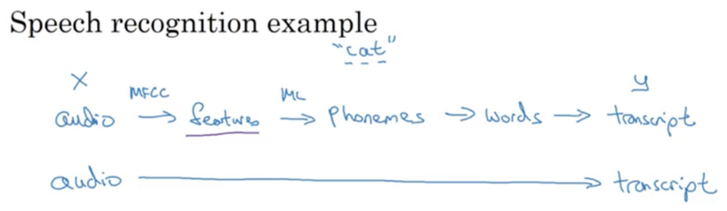


Use case of multitask learning:

- Training on a set of tasks that could benefit from having shared lower level features
- Usually, amount of data we have for each task is quite similar
- Can train a big enough NN to do well on all tasks (if it's not big, it will hurt the performance)

End to end deep learning

For example, for speech recognition, there are two approaches you might take, one is studying and separating the modules and the other one predicting end to end.



Usually the end to end learning works better than predicting in multiple steps. For end to end learning:

- We need a big amount of data (like 10,000 hours of speech data)
- Pros: data speak for itself, less hand designing of components