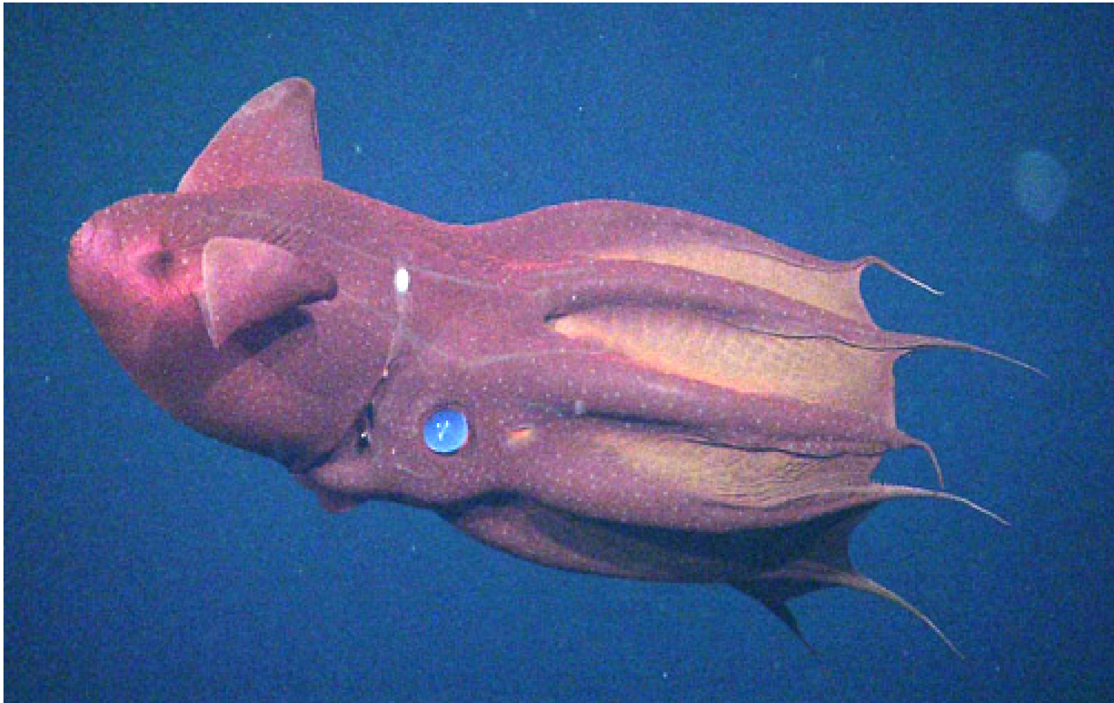
Improving Deep Neural Networks

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 Practical Aspects of Deep Learning 2
2 Optimisation Algorithms 5
3 Hyperparameter Tuning 8
4 Batch Normalisation (BN) 8
5 Multiclass Classification 9

1 Practical Aspects of Deep Learning

Machine Learning is all about decisions: the right set up for training/test/validation sets, amount of layers, amount of hidden units per layer, learning rate, activation functions per layer, etc. These decisions are not universal, but rather depend on factors like the number of input features, type and amount of data, whether we are training on GPU or CPU, and many other things.
These factors are chosen in a mostly iterative process in which we experiment with different values and see what works; this can take a long, long time. Efficiency is determined by how quickly we can go through this process, and the way we set up our data can improve that speed.
Traditionally, we divide our data in three sections:

- Training Set, which is used to train algorithms
- Hold–out/Cross validation/Development Set, which is used to see which of many different models performs best.
- Test Set, which gives us an *unbiased* estimate of how well the algorithm is doing

The percentage of the data that will be allocated to each of these three sections depends on the amount of data itself; the larger the amount of data, the higher the percentage that will be allocated to the training set. This is because the goal of the validation set is to test different algorithms on it and see which one works better; you do not need too many samples for that. In a similar vein, the goal of the test set is to give you a confident estimate of how your end classifier is performing — no need to get a high number of samples.
A dataset with $\tilde{1}$0k samples could perfectly have a 60/20/20 split. However, if the dataset size climbs up a couple degrees of magnitude, it is more advisable to have a split that looks like 99/0.5/0.5.
Another important consideration to keep in mind when setting these sets is to make sure that the samples from the validation and test sets come from the same source(s). For instance, in a Computer Vision application, our algorithm may prove ineffective if the validation set is comprised of high–resolution pictures crawled off the internet, whereas the training set is comprised of humble low–resolution user pictures. In contrast to this, deep

learning algorithms have a huge hunger for training data. The larger the training set, the better the algorithm's performance. A lot of creative tactics are employed in this process, such as cropping, tilting, crawling webpages, etc.

Two terms are essential when analysing the performance of a neural network: **bias** refers to underfitting, a situation where the model is bad at predicting. On the other hand, **variance** refers to overfitting — your model doing too well when predicting to the point that it cannot generalise properly and predict new samples correctly.
We can spot these two metrics by taking a look and comparing the training set error and the validation set error: if the train set error is low, but the validation set error is high, we know the model is overfitting — meaning it has high variance. Similarly, if even the train set error is high, the model has high bias and is thus underfitting the data.

Our approach must adapt to the issue our model is presenting:

- High bias:
  Try a bigger neural network, train it for longer time, find a new neural network architecture
- High variance:
  Get more data, regularisation, find a new neural network architecture

Regularisation is a helpful method to deal with high variance. One of the most common regularisation techniques is called **L2 regularisation**, which involves adding a regularising term to the cost function used in the neural network. For instance, this would look like:

$$J(w^{[1]}, b^{[1]}, ..., w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{n} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||w^{[l]}||^2$$

Where

$$||w^{[l]}||^2 = \sum_{i=1}^{n^{[l]}} \sum_{j=1}^{n^{[l-1]}} (w_{i,j}^{[l]})^2$$

This matrix norm is called the *Frobenius norm.* L2 regularisation is also called *weight decay* because in practise it is the same as multiplying the W matrix by a number smaller than 1, so that it also gets smaller progressively. In every iteration of gradient descent, this is seen as:

$$w^{[l]} = w^{[l]} - \frac{\alpha\lambda}{m} w^{[l]} - \alpha \text{ value obtained from backpropagation}$$

Regularisation aids in the reduction of bias because essentially the regularising term penalises the weight matrix from growing too large; in practice, this can render many of the matrices so close to zero that it virtually eliminates the hidden unit from the neural network. This will decrease how much the model is overfitting the train data.

Another commonly used regularisation technique in Deep Learning is called **dropout regularisation**. It consists of going through each layer and setting some probability of eliminating a node in the neural network. This is usually decided for every training sample.
There are a few ways of implementing dropout:

- Inverted Dropout:

```
'''
Layer l=3
'''
keep_prob = 0.8
d3 = np.random.rand(a3.shape[0], a3.shape[1]) < keep_prob
#generates a tensor of the same size as the activation of
the layer
#elements of tensor either 0 or 1
a3 = np.multiply(a3, d3)
#If multiplied by zero, it cancels out
a3 /= keep_prob
#It scales a3 so that its expected value remains the same
#Otherwise it could affect the value of the next layer

```

During testing, we do not apply dropout.

In general, the number of neurons in the previous layer gives us the number of columns of the weight matrix, and the number of neurons in the current layer gives us the number of rows in the weight matrix.
There are other useful techniques for regularisation:

- Data augmentation: modifying samples by applying random modifications on them (such as distorting, flipping) so that they give more information to the algorithm.
- Early stopping: you plot your training and validation set errors and see at which point the network had not yet overfitted the data and was doing okay for both the training and the evaluation set. You stop training at that point.
- Parameter sharing: this consists in forcing a group of parameters to be equal. This technique is commonly used in convolutional neural networks. CNNs take advantage of the spatial structure of images by sharing parameters across different locations in the input. Since each kernel is convoluted with different blocks of the input image, the weight is shared among the blocks instead of having separate ones.

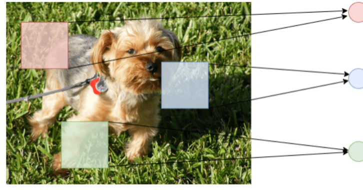


Figure 1: Image and description of Parameter Sharing sourced from here

When training a neural network, a technique that makes the training process a lot faster is **normalisation**. This process consists of two steps: firstly, you calculate the mean μ ($\frac{1}{m}\sum_{i=1}^{m} x^{(i)}$) and subtract it from every sample so that $x := x - \mu$. Then, you calculate the variance $\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)})^2$ and divide each sample by σ^2. If you use this scale to normalise the training set, remember to use it with the test set too! Normalisation is helpful because it decreases the range of values that the cost function can take, rendering the gradient descent process more efficient.

2 Optimisation Algorithms

Optimisation Algorithms can speed up significantly the training process, especially in the presence of big data. One technique commonly used is that of **mini–batch** gradient descent: normally, with gradient descent, the algorithm has to process every sample in the (potentially huge) training set. In order to avoid this, we create "mini–batches" that only have a portion of the larger training set. To run this on our training set, we do

```
for t in range(amount_of_mini_batches):
    Forward prop on X^{t} (vectorised implementation)
    Compute the cost function J for the minibatch
    Back prop to compute gradients
    Update the parameters in the layer
```

This will be an epoch of the gradient descent algorithm. Every epoch, we use a different mini–batch. Typical mini–batch sizes are $64, 128, 256, 512, 1024...$. Because of the way that computer memory is laid out and access, sometimes the code runs faster when the mini–batch size is a power of two.
When you plot the number of iterations vs the cost, the mini–batch approach is significantly noisier than the regular approach; it should still be a downward trend, but the noise comes from the fact that each epoch it is essentially training on a new training set. One set might prove easier than another one.

Exponentially weighted averages are a tool that allows us to access more sophisticated optimisation algorithms. It is a quantitative technique used as a forecasting model for time

series analysis. This consists in setting a value V_t so that $v_t = \beta V_{t-1} + (1-\beta)\theta_t$. Depending on the size of β, we will get a different value. This is because essentially our formula is just a way to sum the product between your samples and a decaying function. If we assume $t = time$, then when $\beta = 0.9$ it is taking the exponentially weighted average focusing on the last ten days. This is because it takes around 10 days for the value of the decaying function to decay $\frac{1}{e}$ from the peak. In more general terms, for $\epsilon = 1-\beta$, $(1-\epsilon)^{\frac{1}{\epsilon}} = \frac{1}{e}$. This method is frequently used in Machine Learning because it is more efficient in both terms of memory and computational efficiency.

Bias correction can render our computation of these averages more accurate. This is needed because at first, since $V_0 = 0$, all our values will be small and innacurate in representing the average. This can be fixed by taking $\frac{V_t}{1-\beta^t}$ instead of just V_t. This means that we will divide both sides by $1-\beta^t$ so you can obtain v_t. Many people don't bother applying bias correction and instead opt to wait out the bias, but if you are concerned with the initial accuracy of the average, then this is a great technique to deal with it.

Gradient Descent with **Momentum** almost always works better than standard gradient descent. As opposed to GD, GDM will compute the derivatives of the current mini–batch, as well as V_{db} and V_{dw}, where $V_{db} = \beta V_{db} + (1-\beta)db$ and $V_{dW} = \beta V_{dW} + (1-\beta)dW$. Once we have these values, we will update our parameters as $w := w - \alpha V_{dw}$ and $b := b - \alpha V_{db}$. This is because knowing what the exponentially weighted average of the derivatives is like can help us regulate the speed at which gradient descent goes and the way it accelerates. Using this analogy, β will be like friction, preventing it from keep on rolling. Let us look at an algorithm to compute this:

On iteration t:

Compute dW, db on the current batch

$$v_{dW} = \beta v_{dW} + (1-\beta)dW$$
$$v_{db} = \beta v_{db} + (1-\beta)db$$
$$W = W - \alpha v_{dw}$$
$$b = b - \alpha v_{db}$$

Another algorithm we can use for optimisation is called **Root Mean Square prop**, RM-

Sprop. It works as

On iteration t:

Compute dW, db on the current batch

$$S_{dW} = \beta S_{dW} + (1-\beta) dW^2$$
$$S_{db} = \beta S_{db} + (1-\beta) db^2$$
$$W = W - \alpha \frac{dW}{\sqrt{S_{dW}}}$$
$$b = b - \alpha \frac{db}{\sqrt{S_{db}}}$$

Based on the size of dW and db, the updates of the parameters will either become larger or smaller depending on how much each parameter oscillates in relation to the minimum point. Very often, to avoid the model from dividing by zero, we add a small term ϵ (e.g. 10^{-8}) to the denominator of the term that is being multiplied with α

The **Adam Optimisation Algorithm** (Adaptive Moment Estimation) is a rare case of an algorithm that has proven to be versatile in adapting to different deep learning architectures. It works as follows:

1. Initialise $V_{dW}, S_{dW}, V_{db}, S_{db} = 0, 0, 0, 0$
2. On iteration t:
   Compute dW, db using the current minibatch
   $V_{dW} = \beta_1 + (1-\beta_1) dW$
   $V_{db} = \beta_1 + (1-\beta_1) db$
   $S_{dW} = \beta_2 S_{dW} + (1-\beta_2) dW^2$
   $S_{db} = \beta_2 S_{db} + (1-\beta_2) db^2$
   $V_{dW}^{\text{corrected}} = \frac{V_{dW}}{(1-\beta_1^t)}$
   $V_{db}^{\text{corrected}} = \frac{V_{db}}{(1-\beta_1^t)}$
   $S_{dW}^{\text{corrected}} = \frac{S_{dW}}{(1-\beta_2^t)}$
   $S_{db}^{\text{corrected}} = \frac{S_{db}}{(1-\beta_2^t)}$
   Now, perform the update:
   $W := W - \alpha \frac{V_{dW}^{\text{corrected}}}{\sqrt{S_{dW}^{\text{corrected}}} + \epsilon}$
   $b := b - \alpha \frac{V_{db}^{\text{corrected}}}{\sqrt{S_{db}^{\text{corrected}}} + \epsilon}$

As you can see, this algorithm combines gradient descent with momentum with RMSprop. This algorithm has a large number of hyperparameters: $\alpha, \beta_1, \beta_2, \epsilon$. When using Adam, people normally iterate through different values of α to see which one proves the most efficient, while using the default values for the other parameters, namely $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$.

One of the things that might help speed up the learning algorithm is to slowly reduce the learning rate over time, a process known as **learning rate decay**. The intuition behind this action is that during the early steps of training, you can afford to take large steps each epoch; however, as you get closer and closer to the minimum, smaller steps are more efficient and prevent the algorithm from going too far away. The general formula for learning rate decay is

$$\alpha_{\text{decay}} = \frac{1}{1 + \text{decayRate x epochNumber}} \alpha_0$$

Other than this formula, other ways people approach learning rate decay are:

- $\alpha =$ number smaller than one $^{\text{epochNumber}} \cdot \alpha_0$
- $\alpha = \frac{k}{\sqrt{\text{epochNumber}}} \cdot \alpha_0$
- Manual decay

3 Hyperparameter Tuning

Machine Learning applications tend to involve a large amount of hyperparameters. When it comes to finetuning them, it is helpful to remember that not all of them are equally as important. In most applications, focusing on finetuning $\alpha, \beta,$ number of hidden units, mini-batch size is the most efficient approach. Even out of this subgroup, only α is constantly messed around with; β tends to remain with its default value of 0.9.

When choosing random values to sample, it is important to find an appropriate scale for the hyperparameter in question. For example, the range of 2 to 200 might be good for the amount of hidden layers in a particular project, but if we are checking out possible values for α, it is better to use a logarithmic scale.

4 Batch Normalisation (BN)

This algorithm makes the hyperparameter search problem easier and the neural network more robust. Batch normalisation consists in either normalising the value of a neural network layer before the activation function is applied (more common) or after.

This is implemented the following way:

- Given some intermediate values $Z^{(1)}, ..., Z^{(m)}$
- Compute:
  $\mu = \frac{1}{m} \sum_i z^{(i)}$
  $\sigma^2 = \frac{1}{m} \sum_i (Z_i - \mu)^2$
  $Z^{(i)}_{\text{norm}} = \frac{Z^{(i)} - \mu}{\sqrt{\sigma^2 + \epsilon}}$
  $\hat{Z}^i = \Gamma Z^{(i)} + \beta$ (This step is needed so that not every hidden union has the same distribution.) Gamma and Beta are learnable hyperparameters.

Usually, batch normalisation is already defined in deep learning frameworks, so there is no need to implement it from scratch.
When doing batch normalisation, the b hyperparameter gets cancelled out (since we subtract the mean from every element), so very often it is just eliminated from the process altogether.
Since at test time we just use a singular sample, we need to calculate an estimate of μ and σ^2 using the EWA across all of the minibatches. These will be the values we will use to apply normalisation to the test sample before feeding it into the (*normalised*) network.

5 Multiclass Classification

Softmax Regression is a generalisation of logistic regression that accounts for several output classes. The way we achieve this is by setting the final layer to be a softmax layer by adding a softmax activation function: $t = e^{z^{[l]}}$, and then

$$a^{[l]} = \frac{e^{z^{[l]}}}{\sum_{i=1}^{C} t_i}$$

where C stands for the number of possible output classes. Then, the output will be a vector with probabilities that the element belongs to several possible output labels.
Training will remain the same when we add this softmax function, the only difference being the loss function we will use. Since there are several possible probabilities represented in the output vector of this activation function, we will measure it as:

$$L(\hat{y}, y) = -\sum_{i=1}^{C} y_i log(\hat{y}_i)$$

Nowadays we use another version of this activation function for multiclass classification: **log-softmax**, namely

$$x - max(x)$$

```
sample!
```