Topic 2: Machine learning basics

Machine learning is all about making the computer discover how to solve problems without explicitly coding a solution for them. This is usually done by training a program to create a model that can be used to solve the problem. One common machine learning technique is called Gradient Descent which works by using the gradient of a function in order to find a local minimum.

Usually we take a model with parameters that makes predictions (computes outputs) and measure how wrong it is with respect to known correct outputs (we measure the error). We then try to find parameters for the model that result in the minimum error. Although gradient descent is efficient at doing this, it requires that the model and error function be differentiable which not be an easy thing to do.

Gradient descent works by repeatedly improving model parameters using the following function:

$$\theta^{t+1} \rightarrow \theta^t - \alpha \nabla_\theta \mathrm{f}(\theta^t)$$

where θ^t is the set of parameters of the model at step t, α is called the learning rate which is used to control how carefully to update the parameters, f is the function that is to be minimised (that measures the error of the model), and $\nabla_\theta \mathrm{f}$ is the partial derivative (gradient) of f with respect to θ. Remember that partial derivatives are with respect to a vector of variables rather than a single variable and return a vector of derivatives (one for each variable). θ^0 is the initial parameter set and is usually set randomly.

The important thing here is that, given a set of parameters θ, which is a point in the multi-dimensional graph (x-y graph, not Tensorflow graph) of f, the gradient gives you the direction on the graph where to move from θ in order to find the greatest value increase in f. Since we want to minimize f, we go in the opposite direction, hence the subtraction. α controls by how much we should move in that direction, which might send us in a completely new region in the graph if it is too big.

1. Minimise quadratic

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/01_-_Minimise_quadratic.py

Let us start simple. We shall use gradient descent to find the minimum of a quadratic equation. What we'll be optimizing (the parameters) is a single number which is the 'x' value of the point where the minimum of the quadratic is found. All parameters in Tensorflow should be expressed as variables rather than placeholders, which means that this minimum 'x' should be a variable. Don't worry though, we can still treat 'x' as a placeholder when running the session. We will arbitrarily initialise the minimum 'x' as 1, but it can be any random number.

2. Minimise error function

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/02_-_Minimise_error_function.py

One useful application of function minimisation is in minimising the error caused by function parameters so that you find the parameters that result in the least error. Here we shall try to find where two graphs intersect by finding the 'x' value where they are equal. Given an 'x' value, its error is how different from each other the two 'y' values of the graphs are. The smaller their difference, the closer we are to the

intersection. To measure how different the 'y' values are, we use the square difference of the two values which has a minimum of zero when the two values are equal. The error function is also called the 'cost function', the 'loss function', the 'objective function', and the 'energy function'. The square error is defined as follows:

$$E = (t - y)^2$$

where E is the error, t is the target output, and y is the actual output.

3. Interpolation

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/03_-_Interpolation.py

Now we will do something interesting. We are going to generate a polynomial to pass through a set of points, a process called interpolation. The points make up our training set which we will use to train the polynomial model. Our parameters to update are going to be the polynomial's coefficients and the error to minimize is going to be the mean square error (MSE) between the polynomial's predicted 'y' values at each 'x' value in the training set points and the target 'y' values of those same points. If our polynomial is large then we'll need to apply a lot of variable assignments in order to use gradient descent. Fortunately, Tensorflow comes with a readymade gradient descent function that does all the updates for us whilst we just have to pass it the learning rate and the error to minimize. We also need to be able to pass to the polynomial all the points in the training set at once in order to be able to measure the error of interpolation. This is accomplished by making the polynomial take in a vector of 'x' values rather than a single scalar. It can then either return a single error number or a vector of 'y' values.

4. Test sets

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/04_-_Test_sets.py

Interpolation is great but that is not machine learning, just optimisation. If we just wanted to fit a polynomial to our points perfectly we could just use Lagrange Polynomials. Machine learning is about generalisation, that is, being able to correctly predict values that weren't in the set of examples used during gradient descent (something that a perfectly fitting curve would probably fail to do well). What we need is a separate test set that is not used during training in order to be able to check whether it also correctly predicts those values after training. The points shown in the example are a quadratic function with some random noise added to it.

5. Early stopping

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/05_-_Early_stopping.py

When the training error keeps getting better but the test error starts getting worse, that is evidence that the model is overfitting, which means that the polynomial is fitting the training points too tightly and is not generalising to where new points would probably be. The most intuitive way to avoid this is to use early stopping where a third separate dataset called a validation set is used to monitor how well an unseen

dataset performs after every epoch. As soon as it starts doing badly on the validation set then the training is stopped abruptly. The resulting model can then be evaluated on the test set.

We usually set a patience, which is a number of epochs to allow the model to perform badly on the validation set. This is to see if the bad performance was just an anomalous epoch that gets back on track after a couple of epochs.

We also usually save the model parameters every time better ones are found and then load the last save parameters so that we only keep the best performing parameters. Unfortunately, given that our current programs are small and fast, saving the parameters after every epoch slows down program noticeably, so we will not be doing this in other programs.

The reason why we keep a separate validation set and test set is because early stopping is influencing the training and so the model might be unintentionally tuned towards performing well on the validation set. By keeping a separate test set that is not seen by either the gradient descent optimiser or the early stopping algorithm we can be more confident that the final evaluation is a fair one.

6. Regularisation

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/06_-_Regularisation.py

Early stopping stops the training right when it stops improving performance on the validation set but that does not make it perform better on non-training set data. Ideally we improve the best test set score and not just keep the best we find. To do this we need to regularise our models, which means changing the training process in a way that it forces the model to generalise better. One popular way to do this is to use weight decay where the optimiser not only tries to minimise the prediction error but also the magnitude of the parameters. The smaller the magnitude of the parameters, which are the polynomial's coefficients in this case, the simpler the polynomial's curve. Simple curves do not fit too tightly to the training set points. There are several things worth noting about this regularisation technique:

- The magnitude of the parameters is measured using either L1-norm (sum of absolute values) or L2-norm (sum of square values). Weight decay uses L2-norm but you can also use L1-norm and see which works best.
- It is important to distinguish between the prediction error and the parameters' magnitude when evaluating the polynomial. For this reason the error/magnitude combination which is to be collectively minimised is called the 'loss' but the error is still just the prediction error (without the parameters' magnitude).
- The first coefficient of the polynomial (the one multiplied by x^0) is not usually included in the parameters being regularised because that does not alter the shape of the curve, only the vertical shift of the curve, which should be allow to freely adjust itself.
- This regularisation method requires to be weighted in order to control how much of an effect it should have on the training process (if too high then the optimiser will not be able to reduce the prediction error and if too low then it will not have any effect).

- The weighting is a hyperparameter, which is a setting that is not optimised during training but must be chosen before training starts and kept fixed during training. In general, you find a good hyperparameter by trial and error. We will discuss ways to set hyperparameters later.
- Optimising two objectives at once (the prediction error and the parameter magnitudes) is called 'multiobjective optimisation'.
- See what happens when you set the weighting to a large positive number and a large negative number (1 and -1). What do you expect to happen?

7. Stochastic gradient descent

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/07_-_Stochastic_gradient_descent.py

Up to now, the optimiser has been seeing the whole training set at once every epoch. This is called batch learning. Another way to do it, which also happens to act as a regulariser, is to keep the optimiser from seeing the whole dataset at once and only optimise on a single item at a time. This is called online learning and if the order of training items being shown is random then the gradient descent becomes stochastic gradient descent. The stochastic gradient descent algorithm does not try to minimise the total prediction error of every point at once but will only minimise the predicted error for a randomly selected point at a time. This makes it harder for the model to overfit to the training set as the training set is changing all the time one point at a time.

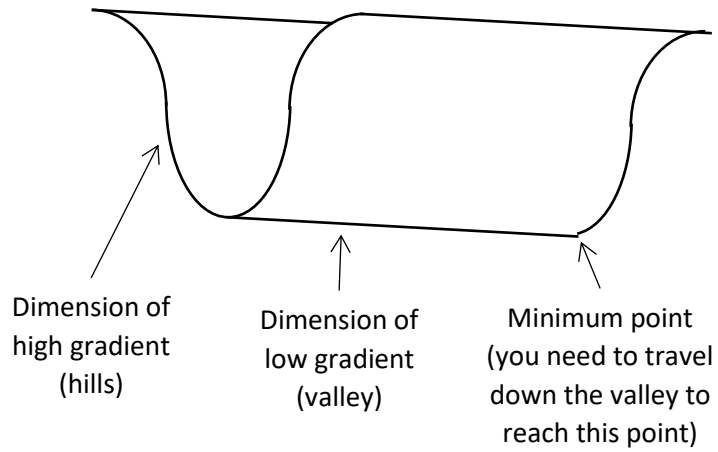
One point at a time can be too slow when the training set is too large, in which case we can compromise by optimising on a minibatch at a time, where a minibatch is a subset of the training set. You randomly shuffle the training set before every epoch and then go through the training set and take k items at a time, where k is the minibatch size and the k items are a minibatch. The optimiser is then shown one minibatch at a time to optimise the model incrementally, each new minibatch being use to further optimise the model for another small step.

The learning rate needs to be made smaller with small minibatches as the sequence of updates made to the model parameters can be noisy given that there is no clear minimum error to reach. Obviously, the early stopping algorithm needs to have more patience before stopping training as well.

8. Momentum

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/08_-_Momentum.py

One reason why gradient descent might be slow is because it gets to a point in the error curve where it's between two high gradient 'hills' in one dimension and a 'valley' that has a small gradient in another dimension. This means that a large learning rate will make it jump between the two hills and a small learning rate will make it move very slowly down the valley.



To mitigate for this we can use momentum in our gradient descent, which means adding the previous gradient to the current gradient before updating the weights. Here is the modified gradient descent equation:

$$g^t \rightarrow \alpha \nabla_\theta \mathrm{f}(\theta^t) + \gamma g^{t-1}$$

$$\theta^{t+1} \rightarrow \theta^t - g^t$$

This modified gradient descent requires another hyperparameter to control how much of the previous gradient to add to the current gradient. What happens here is that jumping between hills has an average gradient of zero since the two sides of the hills have opposite gradients whilst the small gradient valley will keep on accumulating speed.

9. Random initialisation

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/09_-_Random_initialisation.py

Finally, we solve another potential problem with gradient descent which is the initial parameters. Up to now we've been using zeros as initial values for each variable and that works on polynomials, but if two different variables are initialised with the same value and happen to have the same gradient (as will be the case with neural networks) then they will have identical values and update in exactly the same way. This will obviously make the two variables redundant as they are not contributing different information.

To avoid this situation we usually initialise all the variables using random numbers to 'break the symmetry' using a normal distribution with mean zero and a small standard deviation. This ensures that the variables are close to zero, which means that if they need to change their sign then they will not need many updates. It also ensures that no variable has a substantial advantage in terms of how important it is (unless you know that a certain variable is actually more important and thus you'd want to give it a larger random number on purpose). The first coefficient of the polynomial (the one multiplied by x^0) would not have the problems mentioned above and so is usually left as zero.

10. Hyperparameter tuning

https://github.com/mtanti/deeplearningtutorial/blob/master/tensorflow_v1/02_-_Machine_learning_basics/10_-_Hyperparameter_search.py

Hyperparameter tuning is a very important, slow, difficult, and boring process. Not a good combination. If you don't want to manually trying out different possible learning rates and momentums, there are three ways to do it automatically:

- Grid search: Write a bunch of nested for-loops, each going over a finite list of different values for a particular hyperparameter (one for-loop for the learning rate, another for the moment, etc.). Given a combination of hyperparameters, train your model with them and evaluate the model on a separate validation set. Keep the best performing hyperparameter combination.

- Random search: Grid search forces you to pick a finite list of possible values for each hyperparameter, so even though there is an infinite number of different learning rates to try, you need to decide on a handful of values to try. What's worse is that in order to avoid letting the process go on for months, you will usually only try a small list of possible values for each hyperparameter. You are very likely to make a poor choice of values and it is usually only a few hyperparameters that actually matter. So instead of choosing a list values, in random search we randomly sample a hyperparameter value for every hyperparameter (from the whole range of possible values not a random value from a list you make) and evaluate the combination. Repeat for a fixed number of times and pick the best combination.

- Bayesian optimisation: The other two methods are just uninformed search algorithms. As more and more hyperparameter combinations are evaluated, the information gained from these evaluations is not exploited by the search algorithm, which can be used to home in on a region in hyperparameter space that is likely to improve the evaluation results. This is what Bayesian optimisation does. It trains a random forest or Gaussian process model to predict the evaluation result of a hyperparameter combination (random forests generally make better predictions than Gaussian processes when you have categorical hyperparameters and vice versa) and uses the model to predict which combination is most likely to give good results. As more combinations are evaluated, the model gets better at predicting good combinations and so you are more likely to find better hyperparameters as time goes on. A library that does this process for you is called Scikit Optimize, or just skopt.

11. Extra theory

- Minibatches help avoid stationary points in the parameter space. When using batch learning (whole training set is shown to optimizer at once), if the optimizer should reach a stationary point in the parameter space (a point where all dimensions have a gradient of zero but is not a turning point) then the optimizer cannot update the parameters any more. With minibatches, the parameter space changes with every minibatch since the prediction error is defined on the minibatch. If the current parameters result in a stationary point in the parameter space created by a minibatch, the same parameters on the next randomly selected minibatch will probably not be in a stationary point as well so training is rarely stalled because of stationary points.

- Vanilla gradient descent, with momentum or not, is not the only flavor of gradient descent. There are improvements on gradient descent which aim at making the learning rate adapt as training

progresses, such as Adam and RMSProp. See this really good blog that explains them for information on how they work and this Tensorflow page documenting the available optimisers.

- Finally, polynomials are universal approximators, that is, they are able to get as close as you want (but not perfectly) to fitting a finite region of any continuous function. On the other hand, whilst being universal approximators, polynomials have two problems in general when it comes to modelling complex functions: (1) they need to have large exponents in order to be flexible enough which tend to result in numerical overflows; and (2) in order to make them accept more than one input you need to create an enormous polynomial that has a coefficient for every combination of input and exponent e.g. this is what a quadratic equation with two inputs looks like: $y = c_0 + c_{10}x_0 + c_{11}x_1 + c_{20}x_0^2 + c_{21}x_1^2 + c_{22}x_0^2x_1 + c_{23}x_0x_1^2 + c_{24}x_0^2x_1^2$. Thankfully, polynomials are not the only universal approximator functions that are easy to learn with gradient descent. This is where neural networks come in.