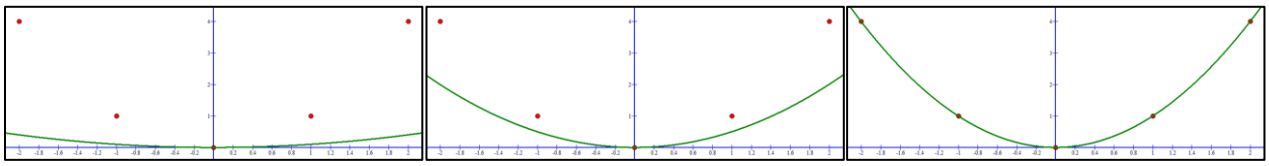
Topic 2: Machine learning basics

Machine learning is all about making a computer automatically discover how to solve a problem without a programmer explicitly coding a solution (although the programmer still has to code the machine learning method). This is usually done by creating a parameterised model where there exists a set of parameter values that result in the model solving the problem. An algorithm called an optimiser would then look for the parameter values that move the model towards solving the problem, a process called training. One common optimisation algorithm is called Gradient Descent, which works by using the gradient of a function in order to find a local minimum.

In supervised learning, the most common kind of machine learning, we take a parameterised model (such as a polynomial function with variable coefficients) and supply it with some inputs (such as a set of x values from points we want the polynomial to pass through). We then make the model predict what the outputs should be (the predicted y values of the points) and measure how wrong it is with respect to known correct outputs (the correct y values of the points), that is, we measure the model's error. The optimisation algorithm would then look for parameters (coefficients) that would minimise this error. The error is calculated by a function called an error function.



A polynomial that is progressively predicting the red points correctly. In practice, the points would be something like the relationship between the distance of a planet from the sun and the time it takes to make a complete orbit. The polynomial would then be trained to predict a planet's orbit time from its distance to the sun (this is not what is shown in the above figures).

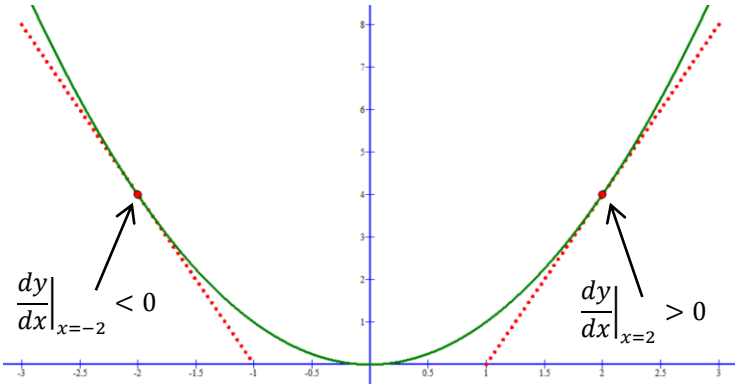
Gradient descent works by repeatedly improving model parameters using the following equation:

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

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.

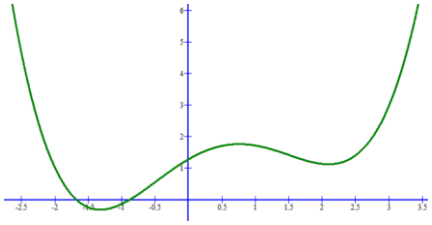
Given that θ is a set of numbers, we can think of it as a point in multi-dimensional space called the parameter space. You can further think of every point in parameter space as giving some error in the model as measured by the error function. Starting from a random point in parameter space, every step of gradient descent gives you the direction and distance to move the point in order to have smaller and smaller errors. Measuring the gradient of the error with respect to the parameters gives you the direction in the space where to find the greatest value increase in error. Since we want to minimize the error, we go in the opposite direction, hence the negative sign in the gradient descent equation. The learning rate controls the distance with which we should jump in the given direction, which might send us in a completely unrelated region in parameter space if it is too big, which prevents learning. Each application

of gradient descent is called an epoch and the number of epochs to use depends on the problem you're solving, just like the learning rate.



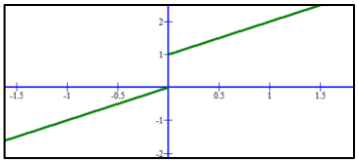
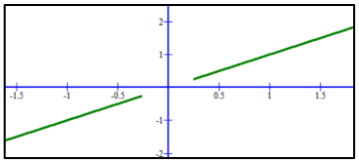
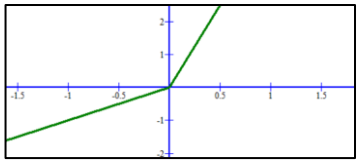
You can find what change in x is needed in order to decrease y by checking the gradient at the point you're at. If the gradient is a negative number then you should increase x whilst if it is a positive number then you should decrease x.

Of course, if there is more than one minimum turning point (the function being optimised is not convex) then you will go down towards the turning point you happen to start within but it might not be the actual global minimum point in the whole function (it would be a local minimum but not a global minimum). We'll ignore this problem for now and talk about it again in the next topic.



Example of a non-convex function. There is no guarantee about which one of the local minima the gradient descent algorithm will approach when starting from a point at random.

At this point, it is important to know what a differentiable function is. Gradient descent can only be used effectively on a model if the gradient of the error function is differentiable with respect to the model's parameters, that is, the error changes gradually as the parameters are changed. If a function changes in sudden vertical jumps then it is not differentiable. If a function has any gaps in its graph then it is also not differentiable. If a function has a sudden change in direction (a cusp) then it is not differentiable either. Here are examples of each of the above cases:

Three examples of non-differentiable functions. The first has a vertical jump in output, the second has a horizontal jump in input (undefined region), and the third has a cusp with a sudden change in direction.

Designing an error function and model that are differentiable is not always an easy thing to do. Fortunately in practice it is possible to still use gradient descent when the error function is differentiable everywhere except a few points as in the first and third cases above since it is very unlikely that you will need to find the gradient at that particular point. We will see examples of differentiable error functions later in this topic.

1. Minimise quadratic

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

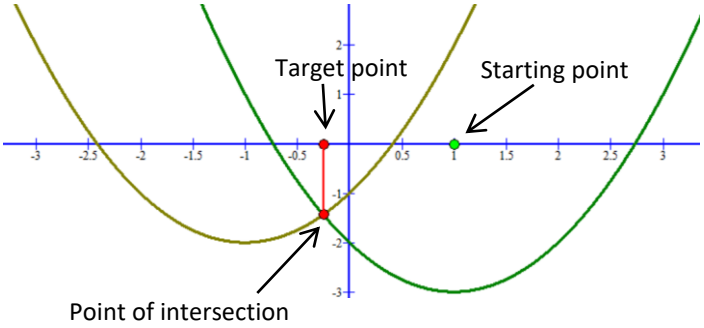
Let us start simple. We shall use gradient descent to find the minimum of $y = x^2$. What we'll be optimizing (the parameters) is a single number which is the 'x' value of the point where the minimum of the quadratic is found. All parameters in Tensorflow should be expressed as variables rather than 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.

After running the script, notice how the steps of gradient descent start getting smaller and smaller as it approaches the minimum. This is by design (hence why we move in proportion to the gradient) in order to avoid overshooting the minimum, but it also means that after a few epochs, the amount of improvement gained starts becoming negligible which means that you might as well stop as you'd be spending a lot of time just for a tiny amount of improvement.

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 error functions which are functions that quantify how far off the model parameters are from the desired point. We can then use function minimisation in order to find the parameters that result in the smallest error. In the script, we shall try to find the point of intersection of two polynomials. Starting from $x = 1$, we need to find the right value of x that, when applied to the two polynomials result in the same y value.



How can we measure the error of an arbitrary x value? We can measure the difference between the current x value and the target x value, but that would require knowing what the target value is which means that we would be solving the problem that we are trying to solve in the first place. A more useful error measure is the difference between the two y values of the polynomials at the current x value. The smaller their difference, the closer we are to the intersection. Rather than just taking the difference between values, which does not have a minimum as it can keep getting negative, we instead measure the square difference of the two values which has a minimum of zero when the two values are equal and positive everywhere else. 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.

The error function is also called the 'cost function', the 'loss function', the 'objective function', and the 'energy function'. It's important to note that the error function might not be what we are actually trying to optimise when the system is being used in practice. For example, when classifying what an object contains, our primary quality measure of the system might be accuracy. But accuracy is not differentiable as it requires that an output either is correct or not, which is not a function that changes gradually. In this case our job is to find a differentiable approximation to the quality measure that can be used to optimise our model. The square error function is differentiable, and although you can find cases where a parameter update can improve the square error but reduce the accuracy (they are not perfectly correlated functions), it is still good enough to be useful. Also, a perfect square error (0) necessarily requires a perfect accuracy (100%).

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.

We're going to be using a polynomial of degree 6, which means that we'll have 7 variables to update. Rather than making 7 separate variable assignments, Tensorflow comes with a readymade gradient descent function that automatically updates every variable in the graph. You can get a list of these

variables by called 'tf.trainable_variables()'. All we have to do is create a 'GradientDescentOptimizer' object and 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 model take in a vector of x values rather than a single scalar and the output would be a vector of y values.

For the sake of understanding what happens during training, this script will be showing the matplotlib figure and updating it during training rather than at the end.

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 not about perfectly fitting the provided points but about being able to correctly predict new unseen points as well. This is called generalisation and is 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. Such a dataset is called the test set whilst the dataset being used to optimise the model is called the training set.

The points shown in the provided script 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 a process called 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 few more epochs.

We also usually save the model parameters every time better ones are found and then load the last saved 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.

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 new 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, just like the learning rate and maximum number of epochs.
- 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 (10 and -10). 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 algorithm becomes known as the Stochastic Gradient Descent algorithm. Updating the parameters on one training item at a time makes it harder for the model to overfit to the training set as the training set is constantly changing (to the model, the training set is that single item). Note that randomly sampling a point for every

parameter update will lead to some items being presented to the optimiser more often than others as it is very unlikely that every item will be selected an equal number of times, at least during the initial few samples made. This will make some training items more important than others, which might be undesirable. In order to guarantee that every item is selected an equal number of times whilst still being selected in random order, the training set is shuffled randomly and then gone through one item at a time.

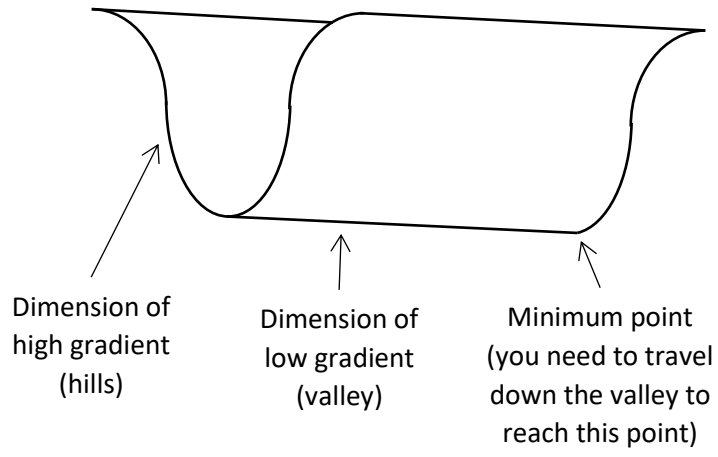
One training item at a time can be too slow when the training set is large, in which case we can compromise by optimising on a minibatch at a time, where a minibatch is a subset of the training set. You randomly shuffle the training set and then go through the training set and take k items at a time, 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.

With minibatches, an epoch is not a single minibatch. That is called a step instead. An epoch is still an entire sweep through the training set. The learning rate needs to be made smaller with small minibatches as the sequence of updates made to the model parameters can be noisy which can throw off the training progress if each update is too big. Also due to the noise, when using minibatches in conjunction with the early stopping algorithm, you need to set a higher patience. Early stopping can also be applied to individual steps instead of entire epochs, especially if the training set is really big.

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 parameter space where it's between two high error gradient 'hills' in one dimension and a 'valley' that has a small error gradient in another dimension. This means that a large learning rate will make it jump between the two hills and only make it progress down the valley very slowly whilst a small learning rate will also make it move very slowly.



To mitigate for this we can use momentum in our Gradient Descent, which means adding the previous 'velocity', that is, amount of update applied to the parameters, to the current amount of update that the Gradient Descent algorithm would make. Here is the modified Gradient Descent equation:

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

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

where v^t is the velocity, that is, the amount of update to apply to the parameters (a number for each parameter) at update t, $\alpha \nabla_\theta f(\theta^t)$ is the plain velocity produced by gradient descent without momentum, and γ is the fraction of the previous velocity to add to the plain velocity. This modified gradient descent requires another hyperparameter to control how much of the previous velocity to add to the current velocity.

In the case of the valley situation with momentum, since velocities are added up, jumping between hills has an average velocity of zero since the two sides of the hills have opposite gradients and so there will be positive velocity on one hill and negative velocity on another. This will cancel out oscillations. On the other hand, the small but consistent gradient in the valley will keep on accumulating velocity and accelerate the descent down the path.

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 (go from positive to negative or vice versa) then they will not need many updates. It also ensures that no variable has a substantial advantage in terms of how important it is. 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 spend hours or days 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. 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 (as the total number of combinations grows as the product of the amounts of values). You are very likely to make a poor choice of finite values and it is usually only a few hyperparameters that actually matter. So instead of choosing a finite list of 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 previous two methods are 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 takes all the evaluated hyperparameters together with their final error and constructs a training set out of them. It then trains a Random Forest or Gaussian Process model to learn to predict the final error from a hyperparameter combination and uses this model to predict which combination is most likely to give good results. Random Forests generally make better predictions than Gaussian Processes when you have categorical hyperparameters and vice versa when you only have real number hyperparameters. As more combinations are added to the hyperparameter training set, 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.

Evaluation of the hyperparameters should not be carried out on the test dataset as, just like in the case of early stopping, the final test set should never influence training in any way. So you need to use a separate validation set, which should also be different from the one used in the early stopping algorithm if you are using both.

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 optimiser 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 prematurely stalled thanks to stationary points.

- Vanilla gradient descent, with momentum or not, is not the only flavour of gradient descent. There are extensions to gradient descent that make 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 (albeit 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 a large degree in order to be flexible enough to model practical data which tends to result in numerical overflows; and (2) in order to make

them accept more than one input you need to create an enormous multidimensional 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.