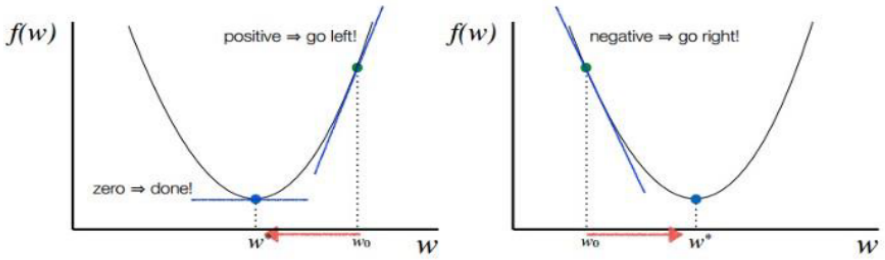
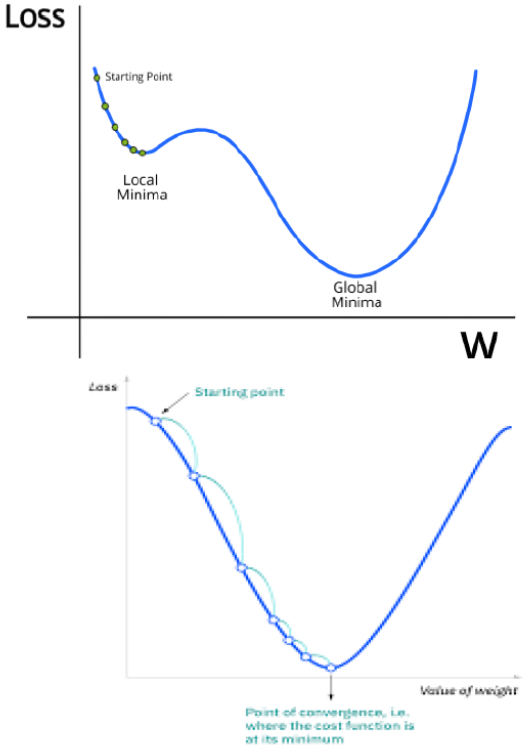
Gradient Descent

- It is dependent on the derivatives of the loss function for finding minima
- It will try to find the least cost function value by updating the weights of your learning algorithm and will come up with the best-suited parameter values corresponding to the Global Minima.
- This is done by moving down the hill with a negative slope, increasing the older weight, and positive slope reducing the older weight.

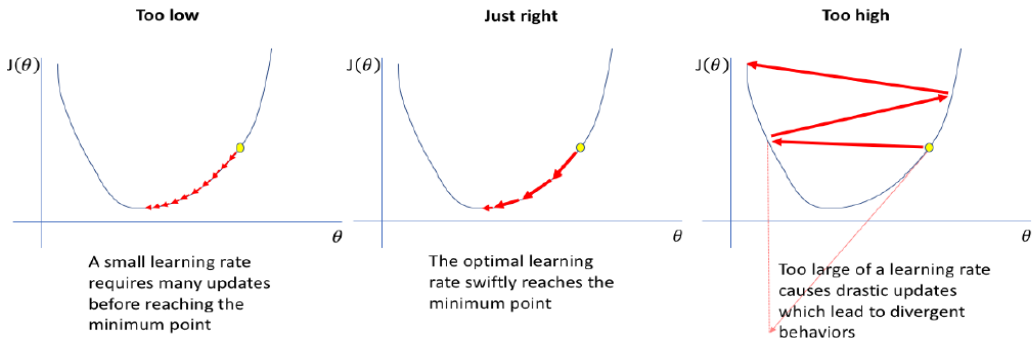


- one big problem with gradient descent. We are not pushing the loss towards the **global minima**, we are merely pushing it towards the closest **local minima.**
- starting from the labeled green dot. Every subsequent green dot represents the loss and new weight value after a **single update** has occurred.
- The gradient descent will only happen till the local minima since the **partial derivative (gradient)** near the local minima is **near zero.** Hence it will stay near there after reaching the local minima and will not try to reach the global minima



Learning Rate

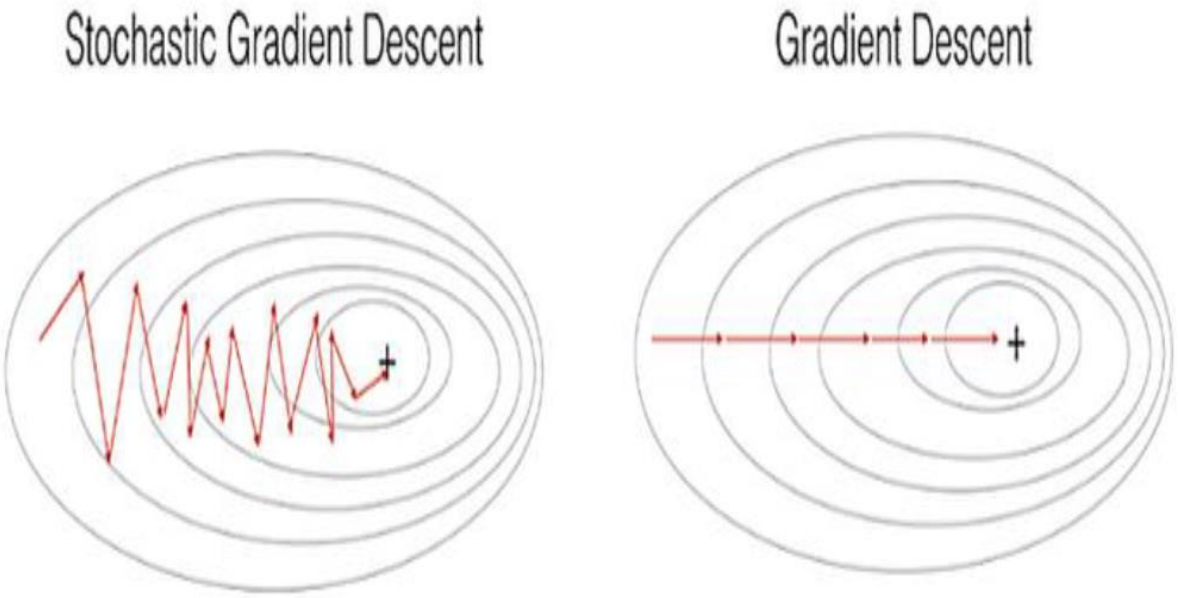
- How big/small the steps are gradient descent takes into the direction of the local minimum are determined by the learning rate, which figures out how fast or slow we will move towards the optimal weights.



- **Advantages of Gradient Descent**
  - Easy to understand
  - Easy to implement
- **Disadvantages of Gradient Descent**
  - May trap at local minima.
  - the calculation is very slow.as Weights are changed after calculating gradient on the whole dataset.
  - It requires large memory to calculate gradient on the whole dataset

Stochastic Gradient Descent

- It's a variant of Gradient Descent.
- It tries to update the model's parameters more frequently. In this, the model parameters are altered after computation of loss on each training example.
- So, if the dataset contains 1000 rows SGD will update the model parameters 1000 times in one cycle of dataset instead of one time as in Gradient Descent.
- **$\theta = \theta - \alpha \cdot \nabla J(\theta; x(i); y(i))$, where {x(i) ,y(i)} are the training examples.**



Advantages:

- Frequent updates of model parameters hence, converges in less time.
- Requires less memory as no need to store values of loss functions.
- Get new minima (didn't stopped at the 1st local minima)

Disadvantages:

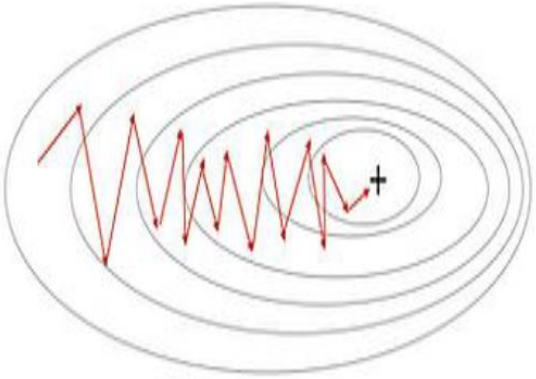
- May continue working even after achieving global minima.
- To get the same convergence as gradient descent needs to slowly reduce the value of learning rate.

Mini-Batch Gradient Descent

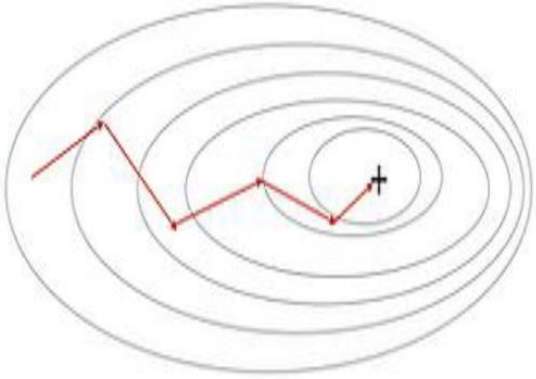
- It's best among all the variations of gradient descent algorithms. It is an improvement on both SGD and standard gradient descent.
- It updates the model parameters after every batch. So, the dataset is divided into various batches and after every batch, the parameters are updated
- $\theta = \theta - \alpha \cdot \nabla J(\theta; B(i))$

  where {B(i)} are the batches of training examples.

Stochastic Gradient Descent



Mini-Batch Gradient Descent



Advantages:

- Frequently updates the model parameters and also has less variance.
- Requires medium amount of memory.

Disadvantages:

- May get trapped at local minima.

- **Batch Gradient Descent : Batch Size = Size of Training Set**
- **Stochastic Gradient Descent: Batch Size = 1**
- **Mini-Batch Gradient Descent: 1 < Batch Size < Size of Training Set**

SGD with momentum

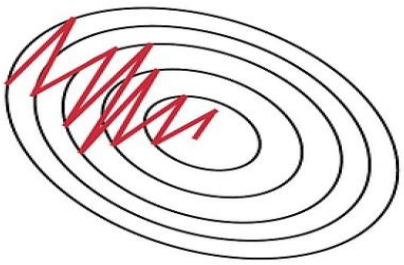
- Momentum was invented for reducing high variance in SGD
- It accelerates the convergence (التقارب)towards the relevant direction and reduces the fluctuation to the irrelevant direction.
- One more hyperparameter is used in this method known as momentum symbolized by '**γ**'.
- **$V(t)=\gamma V(t-1)+\alpha.\nabla J(\theta)$** ... (v is the velocity)
- Now, the weights are updated by **$\theta=\theta-V(t)$.**
- The momentum term **γ** is usually set to 0.9 or a similar value.

Advantages:

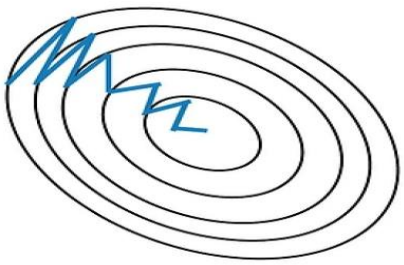
- Reduces the oscillations and high variance of the parameters.
- Converges faster than gradient descent.

Disadvantages:

- One more hyper-parameter is added which needs to be selected manually and accurately.



Stochastic Gradient Descent **withhout** Momentum



Stochastic Gradient Descent **with** Momentum

SGD

SGD class

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False, name="SGD", **kwargs
)
```

Gradient descent (with momentum) optimizer.

Adagrad
(**Ada**ptive **Grad**ient Algorithm)

- All before Optimizers are using constant learning rate for all parameters and for each cycle
- this optimizer uses a different learning rate for each iteration(EPOCH) rather than using the same learning rate for determining all the parameters.

- The epsilon in the denominator is a very small value to ensure division by zero does not occur.

First, each weight has its own **cache** value, which collects the squares of the gradients till the current point.

$$cache_{new} = cache_{old} + (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Cache updation for Adagrad

The cache will continue to increase in value as the training progresses. Now the new update formula is as follows:

$$W_{new} = W_{old} + \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adagrad Update Formula

Advantages:

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

Disadvantage:

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

`Adagrad` class

```
tf.keras.optimizers.Adagrad(
    learning_rate=0.001,
    initial_accumulator_value=0.1,
    epsilon=1e-07,
    name="Adagrad",
    **kwargs
)
```

AdaDelta

- It is an extension of **AdaGrad** which tends to remove the *decaying learning Rate* problem of it

Advantages:

- Now the learning rate does not decay and the training does not stop.

Disadvantages:

- Computationally expensive.

`Adadelta` class

```
tf.keras.optimizers.Adadelta(
    learning_rate=0.001, rho=0.95, epsilon=1e-07, name="Adadelta", **kwargs
)
```

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.
- the decay rate (gamma) value is usually around 0.9 or 0.99.

$$cache_{new} = \gamma * cache_{old} + (1 - \gamma) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Advantages:

- In RMS-Prop learning rate gets adjusted automatically and it chooses a different learning rate for each parameter.
- the learning rate does not decay too quickly, that allowing training to continue for much longer time

Disadvantages:

- Slow Learning

RMSprop class

```
tf.keras.optimizers.RMSprop(
    learning_rate=0.001,
    rho=0.9,
    momentum=0.0,
    epsilon=1e-07,
    centered=False,
    name="RMSprop",
    **kwargs
)
```

Adam
(Adaptive Moment Estimation)

- Adam optimizer is one of the most popular and famous gradient descent optimization algorithms.
- The idea behind Adam optimizer is to utilize the momentum concept from "SGD with momentum" and adaptive learning rate from "Ada delta".
- Also we don't want to roll so fast just because we can jump over the minimum, we want to decrease the velocity a little bit for a careful search

- **Advantages of Adam**
- Easy to implement
- Computationally efficient.
- Little memory requirements.

Adam is a little like combining RMSProp with Momentum. First we calculate our m value, which will represent the momentum at the current point.

$$m_{new} = \beta_1 * m_{old} - (1 - \beta_1) * \frac{\partial(Loss)}{\partial(W_{old})}$$

Adam Momentum Update Formula

The only difference between this equation and the momentum equation is that instead of the learning rate we keep (1-Beta_1) to be multiplied with the current gradient.

Next we will calculate the accumulated cache, which is exactly the same as it is in RMSProp:

$$cache_{new} = \beta_2 * cache_{old} + (1 - \beta_2) * (\frac{\partial(Loss)}{\partial(W_{old})})^2$$

Now we can get the final update formula:

$$W_{new} = W_{old} - \frac{\alpha}{\sqrt{cache_{new}} + \epsilon} * m_{new}$$

Adam weight updation formula

Adam class

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

Arguments

```
opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

```
# pass optimizer by name: default parameters will be used
model.compile(loss='categorical_crossentropy', optimizer='adam')
```