Course 3: Structuring Machine Learning Projects

Content

1. Introduction to ML Strategy

Why do we need a Machine Learning strategy? Let's suppose we are developing a model and we achieve a 90% accuracy, but for some reason it is not high enough. We could do de following:

- Collect more data
- Collect more diverse training set
- Train algorithm longer with GD
- Try Adam instead of GD
- Try bigger/smaller network
- Try dropout
- Add L2 regularization
- Network architecture (activation functions, number of hidden units)
- And so on…

Turns out we have a lot of choices that we could do.

In this course we will see strategies or easy of analyzing a machine learning problem so that we will point in the direction of the most promising things to try.

We will make use of **orthogonalization.** In this context, this means tuning a single "knob" at a time to achieve a single effect. The driving example is clear: on cars, we have a control for steering and a different control for speed. If we had these two controls integrated in a single one, it would be much more difficult to drive, although we would still be able. But how does this relate to ML?

Let's see the chain of assumptions that we take in a ML project and see how we could meet them. We will get into deeper detail later.

- Fit training set well on cost function: if we don't, we should for example try a bigger network, or changing the optimizer.
- Fit dev set well on cost function: if we don't, we probably have an overfit model so we should try regularization or getting a bigger training set.
- Fit test set well on cost function: if we don't, we should try a bigger dev set since we have probably overtuned the dev set.
- Performs well in real world: if it doesn't, we could change dev set or the cost function.

We can see that for each stage, we are changing different things. In addition, it is not very much recommended to use early stopping since it simultaneously affects how well we fit the training and dev sets. Thus, this knob is "less orthogonalized". It still can be used, but it will make our tuning process less systematic.

So the aim of this course is to establish a systematic way of training a model, being able to identify which "piece" of the model is doing wrong and knowing which knob we can tune so as to improve it.

2. Setting up our goal

Single number evaluation metric

When tuning hyperparameters, our progress will be much faster if we have a single real number evaluation metric. This will help in problems such as classification with precision and recall. Calculating the F1 score will help us to get the best model (assuming the best model is the one which balances precision and recall).

Satisficing and Optimizing metric

In cases where it is hard to combine everything we care about into a metric, this can help us. For example, if we care about classification accuracy and running time, and we need this last one to be under a given threshold, we can just discards the models that train for too long and then choose the one with the highest accuracy. In this example, the accuracy would be an *optimizing* metric and the running time a *satisficing* metric.

But these metrics need to be calculated on a training/dev/test set. So we will next see how set up these.

Train/Dev/Test distributions

The way we set up our sets can have a huge impact on our efficiency.We should always try that our sets come from the same distribution. We should follow this guideline:

Choose a dev set and test set to reflect data you expect to get in the future and consider important to do well on.

In addition, we should ensure that at least dev and test sets come from the same distribution. And why not also the training set? Setting the dev set and the metric of evaluation will help us define the target (bullseye) we are aiming at. So it doesn't make sense if we have a target on dev set and a different one on test set. However, training set will only determine how well will we get to the bullseye.

Size of the dev and test sets

It turns out that the guidelines regarding to the size of the dev and test distributions are changing in the deep learning era.

The old 70/30 or 60/20/20 rules of thumb made sense when we had small datasets.

But if we had a million data points, it would seem reasonable to have a 98/1/1 distribution, since 10.000 different data points on the test set should be (depending on the application) a representative sample.

In addition, and although it is not completely recommendable, sometimes we won't need a test set. For example, if we know we are deploying a model no matter what

accuracy we get, since we are just deploying it, we might not need an unbiased estimator of that accuracy.

When to change dev/test sets and metrics

If the error metric is leading us to a model that is not the best for our purposes, it may be because that metric doesn't reflect correctly what is "best". For example, if we have a cat/non-cat classifier with accuracy as a metric but we have pornographic images, which are unacceptable for us. A classifier with low accuracy could show lots of pornographic images. So in cases like this, we should re-define or error metric (for example, including some sort of weight for pornographic images.

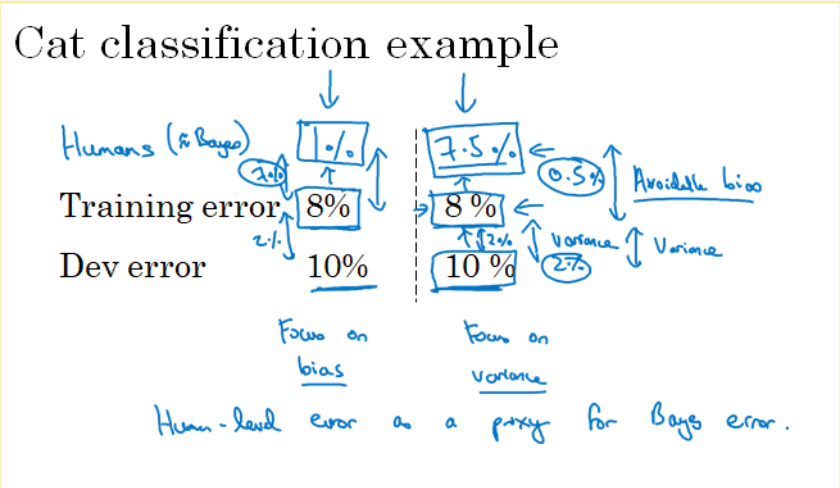
Another example could be if we trained our model with high resolution images but in the clat classification mobile app the users take blurry photos. This means we have different distributions between sets and we should try to make them match as much as possible.

3. Comparing to human-level performance

The intuition here is straightforward: we should keep in mind the human-level performance when training or model. For example, if we assume a human can classify cats with a 1% error and we are getting 8% training error and 10% dev test, we should try not to underfit because we still could do better to achieve human performance. But, if this human level performance was 7.5%, then perhaps we aren't doing that bad and we should focus on lowering the dev error.

This is, depending on human-level performance we should do different tasks when training our model. We should always keep this in mind.

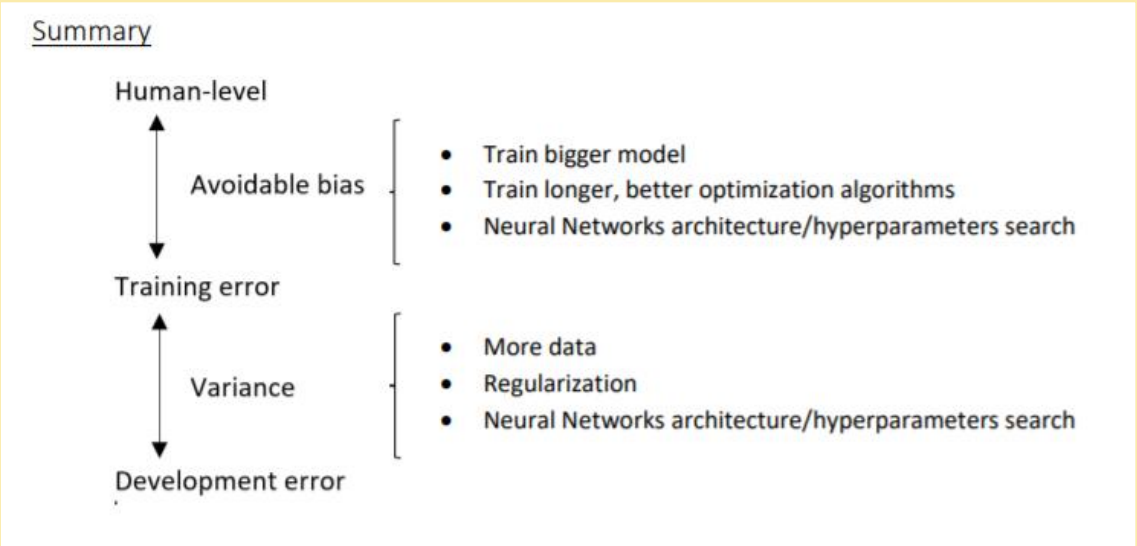
An important concept here is the Bayes optimal error, which is (in a basic way) the lowest error we could get. We will take human-level error as a proxy for the Bayes error in certain tasks.



This is really important: having an 8% accuracy in a problem where the best accuracy is 7.5% is a really good measure, whereas having in a 1% problem is not.

But what happens when we surpass human-level performance? On these cases it is much difficult to gain insights on where to focus (bias or variance).

This slide packs up the whole process of improving our model:



4. Error analysis

When we are trying to improve our model performance, we can carry out error analysis. This consists of randomly looking at a number (i.e. 100) of images and see what kind of misclassifications are being done boy our model. Doing this before making any effort is advisable as we can aim at solving problems that affect more data points and therefore reduce our error fastly.

Another thing that can happen to us is the existence of **incorrectly labeled data**. When we have this problem on the training set, we can say that deep learning systems are robust to random labeling errors. But if we have a systematical error in the labeling process then the model will learn from it.

If we have incorrectly labeled data on the dev/test set, we should try to quantify the source of error coming from this and then decide whether or not to focus on fixing it.

One important take away is that we should carry out an error analysis and manually explore a number of data points in order to gain insights on what is happening.

Finally, if we are working on a new machine learning application, one recommendation is to **build our first system quickly, then iterate**. This means setting up a dev/test set and a metric, training the model and then using bias/variance analysis and error analysis to prioritize next steps (because at the

first steps on model training, there can be many directions that we can follow in order to improve our results).

5. Mismatched training and dev/test set

Let's suppose we have the following situation: we are training a cat/non-cat classifier for a mobile app and we have data from the web (high resolution, well framed pictures) and data from the mobile app (less quality photos, blurred, etc…). We have to decide our train/dev/test set splits. ¿How can we do that?

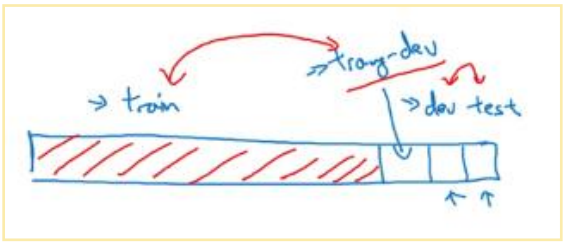
- The first option is to randomly distribute the images across the three sets. This has a main advantage: all sets have the same data distribution.
- The second option is to only have mobile app images on the dev and test sets, and train with the webpage images and a small number of mobile images. Because we want our model to classify images in an app, and therefore we will feed it with only mobile app images when deployed, this would be the better option.

So we can see that having mismatched distributions between training and dev/test data may not be a problem. But we **do** have a problem if dev and test distributions are different. Intuitively, we are training our model to aim at an objective and then moving the objective in deployment.

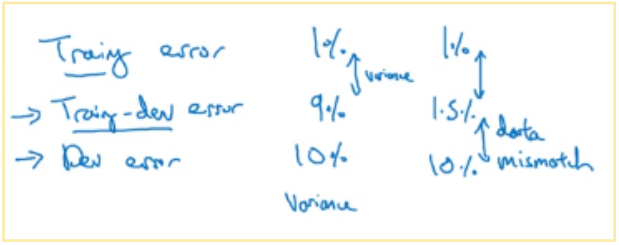
When having different distributions between training data and dev/test data, the way we analyze bias and variance changes. If we use the previous example, suppose we have a 1% training error and a 10% training error. This can be caused by two effects:

- High variance
- Because the images in the training set were different (in terms of distribution) than the ones in the dev set, this can cause this error and not strictly due to variance.

So, because we are having two effects, it is more difficult to know which one of them is present. One thing we can do is get a training-dev set (which has the same distribution as the training set but is not used for training):



Now we can see whether we have or not a variance problem (if train error is really smaller than training-dev error, then we have a variance problem).



We can also combine this analysis with bias analysis.

In conclusion, along with bias and variance, we can now have a third source of error: data mismatch. And, let's suppose we conclude that data mismatch is a huge source of error. How do we address it?

Addressing data mismatch

Once we have identified we have a data mismatch problem, what can we do? We can carry out manual error analysis to try to understand differences between distributions, or make training data more similar, or collect more data similar to dev/test sets.

One way to ndo this is **artificial data synthesis.** This means artificially making the data more similar. For example, if we are in a speech recognition problem and we have noise in the background, we could add this noise to the data that doesn't have it.

6. Learning from multiple tasks

One of the most powerful ideas in deep learning is that sometimes we can take knowledge a neural network has learned from one task and apply that knowledge to a separate task. This is called transfer learning.

Transfer learning

The idea is simple. We can reuse an already trained neural network and substitute the last layers and the training dataset. Then we can choose to train the whole neural network or only the last layers (and therefore maintain the pre-trained weights values).

In fact, this is simply using trained weights instead of randomly chosen ones, and finishing training them in a new dataset and a partially new net architecture.

For example, in image recognition this can be useful because the low-level features (present in the early layers) like detecting edges may be common on both domains.

And when does this make sense?

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

Multi-task learning

In transfer learning we have a sequential process where we learn from task A and then transfer that to task B. Multi-task learning can be useful when we have multi-label problems. For example, when detecting entities on video we could train separate nets to detect every simple entity, but we can train a single one to detect all the entities.

When does it make sense?

- Training on a set of tasks that could benefit from having shared lower-level features.
- The amount of data we have for each task is quite similar.
- We can train a big enough network to do well on all the tasks.

7. End-to-end deep learning

When we talk about end-to-end deep learning we refer to substituting the whole process of getting a prediction by a single neural network. For example, in speech recognition, we could manually extract the words, and then the phonemes, and then perform a prediction based on them. It turns out we could substitute this manual process by a single neural network. But this does not always work. Some pros and cons of end-to-end deep learning are:

- Pros:
    - Let the data speak
    - Less hand-designing of components needed
- Cons:
    - May need large amount of data
    - Excludes potentially useful hand-designed components

The key question we have to ask ourselves is whether or not we have sufficient data to learn a function of the complexity needed to map x to y.