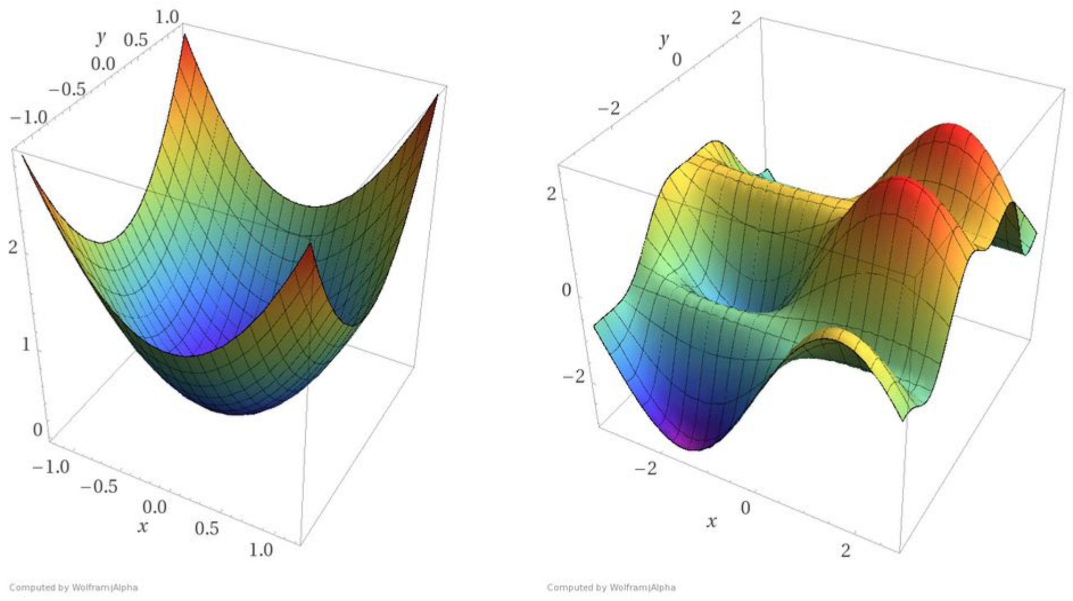
Gradient Descent

Author: **Neeraja Menon**
www.linkedin.com/in/neeraja-j-menon

Let’s say that you want to create a neural network for the purpose of profit forecasting. It will take in various financial indicators as input such as business expenses, total revenue, and the current market conditions, in order to make predictions about future profits. Since there are many variables to consider, we don’t know exactly how much each indicator contributes to the amount of profit a company generates - for example, a change in the market conditions may negatively impact the forecast more than an increase in business expenses. Therefore, the neural network needs to figure out the most effective combination of weights to apply to create a robust model. This is where gradient descent proves useful.

Gradient descent is a type of optimisation algorithm to find the best weights for your Neural Network. This means it takes a cost function and moves in the direction of the steepest decrease of gradient, in order to find the parameters which best reduce the difference between the predicted value and target value of the model. The underlying idea in this approach is the same as with similar problems you may have already come across during school when you were introduced to calculus. A function can have local minima and global minima - these are exactly the points on the curve that gradient descent aims to find.



Source: https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/

The graph on the left displays a function with only one minimum point, while the graph on the right has multiple ones. The x and y axis represent weights, and the z axis represents the cost value at those weights.

Now you may be wondering: why not just differentiate the cost function, equate it to 0 to find the stationary point(s), and solve the equation to calculate the parameter values, like you were taught in school? This may work for simpler models with one or two variables, but when dealing with more difficult models, where cost functions are often non-linear, the number of inputs is much larger, and with neural networks containing multiple hidden layers, it is impossible to achieve them analytically. Gradient descent is a machine learning technique that can be applied to a range of complex problems; it can be parallelised and optimised for computational efficiency, and handle extensive datasets. Its versatility allows it to be regularly used in several machine learning models such as logistic regression, linear regression, and deep learning. In this article, we'll focus on how gradient descent is applied in neural networks to demonstrate the range of complexity it can cope with. However, first, you'll need to know about the building blocks that make up gradient descent.

Cost Functions

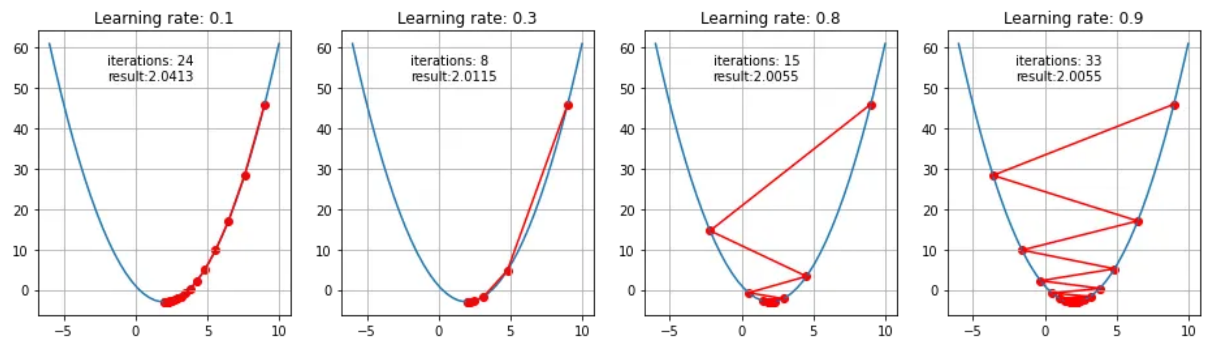
It's important to understand what a cost function is as this can make all the difference in the performance of your model. These functions (also known as loss functions) measure the difference between the predicted values and target values in a machine learning model. It is used to assess the accuracy of a model's predictions and represents its overall performance. Depending on the kind of machine learning problem being solved, the cost function chosen will vary. For instance, Mean Squared Error (MSE) is frequently employed in linear regression while Cross-Entropy Loss is utilised in logistic regression. Each cost function has its own advantages and disadvantages, so in order to tailor them to the problem at hand, they must be chosen according to the type of errors you're trying to highlight. Some functions are more sensitive to outliers, while others tend to only emphasise larger errors. An example of what a cost function looks like can be seen below:

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

The equation above shows how MSE is calculated. The distance between the predicted and target value is first squared for each data point (to remove any negatives), summed together, and divided by the total number of data points present, thus calculating the average error in the model. Now, let's look at how learning rates are applied to these functions.

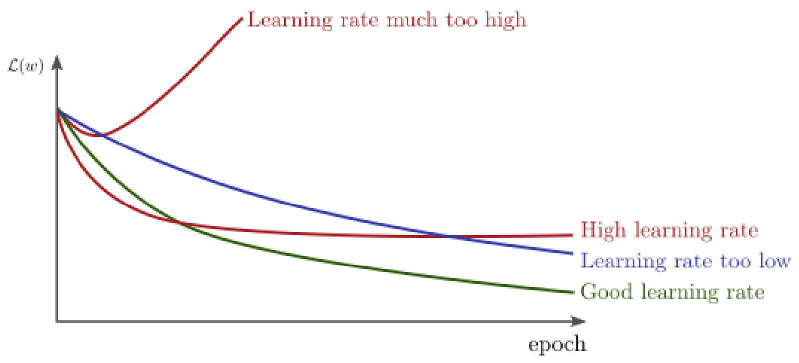
Learning Rates

Another significant factor to consider while performing gradient descent is known as the learning rate. As the name suggests, this is how fast or slow the model learns. Initially, the model starts at a random point on the curve, usually at a high-cost value. The aim of gradient descent is to find the minimum cost value from this starting point. Since it is extremely unlikely that this initial fitting will be close to the optimised one, the learning rate starts off large. This means it takes bigger steps down the curve in the direction of the steepest descent.



Source: https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21

While it gets closer to the minimum point, and the cost value is largely decreasing, the learning rate must be slowly decreased. This is to ensure that the model does not overshoot this minimum point and starts moving up the curve, increasing its cost value again. A good learning rate ensures that the model's parameters converge to a good solution in a reasonable amount of time, while a poor learning rate can cause the model to converge too slowly or not at all (where it will start to diverge). It is a trade-off between the speed of convergence and the risk of overshooting the minimum of the cost function.



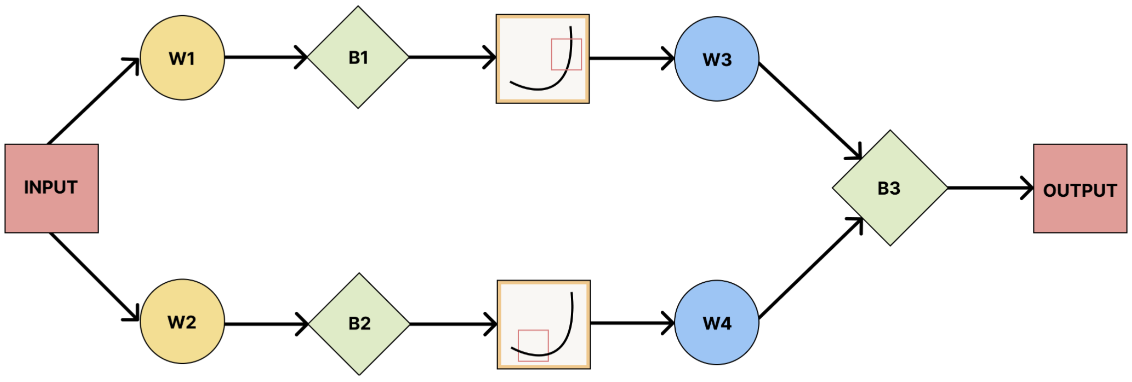
Source: https://cs231n.github.io/neural-networks-3/

In reality, models do not reach the exact minima, but rather keep oscillating around very close to it where there is no decrease in the loss value anymore, making the solution an optimum one. Since you're now familiar with how gradient descent generally works with simpler models, let's see how they're used in neural networks.

Backpropagation

Backpropagation quantifies the gradient of the error function given one forward pass and one target data point. It is based on the chain rule of calculus, which allows the gradient of the cost function (with respect to the parameters) to be decomposed into a series of intermediate gradients. By propagating the error backwards through the network's layers, beginning with the output layer and moving backwards towards the input layer, the intermediate gradients can be determined. Each layer in the network contributes to said gradients, allowing the parameters to be updated periodically.

To help you to better understand and follow the maths associated with this, you can use the diagram below of a simple neural network for visualisation:



The orange squares display the activation function being updated. The values in the green diamonds represent the biases, and the circles represent the weights.

The equation below is used to update the weights in such networks. The variable highlighted in red is the derivative of the loss function with respect to the current weight.

$$W_i = W_i - \mu \frac{\delta L}{\delta W_i}$$

This variable is multiplied by the learning rate μ , subtracted from original weight, to then obtain the new weight for the next iteration of gradient descent. How much does the loss change when we wiggle this one weight? It's only partial as the functions contain more than one unknown variable - so it's more general.

In order to apply this, we need to know the variable highlighted in red. To find this partial derivate of our loss function, we use the chain rule. For this example, lets imagine we are using the network presented above, the cost function as Residual Sum of Squares (RSS), and the current weight as W3.

The formula for the loss function that we'll use is:

$$\textbf{RSS} = \sum_{i=0}^{n=3} (True_i - Predicted_i)^2$$

The value we are trying to find can be expressed the following way:

$$\frac{\text{d RSS}}{\text{d W3}} = \frac{\text{d RSS}}{\text{d Predicted}} \text{ x } \frac{\text{d Predicted}}{\text{d W3}}$$

First, find the derivative of **RSS** with respect to **Predicted**:

$$\frac{\text{d RSS}}{\text{d Predicted}} = \frac{\text{d}}{\text{d Predicted}} \sum_{i=0}^{n=3} (True_i - Predicted_i)^2 = -2 \times \sum_{i=0}^{n=3} \times (True_i - Predicted_i)$$

Next, let's find the derivative of **Predicted** with respect to **W3** (Remember, since it is with respect to W3, variables without W3 will automatically equal 0) :

$$\frac{\text{d Predicted}}{\text{d W3}} = \frac{\text{d}}{\text{d W3}} (y_{1,i} w_3 + y_{2,i} w_4 + b_3) = y_{1,i}$$

Now, putting it all together, we have our equation:

$$\frac{\text{d RSS}}{\text{d W3}} = \frac{\text{d RSS}}{\text{d Predicted}} \text{ x } \frac{\text{d Predicted}}{\text{d W3}} = -2 \times \sum_{i=0}^{n=3} \times (True_i - Predicted_i) \times y_{1,i}$$

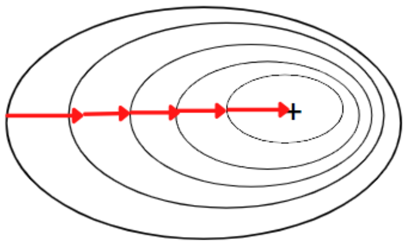
Wait, we're not done just yet! We now need to repeat this for all weights and biases in the network. Luckily, many machine learning software takes care of this for you, so you don't have to worry about the specifics.

Gradient Descent Variations

Gradient descent is commonly used to train neural networks. It calculates how much to change the weights and biases in a network **given an average of multiple back propagations.**

Batch

The type of gradient descent we have explained thus far is also known as 'Batch Gradient Descent'. The gradient of the cost function with respect to the model's parameters is calculated using the **entire training data**. Using all the data can be computationally expensive, especially for large datasets, as it requires processing the entire training data at each iteration. However, it has the advantage of providing a more accurate estimate of the gradient, which can result in better optimization performance.

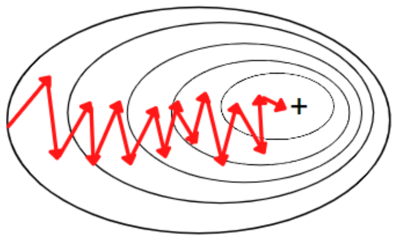


Source: https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/

Visualisation of the path Batch GD tends to follow to reach the minimum point

Stochastic

In Stochastic Gradient Descent (SGD), we only use a random sample (**one datapoint** for each feature) of the training data rather than all of it, making it 'stochastic'. When there is a ton of training data, it largely reduces the amount of processing needed, making it the fastest gradient descent variant. At each iteration, a randomly selected batch of the training data is used to calculate the gradient of the cost function with respect to the model's parameters. The parameters are then updated in the direction of the negative gradient. This process is repeated for a specified number of iterations or until convergence.

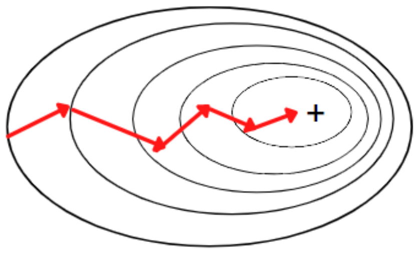


Source: https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/

Visualisation of the path SGD tends to follow to reach the minimum point

Mini-Batch

This variant is a mix of the previous two methods- it has more variance in the gradient estimate than Batch GD but is less erratic than SGD. It works by taking a random small subset or 'mini-batch' of the training data for the optimisation algorithm. In mini-batch gradient descent, the batch size is another hyperparameter alongside the learning rate. It controls how many training instances are used per batch to calculate the gradient at each iteration. Another trade-off is introduced in this type of gradient descent: there must be a compromise between the variance of the gradient estimate and the convergence speed. A larger batch size results in a higher variance in the gradient estimate, whereas a lower batch size has less variance in the gradient estimate but converges much slower.



Source: https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/

Visualisation of the path Mini-Batch GD tends to follow to reach the minimum point

Implement it yourself!

You now know what gradient descent is, how it works, and its variants. In order to really understand and familiarise yourself with it, the next step would be to play around with the code. Here are some libraries to get you started:

We recommend Tensorflow (https://www.tensorflow.org/tutorials/quickstart/beginner) to programming beginners for its shorter and simpler code and PyTorch (https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html) for members with previous programming experience.

In particular, SGD can be implemented using a popular library from scikit-learn (https://scikit-learn.org/stable/modules/sgd.html).

For more detailed steps on how to achieve gradient descent in PyTorch you can check out https://machinelearningmastery.com/implementing-gradient-descent-in-pytorch/.

Author: Neeraja Menon
www.linkedin.com/in/neeraja-j-menon