Part3. Structuring Machine Learning Projects

1. Principle of Orthogonalization
2. Human-level Performance
3. Elaborating ML Process
4. Learning Strategy

1. Principle of Orthogonalization

Why ML strategy?

You have a lot of ideas for how to improve the accuracy of your deep learning system: collect more data with more diverse training set, train algorithm longer with gradient descent, try different optimization algorithm (e.g. Adam) with bigger or smaller network, try dropout or L2 regularization, change network artchitecture with different activation functions with different number of hidden units and layers, etc.

This course will give you some strategies to help analyze your problem to go in which direction that will help you get better results.

Orthogonalization

- Some deep learning developers know exactly what hyperparameter to tune in order to try to achieve one effect. This is a porcess called orthogonalization. In orthogonalization, you have some controls each having a specific task and doesn't affect other controls.
- For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that 4 things hold true:
  - Have to fit training set well on cost function near human level performnace (if possible): bigger network, another optimization algorithm
  - Fit dev set well on cost function: regularization, bigger training set
  - Fit test set well on cost function: bigger dev set
  - Performs well in real world: change dev set, change cost function

Single number evaluation metric

	Predicted Cat	Predicted Non-cat
Actual Cat	3	2
Actual Non-cat	1	4

Precision: P = 3 /4

Recall: R = 3 / 5

Accuracy: (3+4) / 10

- Using a precision/recall for evaluation is good in a lot of cases, but separately they don't tell you which algorithm is better. A better thing is to combine precision and recall in one single (real) number evaluation metric. There is a metric called F1 Score, which combines them. $F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}}$
- It is sometimes hard to get a single number evaluation metric. We can solve that by choosing a single optimizing metric and decide that other metrics are satisfying.

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

Classifier	F1	Running Time
A	90%	80ms
B	92%	95ms

Train/Dev/Test set

1. Distribution

Dev and test sets have to come from the same distribution which reflect data you expect to get in the future and consider important to do well on. Setting up the dev set, as well as the validation metric is really defining what target you want to aim at.

2. Size

- An old way of splitting the data was 70% train, 30% test or 60% train, 20% dev, 20% test with smalle number of data.
- In the modern deep learning, if you have a million or more examples, a reasonable split would be 98% train, 1% dev, 1% test.

3. When to change a dev/test sets and metrics

Classifier	Predicted Cat
Algorithm1	3% error (A lot of porn are classified as cats)
Algorithm2	5% error

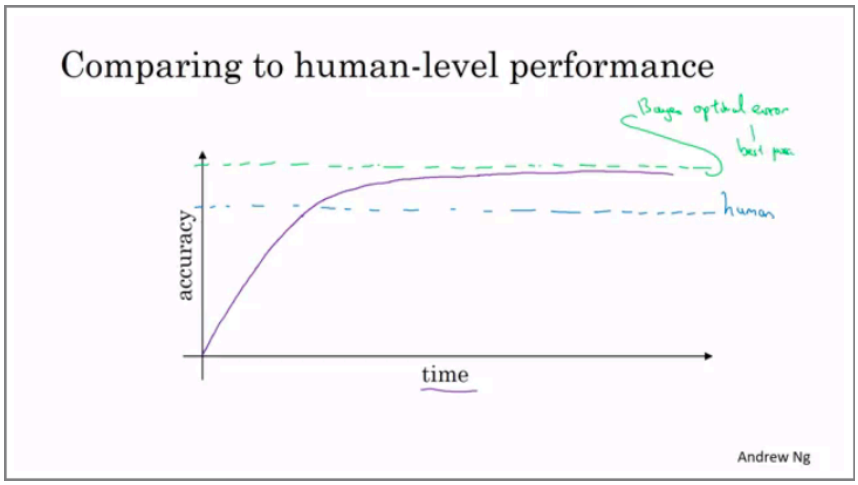
(New metric should be introduced.)

- Figure out how to define a metric that captures what you want to do: place the target.
- Worry about how to actually do well on this metric: how to aim/shoot accurately at the target.
- If doing well no your metric, but dev/test set doesn't correspond to doing well in your application, change your metric and/or dev/test set.

2. Human-level Performance

Why human-level performance?

- We compare to human-level performance because of two main reasons:
  - Thanks to dvances in deep 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.
  - It turns out that the workflow of designing and building a machine learning system is much more efficient when you're trying to do something that humans can also do.
- After an algorithm reaches the human level performance, the progress in accuracy slow down: it cannot surpass an error called "Bayes optimal error". There isn't much error range between human-level error and Bayes optimal error.
- Humans are quite good at a lot of tasks. So as long as machine leraning is worse than humans, you can get labeled data from humans, gain insight from manual error analysis: why did a person get it right?



Accuracy Progress toward Bayes Optimal Error

Aviodable bias

The human-level error as a proxy (estimate) for Bayes optimal error. Bays optimal error is always less (better), but human-level in most cases is not far from it.

Humans	1%	7.5%
Training error	8%	8%
Dev error	10%	10%
Problem	**Bias**	**Variance**

Avoidable bias = Training error - Human Error

Variance = Dev Error - Training Error

- If avoidable bias is big, then it's bias problem and you should use a strategy for bias resolving.
  - Train bigger model with longer/better optimization algorithm (like Momentum, RMS Prop, Adam)
  - Find better NN architecture/ hyperparameers search
- If variance difference is big, then you should use a strategy for variance resolving.
  - Get more training data
  - Regularization (L2, Dropout, Data augmentation)
  - Find better NN architecture/ hyperparameters search

Human-level performance

- When choosing human-level performance, it has to be chosen in the terms of what you want to acheive with the system.
- You might have multiple human-level performnaces based on the human experience. Then you choose the human-level performance (proxy for Bayes error) that is more suitable for the system you're trying to build.
- 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.

3. Elaborating ML Process

Carrying out error analysis

- Process of manually examining mistakes that your algorithm is making. Gvies you insights into what to do next. Helps you to analyze the error before taking an action that could take lot of time with no need.
- Sometimes, you can evaluate multiple error analysis ideas in parallel and choose the best idea. Create a spreadsheet to do that and decide.
- Quck 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.
  - In the cat classification example, if you have 10% error on your dev set and you want to decrease the error. You discovered tthat some of the mislabeled data are dog pictures that look like cats. Should you try to make your cat classifier do better on dogs? (This could take some weeks.)
  - Error analysis approach:
    - Get 100 mislabeled dev set examples at random.
    - Count up how many are dogs.
    - If 5 of 100 are dogs, then training your clasifer to do better on dogs will decrease your error up to 9.5%, which can be too little.
    - If 50 of 100 are dogs, then you could decrease your error up to 5%, which is reasonable and you should work on that.

Cleaning up incorrectly labeled data

Consider these guidelines while correcting the dev/test mislabeled examples:

- Apply the 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 right as well as ones it got wrong. (Not always done if you reached a good accuracy.)
- Train and (dev/test) data may now come from a slightly different distributions.
- It's very important to have dev and test sets to come from the same distribution. But it could be OK for a train set to come from slightly different distribution.

Build your first system quickly, then iterate

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

Training and testing on different distributions

- A lot of teams are working with deep learning applications that have training sets different from the dev/test sets due to the hunger of deep learning to data.
- There are some strategies to follow up when training set distribution differs from dev/test sets distribution.
  - Shuffle all the data together and extract randomly training and dev/test sets. (Not recommended)
    - Pros: all the sets now come from the same distribution.
    - Cons: the other (real world) distribution that was in the dev/test sets will occur less in the new dev/test sets.
  - Take some of the dev/test set examples and add them to the training set.
    - Pros: the distribution you care about is your target now.
    - Cons: the distribution in training and dev/test sets are now different. But you will get a better performance over a long time.

Bias and Variance with mismatched data distributions

- Bias and variance analysis changes when training and Dev/test set is from the different distribution.

Humans	0%
Training error	1%
Dev error	10%

You'll think that this is a variance problem, but because the distributions aren't the same, you can't tell for sure. It could be that the train set was easy to train-on, but the dev set was more difficult.

Humans	0%
Training error	1%
Train-dev error	9%
Dev error	10%

To solve this issue, we create a new set called train-dev set as a random subset of the training set (same distribution). Now we are sure that this is a high variance problem.

Humans	0%
Training error	1%
Train-dev error	1.5%
Dev error	10%

In this case, we have Data mismatch problem.

- Unfortunately, there aren't many systematic ways to deal with data mismatch problem. There are some things to try about this in the next section.

Addressing data mismatch

There aren't completely systematic solutions to this, but there are things you could try.

- Carry out manual error analysis to try to understand the difference between training and dev/test sets.
- Make training data more similar, or collect more data similar to dev/test sets.
  - **Artificial Data Analysis**
    - Combine some of your training data with something that can convert it to the dev/test set distribution.
    - 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 because your NN might overfit these generated data.
      - Combine normal audio with car noise to get audio with car noise example
      - Generate cars using 3D graphics in a car classification example

4. Learning Strategy

Transfer learning

- Apply the knowledge you took in a task A and apply it to another task B. For example, you have trained a cat classifier with a lot of data, you can use the part of the trained NN to solve x-ray classification problem.
- To do transfer learning, delete the last layer of NN and its weights and
  - If you have a small data set- keep all the other weights as a fixed weights. Add a new last layer(s) and initialize the new layer weights and feed the new data to the NN and learn the new weights. (**Fine-tuning)**
  - If you have enough data, you can retrain all the weights. **(Pretraining)**
- When transfer learning makes sense:
  - Task A and B have the same input X (e.g. image, audio)
  - You have a lot of data for the task A from which you are trasnferring, and relatively less data for the task B.
  - Low level features from task A could be helpful for learning task B.

Multi-task learning

- 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 to do several things at the same time. Then each of these tasks hopefully helps all of the other tasks.
- When multi-task learning makes sense:
  - 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 network to do well on all the tasks. (If you can train a big enough NN, the performance of the multi-task learning compared to splitting the tasks is better.)
- Today, transfer learning is used more often than multi-task learning.

End-to-end deep learning

Some systems have multiple stages to implement. An end-to-end deep learning system implements all these stages with a single NN.

To build the end-to-end deep learning system that works well, we need a big dataset. (more data than in non end-to-end system.) If we have a small dataset, the ordinary implementation could work just fine.

- 1. Speech recognition system
  - Audio → Features → Phonemes → Words → Transcript
  - Audio → Transcript
- 2. Face recognition system
  - Image → Face detection → Face recognition # best approach
  - Image → Face recognition
  - In the first implementation, there are two steps where both parts are implemented using deep learning. It works well because it is harder to get a lot of pictures with people in front of the camera than getting faces of people and compare them.
- 3. Machine translation system
  - English → Text analysis → … → French
  - English → French # best approach
- 4. Estimating child's age from the x-ray picture of a hand
  - Image → Bones → Age # best approach
  - Image → Age

Whether to use end-to-end deep learning

- Pros
  - Let the data speak. By having a pure machine learning approach, your NN 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.
  - Less hand-designing of components needed.
- Cons
  - May need a large amount of data.
  - Excludes potentially useful hand-design components (it helps more on the smaller dataset).
- When to apply end-to-end deep learning
  - Key question: Do you have sufficient data to learn a function of the complexity needed to map X to Y?
  - Use ML/DL to learn some individual components.
  - When applying supervised learning, you should carefully choose what types of X to Y mappings you want to learn depending on what task you can get data for.