Course 3, week 1: ML Strategy (1)

Introduction to ML Strategy

Why ML Strategy

- Lets say you have a classifier which has 90% accuracy, but this percentage is not good enough for your application.
  - we have a couple of ideas:
    - collect more data
    - collect more diverse data
    - train algorithm for longer
    - try another optimizer
    - change network size
    - try dropout
    - add regularization
    - Change the network architecture

*how do we know which endeavour will actually improve our models and which one is a waste of resources.

- ML strategy aims to teach the criterion used to take on these decisions.

Orthogonalization

- the process of orthogonalization refers to knowing what to tune to achieve a desired effect.
  - Learn tuning "knobs" is designed in such a way that it only has a well defined effect/function.
- this tuning process relates to machine learning, supervised learning system, as we usually want to tune

the system to make sure that four things happen,

i. Fit the training set well on cost function

└ knobs used to tune: bigger network, optimization algorithm (such as adam) ...

ii. Fit the dev set well on cost function.

└ knobs used to fit properly: Regularization, bigger training set

iii. Fit test set well on cost function

└ knobs to tune: Getting a bigger dev set cuz might have overfitted to our dev set if it doesn't do well in the test set.

iv. Performs well in real world application

└ If the model doesn't perform desirable results at this step, we might have to change the cost function or tune the dev set. Poor performance might mean that the cost function isn't measuring the right thing for our specific problems.

*Training or early stopping isn't recommended by Andrew as it's a knob that affects the training set by measuring it against the dev set performance.

└ less orthogonalized method.

Setting up your Goal

Single # Evaluation Metric

- Precision: how many selected items are relevant
- Recall: how many of the relevant items are selected.

*There's often a tradeoff between precision and recall when optimizing a model (however, we want both to do fairly well).

- The problem with precision and recall is that if one model has higher precision and the other has higher recall, how do we quantify which model is better? With two evaluation metrics, it's hard to know which one of multiple models being evaluated is overall doing better.
  - This issue can be corrected through the usage of F1 scores.

- The F1 score can be thought of as a weighted average of both precision and recall.
  - $F_1 = \frac{2 \cdot (\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$ "harmonic mean"

- Having a well defined dev set + a single evaluation metric helps speed up iterating.

Satisficing and optimizing metrics

- It's not always easy to synthesize an evaluation metric from various performance criteria (like accuracy and running)
  - In these cases it is useful to set up satisficing and

optimizing metrics.

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

- In this example, we want a metric which can both encompass the run-time and accuracy
  - cost = accuracy − 0.5 · run-time
    - we could try a metric like this one, but it's subjective to our choices and definitions.
- alternatively, we can specify a classifier that can maximize the accuracy, but it's also subject to a maximum run-time (such as 100 milliseconds)
  - in this case, accuracy is an optimizing metric (what we are trying to maximize), while the run-time is a satisficing metric (it just needs to meet the requirement and we don't really care about by how much).
    - with this criterion, B would be the best classifier.
- More generally, if we have N metrics that we care about it's most reasonable to pick one as optimizing and n−1 to be satisficing (threshold based).

Train/dev/test distributions

- the way we set up these sets can have a huge impact on how rapidly we can make progress on an application.
- In the ideal case the development/cross-validation set should come from the same distribution as the test set.
  - we utilize the dev set + metric to quantify our performance on the problem and iterate. we want a test set of similar distribution so that the work done to optimize to the dev set translates into performance gains in the test set.
- General guideline, choose a dev and test set to reflect the data that you expect to get in the future and want to do well on.

Size of dev and test sets in deep learning Era

- old way of splitting data: 70/30, train/dev set split or 60/20/20, train/dev/test split
  - this was pretty reasonable in the earlier era of machine learning as dataset sizes were pretty small (hundreds of examples to ten thousand examples).
- In the deep learning era, we might have 1 million examples in the dataset.
  - hence it's more reasonable to do a 98/1/1, train/dev/test split.
- the size of the test set should be large enough to

give high confidence in the overall performance of the system.

- for some applications, we might not need a high confidence in the overall performance of the final system.
  - in these we may only need a train/dev set split.
    - this is a bit unusual and not generally recommended as we want a separate test set to obtain an unbiased estimate of the performance of the estimator.

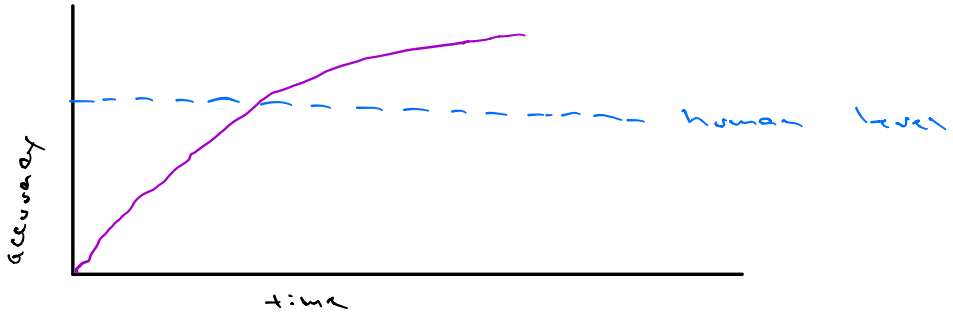
When to change dev/test sets and metrics

- when the evaluation metric is no longer correctly ranking ordering the users preferences between algorithms, then that's a sign that we should change our metric or dev/test set.
- if doing well on your metric + dev/test set does not translate to doing well on your application, change the metric and/or dev/test set.

Comparing to human-level performance

Why Human-level Performance?

- because of advances in deep learning, algorithms are working much better. This has made them competitive with human level performance in many more areas.
  - additionally, the workflow of building a project is much more efficient when we try to do something that humans can also do. Choosing it natural to compare it to human level performance.



- often when working on a task for some time, progress tends to be rapid as we approach human level performance. After surpassing human level performance, the rate of improvement goes down. The algorithm approaches but never surpasses some theoretical limit, known as the **Bayes optimal error** (the best possible achievable error).
  - for example in cat recognition, some images are so blurry that it might be impossible to correctly classify that image. Hence, the perfect level of accuracy might not be 100%.
    - therefore, the bayes optimal error is the best theoretical mapping function that can never be surpassed for the task.
- why does progress slow down when we pass human level performance?
  1. human level performance for many tasks is not that far from the bayes optimal error. By the time we surpass human level performance, there's not much head room.

- So long as our performance is less than human level performance, there's certain tools we can use to improve performance. These are harder to use once we've surpassed the level. These things include,
  - getting labeled data from humans
  - manual error analysis: why did a person get this right? Getting insights on what algo is getting wrong.
  - better analysis of bias / variance

Avoidable bias

- By knowing where human level performance is, we can know the general area where the algorithm performs well, but not too well on the training set.
- Cat classification Example
  - Humans: 1% — 7%
  - training error: 8% — 2%
  - dev error: 10%

the fact that there's a huge gap between the train error and human level tells us that the algorithm isn't even fitting the train set well.

* Hence, here we can focus on reducing bias.
  - train a bigger neural network or run gradient descent for longer.

- different classification example
  - humans: 7.5%
  - training error: 8%

avoidable bias ~ 0.5%

↳ Dev error: 10%

} variance ~ 2%
room more room to reduce the variance

* in this case, even though the error is the same as in the earlier example, maybe we are doing fine on the training set (just a bit worse than human level performance). In this example, we might want to focus on reducing the variance between the train and dev sets. Increasing dataset or using regularization, etc...

* in these examples, we are thinking about human level performance as a proxy for the bayes optimal error. (this is a pretty good proxy in CV tasks)

- the difference between the bayes error and the train set is known as the avoidable bias. The difference between the train and dev is known as the variance.

Understanding human level performance

- how can we define human-level performance more precisely?

Surpassing human-level performance

- once you surpass the human performance threshold, your options of making progress on the problem are less clear.
- Problems where ML significantly outperforms humans,
  - ↳ Online advertising
  - ↳ Product recommendation
  - ↳ Logistics (predicting transit times)

Structured data problems

└ loan approvals

] Most natural perception problems) harder for computers

Improving your model performance

- the two fundamental assumptions of supervised learning
  i. you can fit the training set pretty well.
     └ how avoidable bias can be achieved through training
  ii. the training set performance generalizes well to the dev/test set.
     └ we are saying that variance is not too bad.

- reducing (avoidable) bias and variance

- human level
- avoidable bias → • training bigger models
  • train longer and use a better optimization algorithm (momentum, adam)
  • better NN architecture/ hyperparameters search
- training error
- variance → • more data to train on
  • regularization techniques (data augmentation)
  • better NN architecture/ hyperparameter search
- Dev error