**Optimizer**

Optimizers are methods used to minimize an error function (loss function) or to maximize the efficiency of production. Optimizers are mathematical functions which are dependent on model’s learnable parameters i.e Weights & Biases. Optimizers help to know how to change weights and learning rate of neural network to reduce the losses.

**Optimization with labeled training**

Although there are a few ways to train neural networks, models on the platform are trained using labeled training. This means that need to provide a [examples](https://peltarion.com/knowledge-center/documentation/glossary#Example) of input [features](https://peltarion.com/knowledge-center/documentation/glossary#Feature) – and the correct answer that the model should predict for each of them, namely, the [label](https://peltarion.com/knowledge-center/documentation/glossary#Label).

Models have weights, sometimes referred to as [parameters](https://peltarion.com/knowledge-center/documentation/glossary#Parameter), which are the coefficients of its mathematical functions.

During [training](https://peltarion.com/knowledge-center/documentation/glossary#Training), the model weights are optimized so that the output prediction is as close as possible to the provided label (label is synonymous to [target](https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/target)). Deep networks have many (thousands to millions) weights.

## **Model loss**

The loss is a measure of how much error the model is committing, averaged over all training [examples](https://peltarion.com/knowledge-center/documentation/glossary#Example).

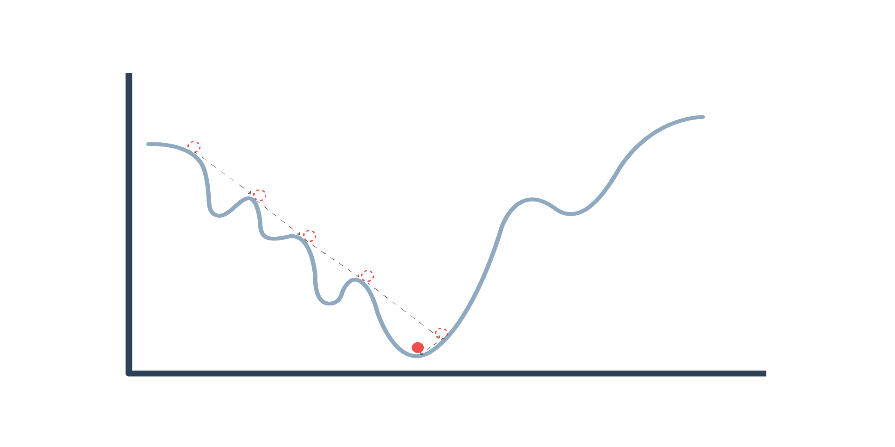
The loss is calculated from the difference between the model output and the provided label using a [loss function](https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions). Different loss functions have different properties, which make them more or less appropriate for different use cases.

With every loss function, the loss is positive and large when the model makes much error. To optimize the model, look at the loss and try to minimize it i.e. as close to 0 as possible.

### **Minimizing the loss**

A very common analogy is that optimizing a model is like rolling a ball on an artificial landscape. In this analogy, the coordinates (x-position in the figure) represent different sets of model weights:

* The altitude of the hill corresponds to the loss, as calculated by a particular loss function.
* The ball is where the model is located at any given time.
* The ball is moved around step by step in an iterative manner.
* The direction of moving is determined by [gradient descent](https://peltarion.com/knowledge-center/documentation/glossary#Gradient_descent), and the size of the steps by the [learning rate](https://peltarion.com/knowledge-center/documentation/glossary#Learning_rate).



Ball rolling in an artificial landscape

### **Backpropagation**

To compute the gradient of the loss over every parameter (i.e. how much it decreases for every possible direction in the landscape) in a neural network we use a method called backpropagation.

Backpropagation starts from the model loss, and calculates its derivative with respect to the weights that are in the last model layer. Derivatives are then calculated with respect to every other layer, working backward from the last to the first layer of the model.

### **Gradient descent**

Once the gradient of the loss is known for every possible direction, i.e. variation of every weight, the model weights are updated in the direction that provides the largest decrease of the loss. In the analogy, this is pushing the ball in the direction of steepest descent.

Although this principle is always the same, how the gradient descent is implemented numerically depends on the optimizer. Different optimizers may have different behavior with different types of problems which lead to faster or better decrease of the loss, but we find that [Adam optimizer](https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimizers/adam) is a good all-round optimizer.

There are also a few settings that affect gradient descent (or its iterative behavior), such as [learning rate](https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimization-principles-(in-deep-learning)#Learning_Rate), [batch size](https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimization-principles-(in-deep-learning)#Batch_size) and [momentum](https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimization-principles-(in-deep-learning)#Momentum).

## **Batch size**

Batch size determines how many samples that should be calculated at the same time. The larger batch size we have, the better estimate we will get of the true gradients. However, the batch size is usually limited by GPU memory.

When backpropagation first was invented people were trying to find the best batch size. The question was, is full batch the best or only one sample at a time?

The problem was that, on one hand, gradient descent becomes unnecessarily slow when the number of samples grows, but on the other hand, when the batch size is very small the gradient estimates becomes very noisy and we have to use a small learning rate, which in turn makes progress slow.

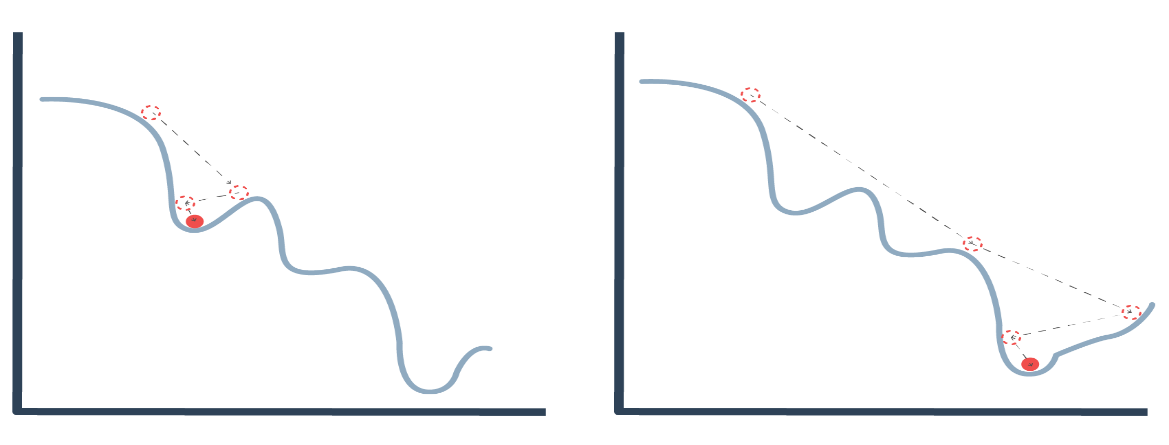
Another negative thing with very small batch size is that the GPUs become under-utilized. Generalization error is often best for a batch size of 1.

## **Learning rate**

The learning rate is controlling the size of the update steps along the gradient. This parameter sets how much of the gradient you update with, where 1 = 100% but normally you set much smaller learning rate, e.g., 0.001.

In our rolling ball analogy, we’re calculating where the ball should roll next in discrete steps (not continuous). How long these discrete steps are is the learning rate.

Choosing a good learning rate is important when training a neural network. If the ball rolls carefully with a small learning rate we can expect to make consistent but very small progress (this corresponds to having a small learning rate). The risk though is that the ball gets stuck in a local minima not reaching the global minima.



Learning rate

Larger steps mean that the weights are changed more every iteration, so that they may reach their optimal value faster, but may also miss the exact optimum.  
Smaller steps mean that the weights are changed less every iteration, so it may take more epochs to reach their optimal value, but they are less likely to miss optima of the loss function.

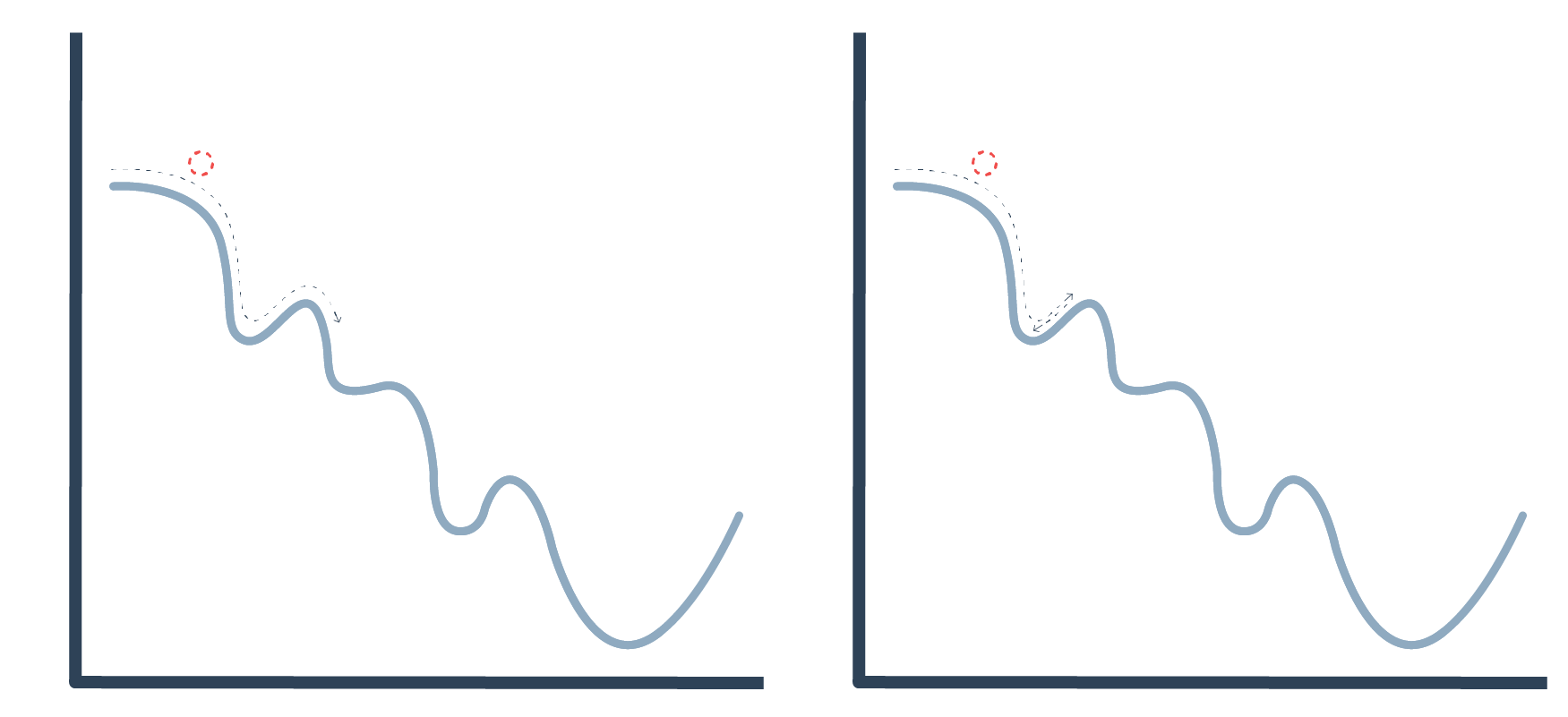
[Learning rate scheduling](https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model/optimization-principles-(in-deep-learning)/learning-rate-schedule) allows you to use large steps during the first few epochs, then progressively reduce the step size as the weights come closer to their optimal value.

## **Momentum**

Momentum is a method that helps accelerate the optimizer in the relevant direction and dampens oscillations. The momentum term increases for dimensions whose gradients point in the same directions, and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way. If we don’t use momentum the ball gets no information on where it was before each discrete calculation step. Without momentum, each new calculation will only be based on the gradient, no history.

In this way, momentum helps the optimizer not to get stuck in local minima.



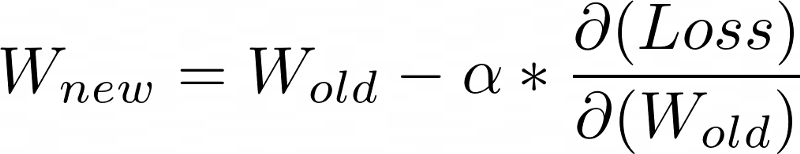
Momentum

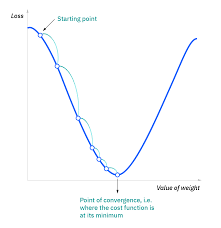
# Types of optimizers

1. **Gradient descent**

**Gradient descent** (GD) is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function. This method is commonly used in machine learning(ML) anddeep learning (DL) to minimise a cost/loss function (e.g. in a linear regression).

Gradient descent is an optimization algorithm based on a convex function and tweaks its parameters iteratively to minimize a given function to its local minimum. Gradient Descentiteratively reduces a loss function by moving in the direction opposite to that of steepest ascent. It is dependent on the derivatives of the loss function for finding minima. uses the data of the entire training set to calculate the gradient of the cost function to the parameters which requires large amount of memory and slows down the process.





**Advantages of Gradient Descent**

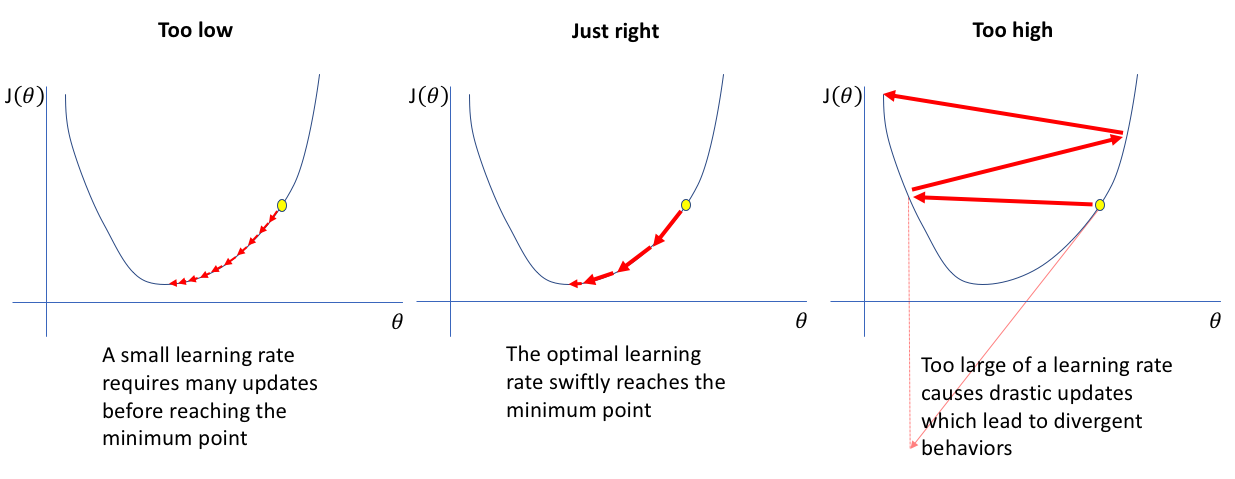
1. Easy to understand
2. Easy to implement

**Disadvantages of Gradient Descent**

1. Because this method calculates the gradient for the entire data set in one update, the calculation is very slow.
2. It requires large memory and it is computationally expensive.

# Learning Rate

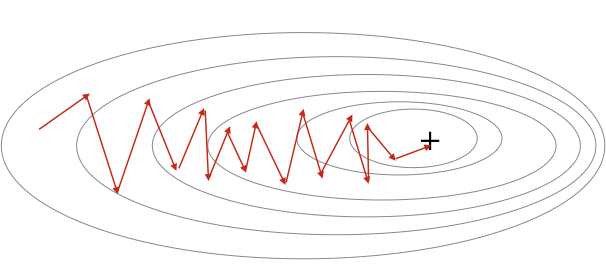
How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



Learning Rate

# 2.Stochastic Gradient Descent

It is a variant of Gradient Descent. It update the model parameters one by one. If the model has 10K dataset SGD will update the model parameters 10k times.



Stochastic Gradient Descent

**Advantages of Stochastic Gradient Descent**

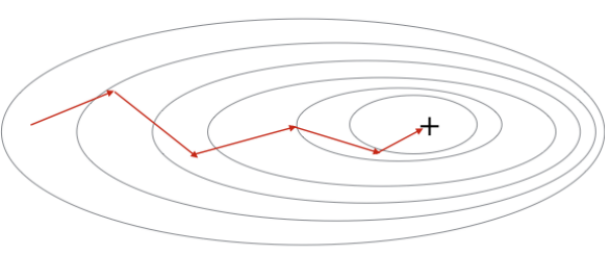
1. Frequent updates of model parameter
2. Requires less Memory.
3. Allows the use of large data sets as it has to update only one example at a time.

**Disadvantages of Stochastic Gradient Descent**

1. The frequent can also result in noisy gradients which may cause the error to increase instead of decreasing it.
2. High Variance.
3. Frequent updates are computationally expensive.

# 3. Mini-Batch Gradient Descent

It is a combination of the concepts of SGD and batch gradient descent. It simply splits the training dataset into small batches and performs an update for each of those batches. This creates a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent. it can reduce the variance when the parameters are updated, and the convergence is more stable. It splits the data set in batches in between 50 to 256 examples, chosen at random.



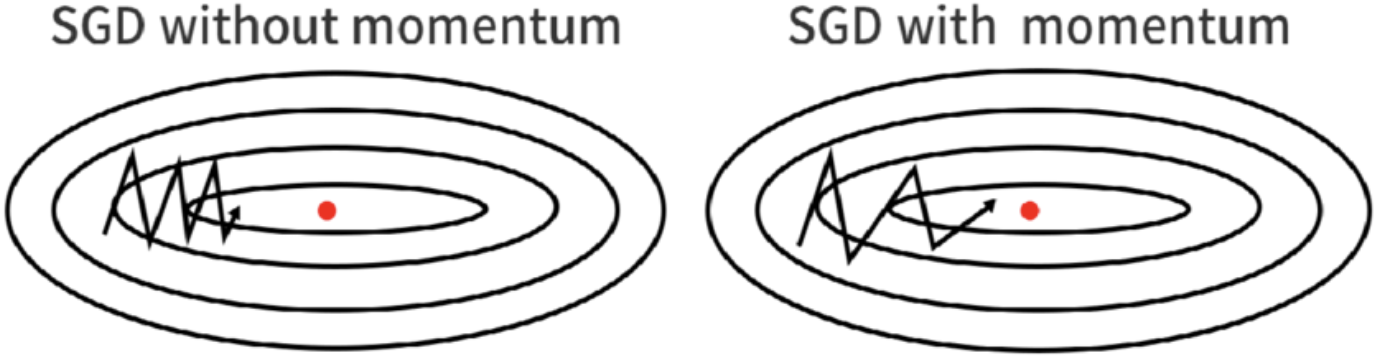
**Advantages of Mini Batch Gradient Descent:**

1. It leads to more stable convergence.
2. more efficient gradient calculations.
3. Requires less amount of memory.

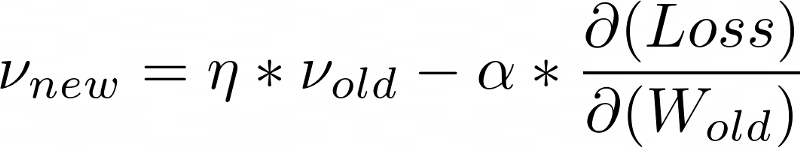
**Disadvantages of Mini Batch Gradient Descent**

1. Mini-batch gradient descent does not guarantee good convergence,
2. If the learning rate is too small, the convergence rate will be slow. If it is too large, the loss function will oscillate or even deviate at the minimum value
3. **SGD with Momentum**

**SGD with Momentum** is a stochastic optimization method that adds a momentum term to regular stochastic gradient descent. Momentum simulates the inertia of an object when it is moving, that is, the direction of the previous update is retained to a certain extent during the update, while the current update gradient is used to fine-tune the final update direction. In this way, you can increase the stability to a certain extent, so that you can learn faster, and also have the ability to get rid of local optimization.



SGD with Momentum



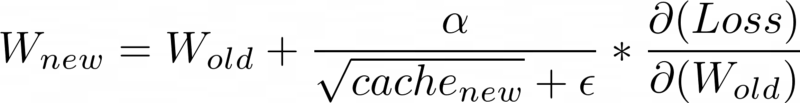
**Advantages of SGD with momentum**

1. Momentum helps to reduce the noise.
2. Exponential Weighted Average is used to smoothen the curve.

**Disadvantage of SGD with momentum**

1. Extra hyperparameter is added.
2. **AdaGrad(Adaptive Gradient Descent)**

In all the algorithms that we discussed previously the learning rate remains constant. The intuition behind AdaGrad is can we use different Learning Rates for each and every neuron for each and every hidden layer based on different iterations.



**Advantages of AdaGrad**

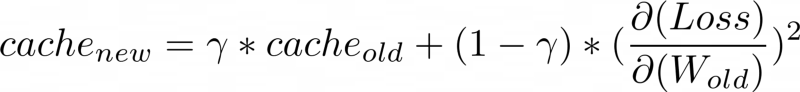
1. Learning Rate changes adaptively with iterations.
2. It is able to train sparse data as well.

**Disadvantage of AdaGrad**

1. If the neural network is deep the learning rate becomes very small number which will cause dead neuron problem.

**6. RMS-Prop (Root Mean Square Propagation)**

RMS-Prop is a special version of Adagrad in which the learning rate is an exponential average of the gradients instead of the cumulative sum of squared gradients. RMS-Prop basically combines momentum with AdaGrad.



**Advantages of RMS-Prop**

1. In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.

**Disadvantages of RMS-Prop**

1. Slow Learning
2. **AdaDelta**

Adadelta is an extension of Adagrad and it also tries to reduce Adagrad’s aggressive, monotonically reducing the learning rate and remove decaying learning rate problem. In Adadelta we do not need to set the default learning rate as we take the ratio of the running average of the previous time steps to the current gradient.

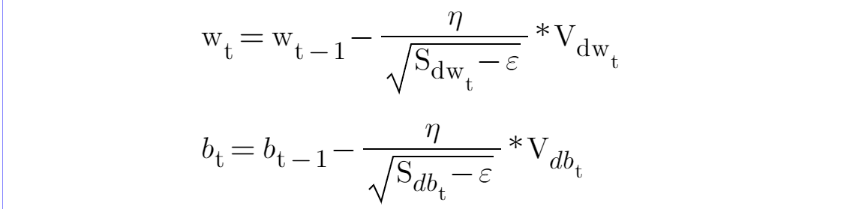
**Advantages of Adadelta**

1. The main advantage of AdaDelta is that we do not need to set a default learning rate.

**Disadvantages of Adadelta**

1. Computationally expensive
2. **Adam(Adaptive Moment Estimation)**

Adam optimizer is one of the most popular and famous gradient descent optimization algorithms. It is a method that computes adaptive learning rates for each parameter. It stores both the decaying average of the past gradients , similar to momentum and also the decaying average of the past squared gradients , similar to RMS-Prop and Adadelta. Thus, it combines the advantages of both the methods.



**Advantages of Adam**

1. Easy to implement
2. Computationally efficient.
3. Little memory requirements.

# How to choose optimizers?

* If the data is sparse, use the self-applicable methods, namely Adagrad, Adadelta, RMSprop, Adam.
* RMSprop, Adadelta, Adam have similar effects in many cases.
* Adam just added bias-correction and momentum on the basis of RMSprop,
* As the gradient becomes sparse, Adam will perform better than RMSprop.