Course 3 - Structuring ML Projects

Mohammad Khalaji

August 23, 2020

1 Week 1

1.1 Orthogonalization

What to tune in order to get specific optimization. For example, in a TV, each knob is dedicated to tuning one and only one parameter. This makes it much easier to tune the TV. As another example, if the steering wheel in a car changed the acceleration too, it would be extremely difficult to control the car properly. In fact, the steering wheel, the break, and the acceleration pedal are orthogonal, and they make it much easier for the driver to control the car.

Chain of assumptions in ML:

- Fit training set well on cost function (in some occasions, comparable to human-level performance)

  How to tune?

  - Adam
  - Bigger network
- Fit dev set well on cost function

  How to tune?

  - Regularization
  - Bigger train set
- Fit test set well on cost function

  How to tune?

  - Bigger dev set
- Perform well in real world

  How to tune?

  - Change dev set
  - Change the cost function

Normally, it is better not to use early stopping because it is a "knob" that modifies the performance on both training set and dev set.

1.2 Single Number Evaluation Metrics

There often is a trade-off between precision and recall, and we care about both of them. As a result, rather than using 2 numbers to compare different models, we only use one number.

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} \ (Harmonic\ Mean)$$

1.3 Satisficing and Optimizing Metrics

It's not always easy to have one single evaluation metric, while many important metrics are present.

Classifier	Accuracy	Running Time
A	90%	80ms
B	92%	95ms
C	95%	1500ms

It is not a good idea to set $metric = accuracy - 0.5 \times runningTime$. However we could take one approach: maximize the accuracy, and keep the running time below $100ms$. This way, accuracy is a **Optimizing Metric**, whereas running time is a **Satisficing Metric**.

In another example, consider a situation where you are training a wakeword/trigger detector neural network. Although accuracy is very important in this network, the number of false positives is a very important metric too because you don't want too wake your device up without a valid trigger. As a result, your metrics might be like this:

- Maximize accuracy (Optimizing Metric)
- Keep the number of false positives at most 1 in every 24 hours (Satisficing Metric)

1.4 Train/Dev/Test Sets Distribution

Dev and test sets must not come from different distributions. Having dev and test sets from different distributions is like setting a target, spend many month on reaching it, only to realize that you have moved the target to a different position for the testing phase!

1.5 Size of Dev and Test Sets

In the modern era of deep learning, dataset sizes are growing larger and larger. In past, researchers would use a 60%/20%/20% split for Train/Test/Dev sets. However, when datasets have 1000000 examples, for instance, using only 2% of the data for dev and test sets is reasonable because 1 percent of 1000000 still contains a lot of examples.

Size of your test set must be big enough to give us enough confidence about the model.

Sometimes, a Train/Dev split is enough. However, it is not recommended.

1.6 When to Change Dev/Test Sets

For example, when your model performs poorly on specific kinds of data, you must add those kinds to your Dev/Test sets in order to train a more sensitive method. You can also change your metric in order to incorporate your sensitivity to that kinds of data.

1.7 Why Human Level Performance?

Neural net models surpass human level performance quickly, but they struggle while trying to perform even better. This is because of **Bayes Optimal Error**, which is the best that we can possibly get. Since humans are pretty good at doing certain tasks, the gap between human performance and Bayes error is small. As a result, it is difficult for AI models to get closer to Bayes error.

So long as ML is worse than humans, we can do the following:

- Get labeled data from humans
- Analyze bias/variance
- Conduct manual error analysis

1.8 Avoidable Bias

We don't want to do *too well* on the train set. When human error is 1%, a model with 8% error can be still improved. However, when human error on your dataset is 7.5%, a model with 8% might not be bad at all!

Take human error as a proxy for Bayes error. Then decide on whether to improve the bias or the variance.

- If there is a noticeable difference between train error and Bayes error ($\approx$ human error), focus on your bias problem in order to get closer to Bayes error.
- Otherwise, focus on variance problem (if any)

The difference between Bayes error and human error is called the **Avoidable Bias**. It means that there is some bias that you cannot possibly get below, unless you are overfitting, or you're dealing with special cases that are explained later.

1.9 Surpassing Human Level Performance

- Online Advertising
- Product Recommendations
- Logistics (Predicting Transit Time)
- Loan Approvals

Generally, they come from structured data, and they are not *natural perception* problems (unlike image classification or NLP)

1.10 Improving Model Performance

To solve the problem of avoidable bias and getting closer to human-level performance:

- Train a bigger model
- Train longer/better optimization algorithms
- NN architecture/hyperparameters search

To solve the problem of variance:

- More data
- Regularization
- NN architecture/hyperparameters search

2 Week 2

2.1 Carrying Out Error Analysis

Manually examining the examples in which our network is not performing well. Consider a cat classifier neural network that has 90% accuracy, and most of examples on which the network performs poorly are dog pictures. As a result, should we try to make our classifier do better on dog pictures?

- Collect 100 mislabeled dev set pictures
- Count up to how many are dogs
  - 5/100 are dog pictures. So if we try to do better on dog pictures, in the best case, the error will go from 10% to 9.5%. It's not worth it.
  - 50/100 are dog pictures. So if we try to do better on dog pictures, in the best case, the error will go from 10% to 5%. It's not worth it.

You can also take this approach for multiple kinds of mislabeled data.

2.2 Cleaning Up Incorrectly Labeled Data

DL algorithms are quite robust to random errors in the training set. So it is probably ok to leave the errors as they are, and not spend too much time fixing them.

However, DL algorithms are less robust to systematic errors: for example, all white dogs labeled as cats.

A good approach is to conduct an error analysis (like in previous subsection) about incorrectly labeled data. In other words, for data examples that your classifier did well, but the actual data label was wrong. Then study if it is worthwhile to fix incorrectly labeled data. Often, it is worth it because you need to trust your dev set.

2.3 Build Your First System Quickly, Then Iterate

- Set up dev/test sets and metric
- Build initial system quickly
- Use bias/variance analysis and error analysis to prioritize next steps

2.4 Training and Testing on Different Distributions

Consider a situation where you have 200000 cat pictures from distribution A, and 10000 cat pictures from distribution B, but the distribution you really care about is distribution B. As a result, shuffling 210000 examples might not be a good idea because a small number of data examples from distribution B will get into your dev/test sets. Consequently, it is better if we assigned more data from distribution B to dev/test sets.

2.5 Bias and Variance with Mismatched data

Sometimes the difference between train error and dev error is a lot, but it does not necessarily mean that we have a variance problem. Maybe the distributions of train and dev sets are quite different. What to do?

Solution: Keep a *train-dev* set, which is of the same distribution as the train set, and it is not used for training. If the difference between train error and train-dev error is a lot, we do have a variance problem. However, if the difference is not a lot, the problem is called the *data mismatch problem.*

Another example:

Human Error	0%
Train Error	10%
Train-Dev Error	11%
Dev Error	20%

We clearly have an avoidable bias problem, as well as the data mismatch problem, but not a variance problem.

The general principles:

- Avoidable bias: huge difference between human error and train error
- Variance problem: huge difference between train error and train-dev error
- Data Mismatch: huge difference between train-dev error and dev error
- Overfitting to the Dev set: huge difference between dev error and test error

2.6 How to Address the Data Mismatch Problem?

- Carry out an error analysis, and try to understand the differences between training set and dev and test sets
- Make training data more similar; or collect more data similar to dev/test sets.
- Artificial data synthesis (e.g. add artificial noise, but with the caution that our model might overfit to the noise).
-

2.7 Transfer Learning

Two main steps:

- Pre-training: Using the weights of NN A to initial the weights of NN B.
- Fine-tuning: Using the training data to update weights (selectively (don't have to re-train all weights)).

It is especially useful for situations where there isn't enough training data available. Weights from complex networks can be used for recognizing basic features in initial layers. Needless to say, NN A and NN B must take similar inputs for their tasks in order to implement transfer learning.

2.8 Multitask Learning

Building a single neural network that does multiple things. For example, for a neural network corresponding to an autonomous car, it must be able to recognize many things such as pedestrians, cars, stop signs etc. As a result, we need a network that assigns *multiple* labels to single inputs because it is possible for an image to contain both a pedestrian and a car, for instance.

We might think that it would be better to train several neural networks for separate labels. But some of the basic features are shared between those networks, and it does not make sense to train multiple ones when multitask learning can even yield better results.

Another perk of using multitask learning is that labelers can put *don't care* values for labels of different clases.

When multitask learning makes sense:

- Training on a set of features that could benefit from having similar low-level features.
- Usually: amount of data you have for each task is quite similar.
- Can train a big enough neural network that performs well on all the tasks.