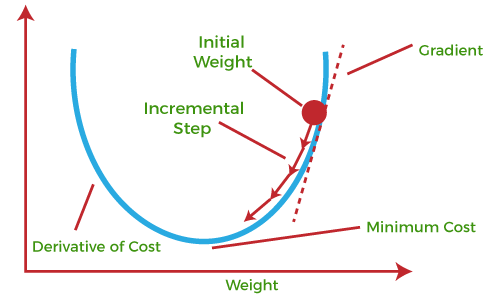
# Gradient Descent in Machine Learning

Gradient descent is an optimization algorithm which is commonly-used **to train machine learning models and neural networks**. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. An optimization algorithm is **a procedure which is executed iteratively by comparing various solutions till an optimum or a satisfactory solution is found**.

**It helps in finding the local minimum of a function.**

The best way to define the local minimum or local maximum of a function using gradient descent is as follows:

* If we move towards a negative gradient or away from the gradient of the function at the current point, it will give the **local minimum** of that function.
* Whenever we move towards a positive gradient or towards the gradient of the function at the current point, we will get the **local maximum** of that function.



This entire procedure is known as Gradient Ascent, which is also known as steepest descent. ***The main objective of using a gradient descent algorithm is to minimize the cost function using iteration.*** To achieve this goal, it performs two steps iteratively:

* Calculates the first-order derivative of the function to compute the gradient or slope of that function.
* Move away from the direction of the gradient, which means slope increased from the current point by alpha times, where Alpha is defined as Learning Rate. It is a tuning parameter in the optimization process which helps to decide the length of the steps.

### **What is Cost-function?**

**The cost function is defined as the measurement of difference or error between actual values and expected values at the current position and present in the form of a single real number.**

It helps to increase and improve machine learning efficiency by providing feedback to this model so that it can minimize error and find the local or global minimum. Further, it continuously iterates along the direction of the negative gradient until the cost function approaches zero. At this steepest descent point, the model will stop learning further.

Advantages:

* Easy computation.
* Easy to implement.
* Easy to understand.

Disadvantages:

* May trap at local minima.
* Weights are changed after calculating gradient on the whole dataset. So, if the dataset is too large than this may take years to converge to the minima.
* Requires large memory to calculate gradient on the whole dataset.

## **Types of Gradient Descent**

Based on the error in various training models, the Gradient Descent learning algorithm can be divided into **Batch gradient descent, stochastic gradient descent, and mini-batch gradient descent.** Let's understand these different types of gradient descent:

### **1. Batch Gradient Descent:**

Batch gradient descent (BGD) is used to find the error for each point in the training set and update the model after evaluating all training examples. This procedure is known as the training epoch. In simple words, it is a greedy approach where we have to sum over all examples for each update.

**Advantages of Batch gradient descent:**

* It produces less noise in comparison to other gradient descent.
* It produces stable gradient descent convergence.

**Disadvantages of Batch gradient descent**:

* It is Computationally efficient as all resources are used for all training samples. As we need to calculate the gradients for the whole dataset to perform just one update, batch gradient descent can be very slow and is intractable for datasets that don't fit in memory.
* Batch gradient descent also doesn't allow us to update our model online, i.e. with new examples on-the-fly.

### **2. Stochastic gradient descent**

Stochastic gradient descent (SGD) is a type of gradient descent that runs one training example per iteration. Or in other words, it processes a training epoch for each example within a dataset and updates each training example's parameters one at a time. As it requires only one training example at a time, hence it is easier to store in allocated memory. However, it shows some computational efficiency losses in comparison to batch gradient systems as it shows frequent updates that require more detail and speed. Further, due to frequent updates, it is also treated as a noisy gradient. However, sometimes it can be helpful in finding the global minimum and also escaping the local minimum.

**Advantages of Stochastic gradient descent:**

In Stochastic gradient descent (SGD), learning happens on every example, and it consists of a few advantages over other gradient descent.

* It is easier to allocate in desired memory.
* It is relatively fast to compute than batch gradient descent.
* It is more efficient for large datasets.

### **3. MiniBatch Gradient Descent:**

Mini Batch gradient descent is the combination of both batch gradient descent and stochastic gradient descent. It divides the training datasets into small batch sizes then performs the updates on those batches separately. Splitting training datasets into smaller batches make a balance to maintain the computational efficiency of batch gradient descent and speed of stochastic gradient descent. Hence, we can achieve a special type of gradient descent with higher computational efficiency and less noisy gradient descent.

**Advantages of Mini Batch gradient descent:**

* It is easier to fit in allocated memory.
* It is computationally efficient.
* It produces stable gradient descent convergence.

**Challenges of SGD**

* Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
* Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. These saddle points are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

### **Vanishing Gradients:**

Vanishing Gradient occurs when the gradient is smaller than expected. During backpropagation, this gradient becomes smaller that causing the decrease in the learning rate of earlier layers than the later layer of the network. Once this happens, the weight parameters update until they become insignificant.

### **Momentum (SGD with momentum)**

Momentum or SGD with momentum is method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it.

Let’s consider two extreme cases to understand this decay rate parameter better. If the decay rate is 0, then it is exactly the same as (vanilla) gradient descent. If the decay rate is 1 (and provided that the learning rate is reasonably small), then it rocks back and forth endlessly like the frictionless bowl analogy we mentioned in the beginning; you do not want that. Typically, the decay rate is chosen around 0.8–0.9 — it’s like a surface with a little bit of friction so it eventually slows down and stops.

So, in what ways is Momentum better than vanilla gradient descent? Two advantages:

1. Momentum simply moves faster (because of all the momentum it accumulates)
2. Momentum has a shot at escaping local minima (because the momentum may propel it out of a local minimum).

**Adagrad**

Instead of keeping track of the sum of gradient like momentum, the **Ada**ptive **Grad**ient algorithm, or AdaGrad for short, keeps track of the sum of gradient squared and uses that to adapt the gradient in different directions.  Adagrad is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data**.**

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that**.**

Adagrad's main weakness is its accumulation of the squared gradients in the denominator:

* Since every added term is positive, the accumulated sum keeps growing during training.
* This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.

# RMSProp

The problem of AdaGrad, however, is that it is incredibly slow. This is because the sum of gradient squared only grows and never shrinks. RMSProp (for **R**oot **M**ean **S**quare **Prop**agation) fixes this issue by adding a decay factor.

The more you have updated a feature already, the less you will update it in the future, thus giving a chance for the others features (for example, the sparse features) to catch up.

To see the effect of the decaying, in this head-to-head comparison, AdaGrad white) keeps up with RMSProp (green) initially, as expected with the tuned learning rate and decay rate. But the sums of gradient squared for AdaGrad accumulate so fast that they soon become humongous (demonstrated by the sizes of the squares in the animation). They take a heavy toll and eventually AdaGrad practically stops moving. RMSProp, on the other hand, has kept the squares under a manageable size the whole time, thanks to the decay rate. This makes RMSProp faster than AdaGrad**.**

**Adam**

Last but not least, Adam (short for Adaptive Moment Estimation) takes the best of both worlds of Momentum and RMSProp. Adam empirically works well, and thus in recent years, it is commonly the go-to choice of deep learning problems.

Another method that calculates the individual adaptive learning rate for each parameter from estimates of first and second moments of the gradients.

It also reduces the radically diminishing learning rates of Adagrad

Adam can be viewed as a combination of Adagrad, which works well on sparse gradients and RMSprop which works well in online and nonstationary settings.

Adam implements the exponential moving average of the gradients to scale the learning rate instead of a simple average as in Adagrad. It keeps an exponentially decaying average of past gradients

Adam is computationally efficient and has very little memory requirement

Adam optimizer is one of the most popular gradient descent optimization algorithms

**Bias**

Bias is considered a systematic error that occurs in the machine learning model itself due to incorrect assumptions in the ML process. Technically, we can define bias as the error between average model prediction and the ground truth. Moreover, it describes how well the model matches the training data set:

Your model is underfitting the training data when the model performs poorly on the training data.

Overfitting: A statistical model is said to be overfitted when the model does not make accurate predictions on testing data.

*The ultimate goal of any supervised machine learning* problem is to find a model or function that predicts a target or label and minimizes the expected error over all possible inputs and labels. Minimizing error over all possible input means the function must be able to generalize and make accurate predictions on unseen inputs. In other words, the fundamental goal of machine learning is for the algorithm to generalize beyond the training sets.

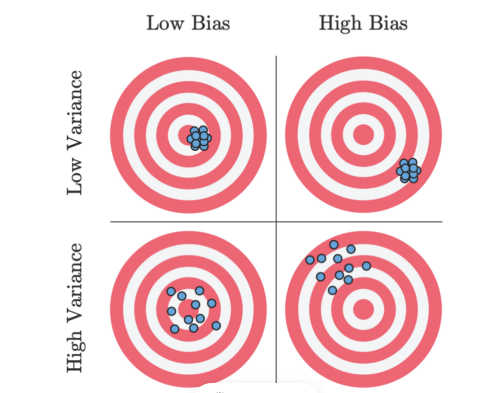
Regularization performs feature selection by shrinking the contribution of each feature. Regularization will help select a midpoint between the first scenario of high bias and the later scenario of high variance. This ideal goal of generalization in terms of bias and variance is a low bias and a low variance which is near impossible or difficult to achieve. Hence, the need of the trade-off. We might have to reduce accuracy on training data from 100% to 80% and increase accuracy on unseen data from 50% to 80%.

## High Bias

* Overly-Simplified Model
* Under-Fitting
* High error on both test and train data

## High Variance

* Overly-complex model
* Overly-Fitting
* Low error on train data
* High error on test data
* Starts modeling the noise in the input



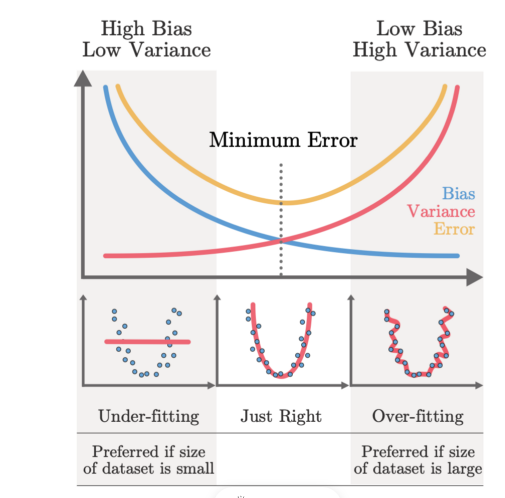
**Bias Variance Trade-Off**

* Increasing bias reduces variance and vice-versa

**Error = Bias² + Variance + irreducible error**

* The best model is where the error is reduced.
* Compromise between bias and variance.

**Regularization**



# Early Stopping

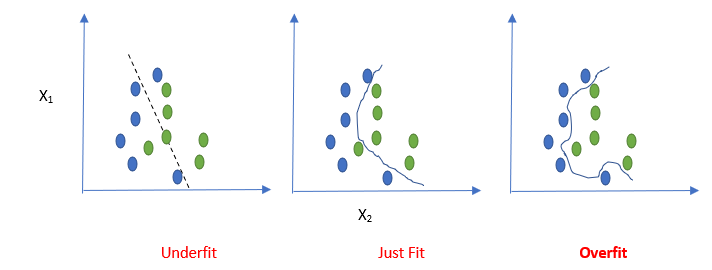
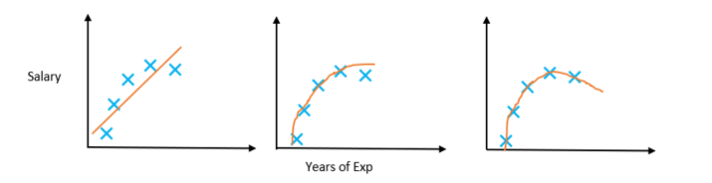


Fig 1: Decision Boundary for three models (Image Source: Author)

*However, the real culprit is the third model, it is trying to memorize the training data, picking up minor fluctuations in the training data, actually going way too far for a perfect training set accuracy.****This tendency of the models is called overfitting****, and these models do not learn much, hence can not apply their knowledge further.*

# Relation of overfitting with model parameters:

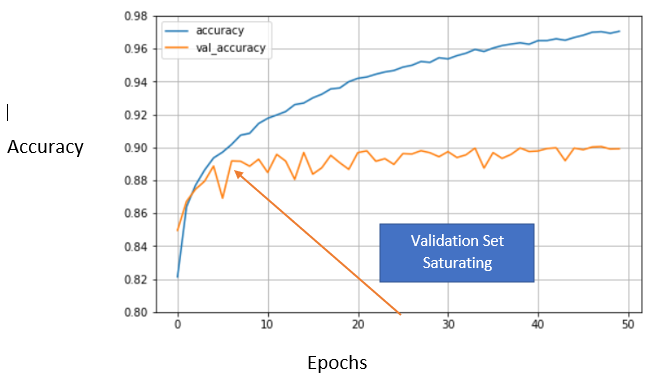


*Here, comes the most important part, the model with more number of parameters is more prone to over-fitting and****as neural networks have more number of parameters it is more exposed to overfitting.***

**Regularization and Early Stopping:**

The general set of strategies against this curse of **overfitting**is called **regularization**and early stopping is one such technique.

The idea is very simple. The model tries to chase the loss function crazily on the training data, by tuning the parameters. Now, we keep another set of data as the validation set and as we go on training, we keep a record of the loss function on the validation data, and when we see that there is no improvement on the validation set, we stop, rather than going all the epochs. This strategy of stopping early based on the validation set performance is called **Early Stopping.**This is explained with the below diagram.



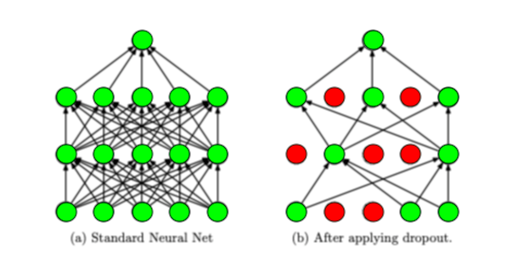
From Figure 3, it can be observed

* The training set accuracy continues to increase, through all the Epochs
* The validation set accuracy, however, saturates between 8 to 10 epochs. This is where the model can be **stopped**training.

*Early Stopping, hence does not only protect against overfitting but needs considerably less number of Epoch to train.*

**Dropout regularization**

Dropout regularization is a technique to prevent neural networks from overfitting. Dropout works by randomly disabling neurons and their corresponding connections. This prevents the network from relying too much on single neurons and forces all neurons to learn to generalize better



When we apply dropout to a neural network, we’re creating a “thinned” network with unique combinations of the units in the hidden layers being dropped randomly at different points in time during training. Each time the gradient of our model is updated, we generate a new thinned neural network with different units dropped based on a probability hyperparameter p.