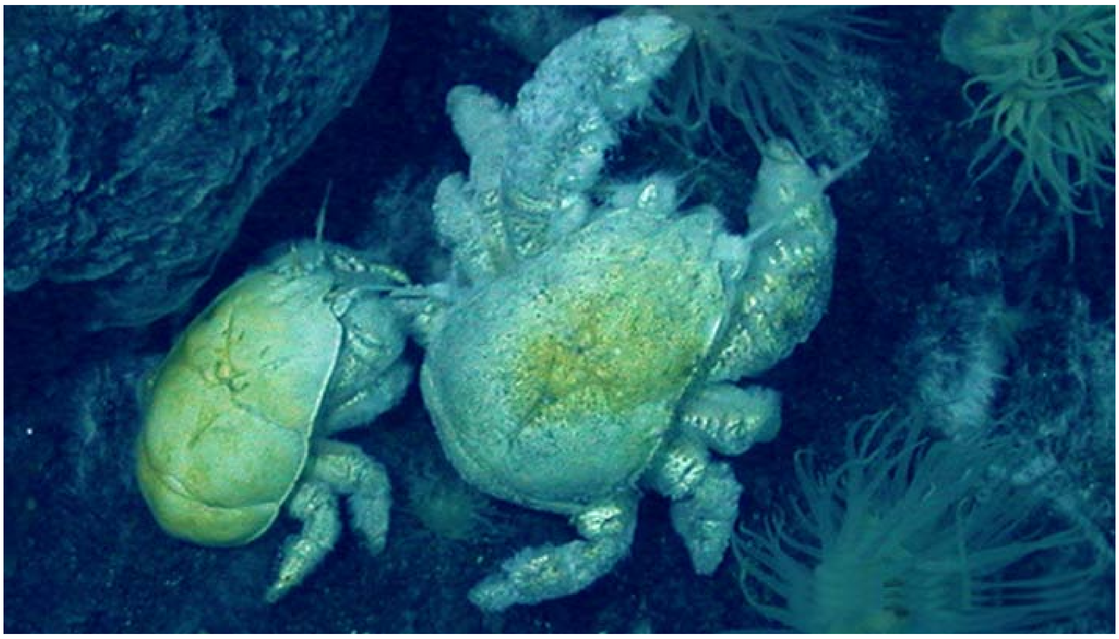
Structuring Machine Learning Projects

Notes by José A. Espiño P. [1]

Summer Semester 2022–2023



[1] The content in these notes is sourced from what was covered in the MOOG the document is named after. I claim no autorship over any of the contents herein.

Contents

1 Machine Learning Strategy

Machine Learning Strategy refers to the process of choosing the best hyperparameters for a given problem. In this document, we will cover diverse ways we can achieve this — and how to choose which way is the most efficient. **Orthogonalisation** is a process that involves isolating different hyperparameters and tuning them independently. This is useful because it allows us to focus on one thing at a time, and thus to make the most of our time. Imagine the process of having to tune all the hyperparameters at the same time — that would render the process so much more painful. For a supervised system to work well, we need to make sure four things hold true:

- **Fit training set well** — low bias
- **Fit dev set well** — low variance
- **Fit test set well** — low variance
- **Perform well in real world** — low avoidable bias

Orthogonalisation comes into play here because if, for instance, you want to reduce the bias, you choose some methods such as creating a bigger network or another optimiser. Similarly, if you want to reduce the variance, you can use regularisation or get more data. These methods are orthogonal insofar that they do not impact (much) the other three things we are paying attention to in this list. A method such as early stopping is not as desirable because it can impact how the model fits both the training and the validation set, which is not ideal.
At many different stages of the model tuning process, it is important to have a **single number evaluation metric**, a well–defined numerical measure of how well the model is doing. Using this metric, we can compare the model in question with different other models and assess which one is performing best.
Two common metrics are the **precision** and the **recall**. The precision is defined as the number of true positives divided by the number of true positives plus the number of false positives while the recall is defined as the number of true positives divided by the number of true positives plus the number of false negatives. The problem arises when we are comparing n

models, where some perform better in recall and others perform better in precision — how do we choose which one is better?
Here is where a *single value* metric becomes helpful. In this specific case, we would use the **F1 score**, the harmonic mean of the precision and the recall. Since this score is a single value, it helps us directly compare models.
There are other important metrics used to assess the performance of a model, such as the **ROC curve** and the **Area Under Curve (AUC)**. The ROC curve is a plot of the true positive rate (recall) against the false positive rate (1 - precision) for different values of the threshold. The AUC is the area under the ROC curve.
It is not always easy to combine all the metrics you want to evaluate into a single value. In those cases, we can use two metrics, an **optimising metric** and a **satisficing metric**. The former is the one we want to improve constantly, and the latter is the one we want to keep above/below a certain threshold. For instance, we can make accuracy be our optimising metric and the running time of the model be our satisficing metric: as long as all the models we are comparing run below a certain time, we can compare them against each other based on their accuracy.
These evaluation metrics can be calculated on the training, development, or test sets. The way you set these datasets can have a big impact on the efficiency of the machine learning application building process. The general rule of thumb is having the development set and the test set sourced from the same distribution.
The size of the dev and test sets are also essential. If the dev set is too small, it will be hard to evaluate the model. If the test set is too small, it will be hard to evaluate the model's performance in the real world. You should set the test set to be big enough to give high confidence in the overall performance of the system, while the dev set should be big enough to evaluate the different iterations of the model.

2 Human Performance

The workflow of designing machine learning algorithms is a lot smoother when you are trying to do something humans can also do. An important concept in this regard is **Bayes optimal error**, the lowest possible error rate for a given problem — there is no way to surpass this error rate. For instance, in an audio recognition algorithm, there are some sounds that are so similar that even humans cannot tell them apart. In this case, the Bayes optimal error is the error rate of a human.
Usually, by the time the model's accuracy surpasses human level performance, the improvement to the model's accuracy will become significantly slower. At this point, you may want to weight if it is better to stop the training process or to aim for the Bayes optimal error.
In some cases, the Bayes optimal error is not known. In these cases, we can use the **proxy** of human level performance. For instance, if we are trying to build a speech recognition system, we can use the word error rate of a group of humans as a proxy for the Bayes optimal error.
Human–level performance is not a fixed number. It can vary depending on the human performing the task. For instance, a professional radiologist will have a lower error rate than

a general practitioner when diagnosing a patient. In cases like these, we can use the **proxy** of the **best** human performance. This is not a strict definition; depending on the context in which your application will be used, you can define human level performance in different ways. In a model, the variance is the difference between the training error and the dev error. The bias is the difference between the dev error and the Bayes optimal error.
If the training error is low and the dev error is high, the model is overfitting. If the training error is high and the dev error is high, the model is underfitting. If both the training error and the dev error are high, the model is underfitting. If both the training error and the dev error are low, the model is fitting well.

3 Error Analysis

Whenever you are not satisfied with the performance of a model, see which samples the model is getting wrong and try to understand why. This process is called **error analysis**. If you see a high percentage of the same kind of error amongsty the samples, you can focus on that specific error and try to fix it. Sometimes, several different kinds of mistakes come up. It is important to analyse them and prioritise which error is worth tackling first.
Sometimes, it is possible that some of our samples are mislabeled. Our way of handling this depends on where the error is found. If it is a random error in the training set, there is no need to fix it; deep learning algorithms are robust to random errors. One caveat to this is that if the error is not random, but rather systematic, that may create big problems and lead to a high error rate, so they should be fixed. If it is a random error in the dev/test set, we should assess if it makes an impact in the way we evaluate the model. If that is the case, we can try to fix it by manually correcting the labels. If it does not make an impact, we can leave it as it is.
In order to maximise efficiency, it is recommended to build the first version of the model as fast as possible and then use error analysis to decide what to do next.

4 Mismatched Training and Dev/Test Set

Deep learning algorithms have hunger for training data. Because of this, very often teams strive to feed the model as much data as possible, even when this data might come from different distributions than the dev/test set. When this is the case, there are certain subtleties that need to be taken into account.
Firstly, we need to make sure that the dev/test set have the same distribution **and** that this distribution is the one we are aiming for — otherwise, we are optimising for the wrong thing. Sometimes it might not be advisable to use all the data you can get your hands on. For instance, when analysing model performance, a reason behind a gap between training error and dev error might be that the training set is not representative of the dev set. In this case, adding more data to the training set will not help.
In order to deal with this, we can split the training set into two parts: a training set and a **training-dev** set. The latter should have the same distribution as the training set. We can then use the training set to train the model and the training-dev set to evaluate the model. If

the training error is high compared to the Bayes optimal error, we have an underfitting issue (bias). If the training error is low and the training-dev error is high, the model is overfitting (variance problem). If both the training error and the training-dev error are high, the model is underfitting. If both the training error and the training-dev error are low, the model is fitting well. In a case where the training and training-dev errors are low, but the dev error is high, we have a data mismatch problem. There a different ways to address data mismatch. The first one is to carry out manual error analysis and try to understand the differences between the training set and the dev set. Once we have identified the differences, we can try to make the training set more similar to the dev set. There are several techniques we can use to do this, such as artificial data synthesis, or collecting more data. There is one note of caution when it comes to data synthesis: there is a chance that your algorithm will overfit to the specific synthetic data you have created. For instance, if we are building a speech recognition system for a car, we must be careful not to overfit to the specific sounds of the car we are using to create the synthetic data by using different cars to create the synthetic data.

5 Learning from Multiple Tasks

One of the most powerful ideas in deep learning is **transfer learning**, which consists in taking a model that has been trained for a task and reusing it for a different task. This is possible because the first layers of a deep learning model learn very general features that can be applied to different tasks. The way this is typically done is by deleting the last layer of the model and replacing it with a new one that is specific to the new task. This new layer will be trained from scratch, while the rest of the model will be *frozen*.
In some cases, it is possible to train the whole model, but this is only advisable when the new dataset is large enough.
Another way to use transfer learning is to use the weights of the first layers of a model that has been trained for a task and use them to initialise the weights of a new model that will be trained for a different task. This is called **pre–training**. Similar to pre–training, finetuning consists in using the weights of a model that has been trained for a task and using them to initialise the weights of a new model that will be trained for a different task. The difference is that in finetuning, we will not only initialise the weights of the first layers, but also the weights of the last layers.
In order to decide whether to use transfer learning or not, we need to take into account the following factors:

- How much data do we have? When we have a small dataset for the new task, but a lot of data for the task the model has been trained for, it is advisable to use transfer learning.
- How similar is the task we are trying to solve to the task the model has been trained for? If the tasks are very different, it is not advisable to use transfer learning.
- How similar is the input data to the input data the model has been trained for? If the input data is very different, it is not advisable to use transfer learning.

- How similar are the low–level features we are trying to extract to the low–level features the model has been trained for? If the low–level features are very different, it is not advisable to use transfer learning.

A related concept to transfer learning is **multi–task learning**, which consists in training a model for multiple tasks at the same time. This is possible because the first layers of a deep learning model learn very general features that can be applied to different tasks.
The loss function for multi–task learning is the sum of the loss functions for each task, namely:

$$J = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} J_j$$

where n is the number of tasks.
Even if some samples only have a label for one of the tasks, we still need to include them in the loss function. When we do that, we just omit the unknown labels from the loss function.
Multi–task learning makes sense when:

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

6 End–to–end Deep Learning

End–to–end deep learning is a design philosophy that consists in building a system that learns all the steps of a pipeline automatically. As opposed to traditional machine learning, where we would have to manually design the features, end–to–end deep learning allows us to learn the features automatically.
One of the challenges of this technique is how much data it needs to perform on a satisfying capacity. In order to decide whether to use end–to–end deep learning or not, we need to take into account the following factors:

- Do we have sufficient data to learn a function of the complexity needed to map x to y? Sometimes it might be helpful to break down our task into smaller subproblems and solve them separately. For instance, when it comes to face recognition in a picture, we might want to first detect the face and then recognise it. This is because we have a lot of data for face detection, but not so much for face recognition.
- Do we have a reasonable way of mapping x to y?

Some of the benefits of applying this technique are:

- Let the data speak.
- Less hand–tuning of components needed.With end–to–end deep learning, we can learn features automatically, which means that we do not need to spend time manually designing them.

- Easier to iterate quickly.

Similarly, some of the drawbacks of applying this technique are:

- May need large amount of data.
- Excludes potentially useful hand–engineered components. These are particularly useful when we have little data.