ALL about optimizers

Table of contents

1. Load data from tensorflow.
2. Building model.
3. What are optimzers?
4. optimizers
    A. Gradient Descent
    B. Stochastic Gradient Descent
    C. Momentum
    D. AdaGrad
    E. AdaDelta
    F. Adam
    G. Mini-Batch Gradient Descent
    H. Root Mean Squared Propagation (RMSprop)
    I. Nesterov Accelerated Gradient (NAG)
    J. Follow The Regularized Leader (FTRL)

In [9]:
```
# import necessary libraries
```

Load Dataset

In [10]:
```
# Get data mnist data from tensorflow
```

In [11]:
```
# load data in x_train, y_train, x_tets, y_test


# scale the data by dividing dataset by 255.0
```

Building model

In [12]:
```
# create a modle using sequential
model = Sequential([
            # add Flatten layer with input shape 28,28

            # add dense layer with units 128 and activation relu

            # add a dropout layer

            # add a dense layer with number of class as units.


])
```

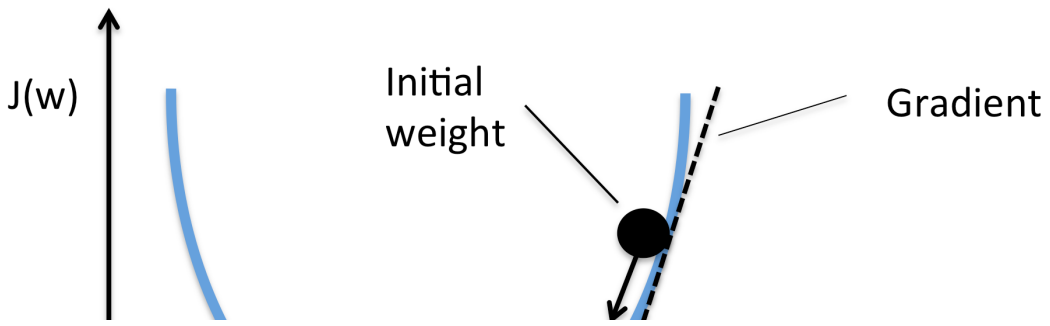
In [13]:
```
# print model summary
```

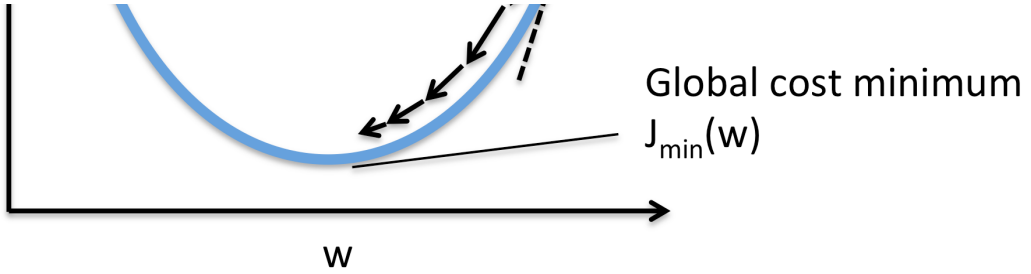
```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_1 (Flatten)         (None, 784)               0

 dense_2 (Dense)             (None, 128)               100480

 dropout_1 (Dropout)         (None, 128)               0

 dense_3 (Dense)             (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

Optimizers

Optimizers are the expanded class, which includes the method to train your machine/deep learning model. Right optimizers are necessary for your model as they improve training speed and performance, Now there are many optimizers algorithms we have in TensorFlow library and we will learn to implement those.

Gradient Descent





Gradient Descent is the most basic but most used optimization algorithm. It is used heavily in linear regression and classification algorithms. Backpropagation in neural networks also uses a gradient descent algorithm.

Gradient descent is a first-order optimization algorithm which is dependent on the first order derivative of a loss function.
It calculates that which way the weights should be changed so that the function can reach a minima.
The loss is transferred from one layer to another and the model's parameters also known as weights are modified depending on the losses so that the loss can be minimized, with the help backpropagation.

Formula:

Cost function

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

gradient descent

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$

here,

Hypothesis: h(x) = theta0 + (theta1)(x)

Parameters: theta0, theta1

Note: We don't use simple gradient descent in deeplearning instead we use Stochastic gradient descent

Stochastic Gradient Descent

Stochastic gradient descent algorithms are modification of gradient descent. It is a popular algorithm for training a wide range of models in machine learning, including (linear) support vector machines, logistic regression. It is used in neural network when combined with backpropagation algorithm.

In SGD, you calculate the gradient using just a random small part of the observation instead of all of them. This approach reduces computation in some cases.

Formula:

Cost function

$$here \;\; J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\widehat{y^i} - y^i)X_j^i$$

Stochastic gradient descent

$$\theta_j = \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta)$$

Frequently updates the model hence take less time to converge.

Requires less memory usage as there is no need to store values of loss functions.

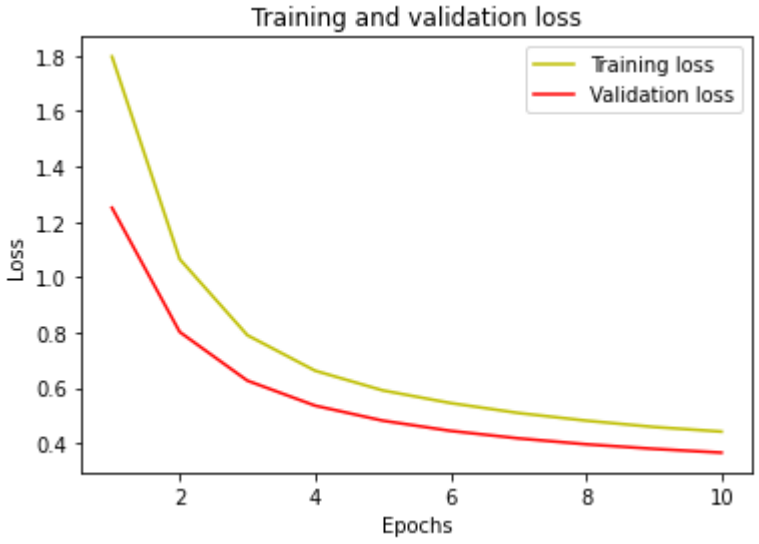
In [ ]:
```
# import SGD from tensorflow library

# set a variable optimizer to SGD(learning_rate = 'of your choice, bu

# compile the model using the optimizer selected with appropriate los
```

We used Sparse categorical cross-entropy as loss function as the classes/labels we have are in interger format.

In [15]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 1.
7971 - val_loss: 1.2509
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 1.
0650 - val_loss: 0.8017
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
7899 - val_loss: 0.6256
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
6616 - val_loss: 0.5353
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
5904 - val_loss: 0.4811
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
5447 - val_loss: 0.4442
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
5088 - val_loss: 0.4172
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
4812 - val_loss: 0.3958
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
4583 - val_loss: 0.3792
```

In [16]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training


# plot line plot of Training Loss


# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of SGD

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD)

Mini-Batch Gradient Descent

It is a imporved version of both gradient descent and stocashtic gradient descent. The functioning of the optimizer is that it updates the model parameters after every batch.

Thus we need to divide the dataset into various batches and then after each batch the model's parameters gets updated.

In this example we took batch size as 20.

Formula:

$$\theta_j = \theta_j - \epsilon \frac{1}{N} \sum_{i=1}^{N} \nabla_{\theta_j} \mathcal{L}(\hat{y}_i, y_i), N = 6$$

In the above formula the N=6 that is the batch size.

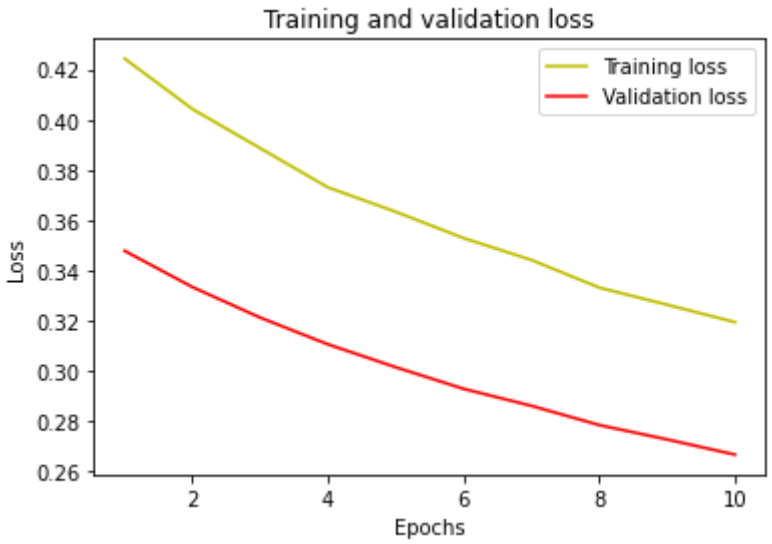
In [17]:
```
# import SGD from tensorflow library

# set a variable optimizer to SGD(learning_rate = 'of your choice, bu

# compile the model using the optimizer selected with appropriate los
```

Fit the model using batch size = 20 you can choose any value in range from 1 to length of train data.

In [18]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
3000/3000 [==============================] - 8s 2ms/step - loss: 0.
4243 - val_loss: 0.3477
Epoch 2/10
3000/3000 [==============================] - 7s 2ms/step - loss: 0.
4042 - val_loss: 0.3333
Epoch 3/10
3000/3000 [==============================] - 8s 3ms/step - loss: 0.
3886 - val_loss: 0.3211
Epoch 4/10
3000/3000 [==============================] - 8s 3ms/step - loss: 0.
3730 - val_loss: 0.3105
Epoch 5/10
3000/3000 [==============================] - 7s 2ms/step - loss: 0.
3633 - val_loss: 0.3014
Epoch 6/10
3000/3000 [==============================] - 12s 4ms/step - loss:
0.3528 - val_loss: 0.2928
Epoch 7/10
3000/3000 [==============================] - 7s 2ms/step - loss: 0.
```

In [19]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```



Momentum

In gradient descent the convergence happens very slowly as the gradinet at the gentle slope becomes smaller thus it takes much more time to update.

Momentu mas invented for this problem.

Now question is, what is Momentum?

Momentum acceleartes the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

How it works?

It looks at current as well as history of updates. If the history and current updates are same that means that the gradient is converging in right direction and it will gain acceleration.

Formual:

Gradient Descent Update Rule

$$w_{t+1} = w_t - \eta \nabla w_t$$

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

As compared to gradient descent the momentum based gradient descent is able to take larger steps because of the momentum it gained in direction it keeps on getting faster.

Note:

Momentum based gradient decent has a problem. It oscillates in and out of the minima valley as the momentum carries it out of the valley, it takes lot of u-turns before finally

converging Despite these u-turns it still converges faster than simple gradient.

in simple terms becuase of momentumit gained it misses the minima valley and take a u-turn to get back to minima and again misses by some distance and keeps trying to get to minima.

Even after having this problem, it reaches the minima faster than simple gradient descent.

The value for the hyperparameter is defined in the range 0.0 to 1.0 and often has a value close to 1.0, such as 0.8, 0.9, or 0.99. A momentum of 0.0 is the same as gradient descent without momentum.

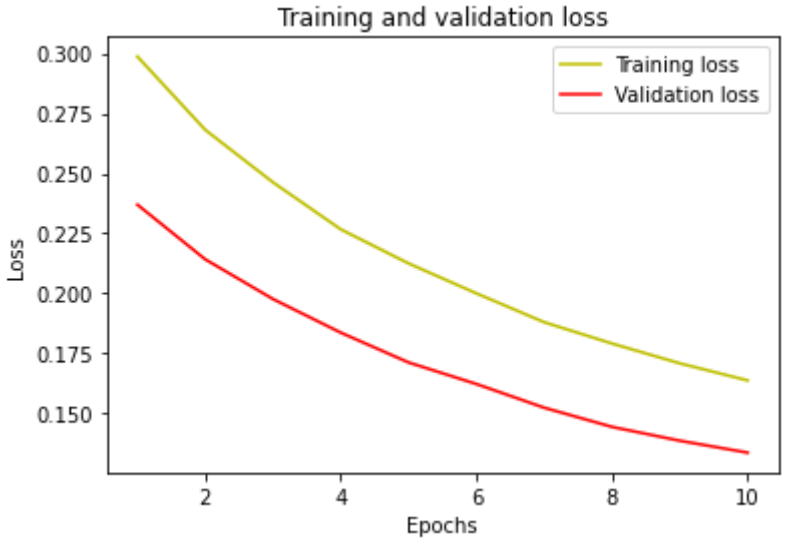
Disadvantage:

In [20]:
```
# import SGD from tensorflow library

# set a variable optimizer to SGD(learning_rate = 'of your choice, bu


# compile the model using the optimizer selected with appropriate los
```

In [21]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
```

In [22]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of SGD with momentum

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD)

AdaGrad (Adaptive gradient)

Assume that there is dataset whci has consisting of both dense features and sparse feature. During training if learning rate is fixed to some value (for eg lr = 0.001) then training happens with same learning rate across the dataset.

The above statement sounds good. But wait there is a problem here.

Problem:

If the feature is dense then there updates will be fatser and when the feature are sparse

then updates will be slower. hence, the convergence is slower because of this problem.

To avoid this problem AdaGrad (Adaptive gradient) was invented.

What is AdaGrad?

In AdaGrad there are different learning rates for different features at each iteration.

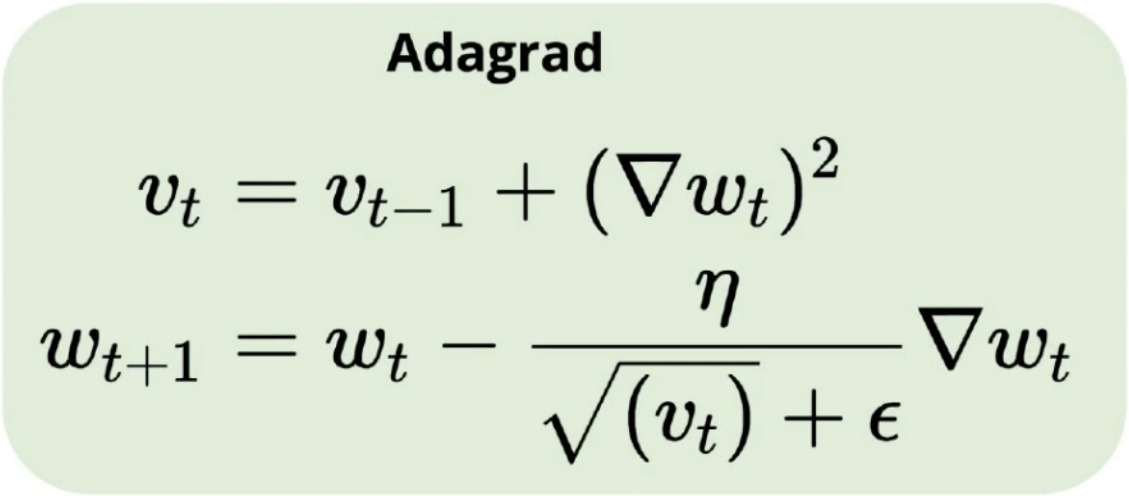
How it works?

If the feature undergone more updates then make learning rate smaller else if the feature undergone lesser updates make learning rate smaller.

For parameters associated with frequently occuring features it performs smaller updates.

For parameters associated with infrequent features it performs larger updates.

Formula:



Note:

There is a problem with this approach. If the vt in denominator becomes larger then learning rate becomes very small resulting in "very small convergence" and "vanishing gradient problem"

In [23]:
```
# import Adagrad from tensorflow library

# set a variable optimizer to Adagrad(learning_rate = 'of your choice

# compile the model using the optimizer selected with appropriate los
```

In [24]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1576 - val_loss: 0.1313
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1540 - val_loss: 0.1302
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1540 - val_loss: 0.1290
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1516 - val_loss: 0.1282
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1509 - val_loss: 0.1274
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1487 - val_loss: 0.1263
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1477 - val_loss: 0.1256
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
1487 - val_loss: 0.1249
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1468 - val_loss: 0.1243
Epoch 10/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
```

In [25]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```

For more details refer the official tensorflow document of Adagrad

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adagrad (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adagrad)

AdaDelta

AdaDelta is a small enhancement

In AdaDelta instead simply using the square root summation of squared history gradient we use exponential decaying average(eda)

So by applying the running average we are controlling the growth of the denominator.

Formula:

$$\mathbb{E}[g^2]_t = \gamma \mathbb{E}[g^2]_{t-1} + (1-\gamma) g_t^2,$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\mathbb{E}[g^2]_t + \epsilon}} \cdot g_t.$$

In simpler words, we are using the mean of the history gradient to decay the learning rate.

In [26]:
```
# import Adadelta from tensorflow library

# set a variable optimizer to Adadelta(learning_rate = 'of your choi

# compile the model using the optimizer selected with appropriate los
```

In [27]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1453 - val_loss: 0.1236
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1455 - val_loss: 0.1236
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1464 - val_loss: 0.1235
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1434 - val_loss: 0.1235
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1465 - val_loss: 0.1235
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1462 - val_loss: 0.1234
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1465 - val_loss: 0.1234
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1460 - val_loss: 0.1233
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1443 - val_loss: 0.1233
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1436 - val_loss: 0.1233
```

In [28]:
```
# Define training loss from history of model

# Define validation loss from history of model
```

```
# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```

For more details refer the official tensorflow document of AdaDelta

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adadelta (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adadelta)

Adam (Adaptive moment estimator):

Idea of Adam optimizer is that instead of only using the exponentially weighted average of square of history of gradients,why can't we use the exponentially weighted average of square of history of gradients.

Formula:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$
$$\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$$
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t}} m_t$$

$$\sqrt{\hat{v}_t} + \epsilon$$

Note:

The reason we are doing this is that we don't want to rely too much on the current gradient and instead rely on the overall behaviour of the gradients over many timesteps

mt → 1st order moment of history of gradients.

vt → 2nd order moment of history of gradients.

In statistics, mt is similar to mean of history gradients and vt is similar to variance of history gradient.

mˆt and vˆt are bias correction for the mean and variance.

Why we need bias correction?

Mean of history gradient ≈(approximately equal) Exponentially weighted average (EWA) of history gradient.(mt) → 1

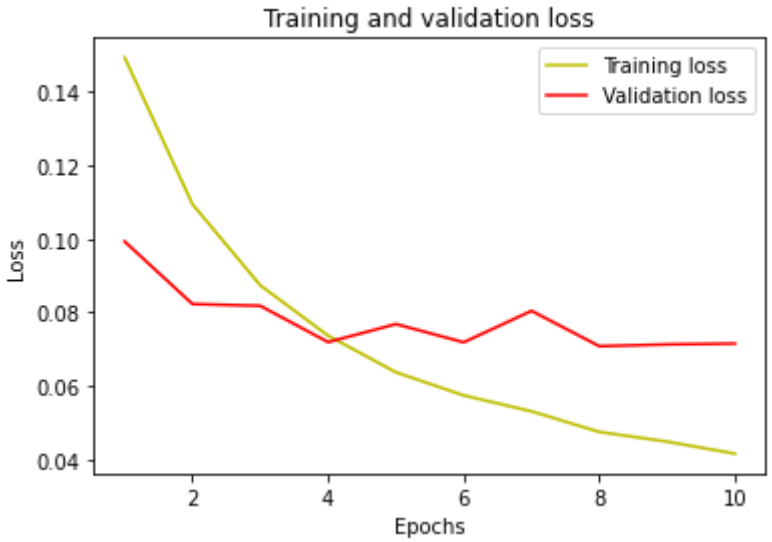
Variance of history gradient ≈(approximately equal) Exponentially weighted average of history gradient.(vt) → 2

In [29]:
```
# import Adam from tensorflow library


# set a variable optimizer to Adam(learning_rate = 'of your choice, l


# compile the model using the optimizer selected with appropriate los
```

In [30]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.
1494 - val_loss: 0.0993
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
1095 - val_loss: 0.0823
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
0874 - val_loss: 0.0818
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
0737 - val_loss: 0.0719
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.
0637 - val loss: 0.0769
```

In [31]:
```
# Define training loss from history of model

# Define validation loss from history of model


# create a list of number of epochs done while training


# plot line plot of Training Loss


# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of Adam

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam)

Root Mean Squared Propagation (RMSprop)

RMSProp is an extension of gradient descent and the AdaGrad version of gradient descent that uses a decaying average of partial gradients in the adaptation of the step size for each parameter.

RMSProp is designed to accelerate the optimization process, e.g. decrease the number of function evaluations required to reach the optima, or to improve the capability of the optimization algorithm, e.g. result in a better final result.

It is related to another extension to gradient descent called Adaptive Gradient, or AdaGrad

RMSProp is a very effective extension of gradient descent and is one of the preferred approaches generally used to fit deep learning neural networks.

Formula:

RMSProp

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

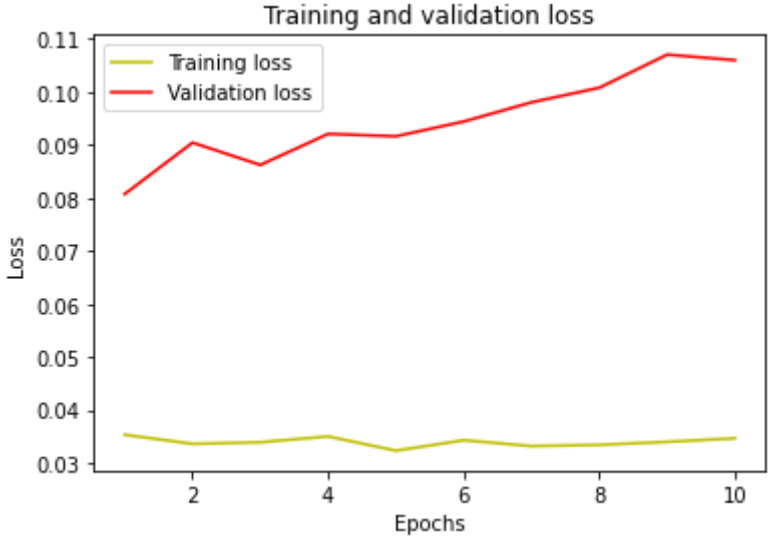
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

In [32]:
```
# import RMSprop from tensorflow library

# set a variable optimizer to RMSprop(learning_rate = 'of your choice

# compile the model using the optimizer selected with appropriate los
```

In [33]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0353 - val_loss: 0.0808
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0336 - val_loss: 0.0905
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0339 - val_loss: 0.0862
Epoch 4/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0350 - val_loss: 0.0921
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0323 - val_loss: 0.0917
Epoch 6/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
0343 - val_loss: 0.0944
Epoch 7/10
```

In [34]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of RMSprop

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/RMSprop)

Nesterov Accelerated Gradient (NAG):

Do you remember Momentum based optimization we just studies above?

It was having a problem of oscialltion at minima.

This problem can be solved using Nesterov Accelerated Gradient.

What is nesterov accelerated gradient?

nesterov accelerated gradient was developed to solve issue with momentum approach.

Formual:

With momentum:

$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$
$$w_{t+1} = w_t - update_t$$

With NAG:

$$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead}$$
$$w_{t+1} = w_t - update_t$$

We follow similar update rule for b_t.

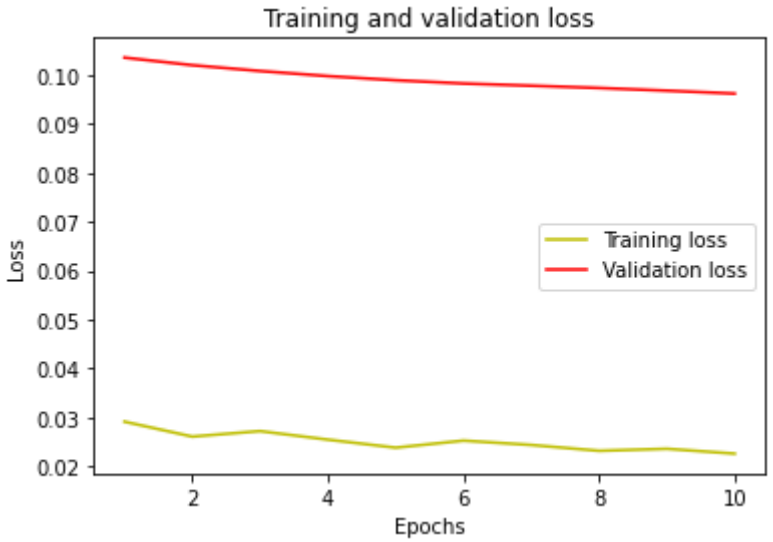
We know we will be using yV(t−1) for modifying the weights so, theta−yV(t−1) approximately tells us the future location. Now, we will calculate the cost based on this future parameter rather than the current one.

In [35]:
```
# import SGD from tensorflow library

# set a variable optimizer to SGD(learning_rate = 'of your choice, bu

# compile the model using the optimizer selected with appropriate los
```

In [36]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0291 - val_loss: 0.1035
Epoch 2/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0260 - val_loss: 0.1020
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0272 - val_loss: 0.1008
Epoch 4/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0254 - val_loss: 0.0998
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0238 - val_loss: 0.0989
Epoch 6/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0252 - val_loss: 0.0982
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0243 - val_loss: 0.0978
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.
0231 - val_loss: 0.0973
```

In [37]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training

# plot line plot of Training Loss

# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of SGD with nesterov

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/SGD)

Nesterov-accelerated Adaptive Moment Estimation (Nadam)

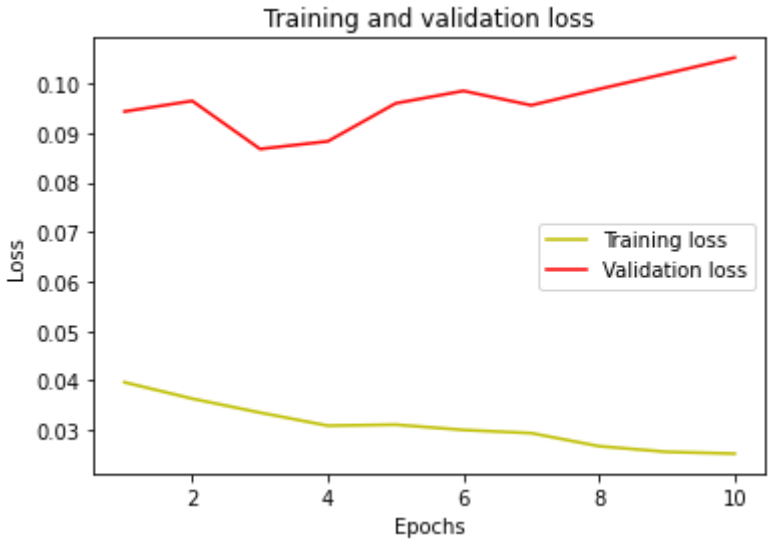
Nadam is an extension of the Adam algorithm that incorporates Nesterov momentum and can result in better performance of the optimization algorithm.

In [38]:
```
# import Nadam from tensorflow library


# set a variable optimizer to Nadam(learning_rate = 'of your choice,

# compile the model using the optimizer selected with appropriate los
```

In [39]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 10s 5ms/step - loss:
0.0397 - val_loss: 0.0943
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.
```

In [40]:
```
# Define training loss from history of model

# Define validation loss from history of model

# create a list of number of epochs done while training


# plot line plot of Training Loss

# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of Nadam

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Nadam (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Nadam)

Follow The Regularized Leader (FTRL)

It is an optimization algorithm developed at Google for click-through rate prediction in the early 2010s. FTRL is a online learning algorithms. In online learning, the learner observes a sequence of functions fi's, which can be deterministic, stochastic, or even adversarially chosen. It is most suitable for shallow models with large and sparse feature spaces

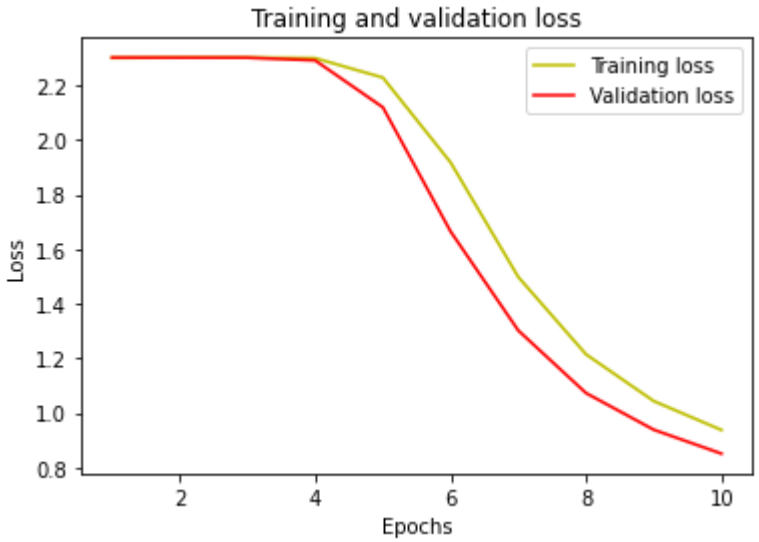
In [41]:
```
# import Ftrl from tensorflow library


# set a variable optimizer to Ftrl(learning_rate = 'of your choice, l

# compile the model using the optimizer selected with appropriate los
```

In [42]:
```
# fit the model for 10 epochs and store the result in history variab
```

```
Epoch 1/10
1875/1875 [==============================] - 8s 4ms/step - loss: 2.
3011 - val_loss: 2.3022
Epoch 2/10
1875/1875 [==============================] - 7s 4ms/step - loss: 2.
3022 - val_loss: 2.3021
Epoch 3/10
1875/1875 [==============================] - 7s 4ms/step - loss: 2.
3020 - val_loss: 2.3019
Epoch 4/10
1875/1875 [==============================] - 7s 4ms/step - loss: 2.
3002 - val_loss: 2.2919
Epoch 5/10
1875/1875 [==============================] - 7s 4ms/step - loss: 2.
2289 - val_loss: 2.1201
Epoch 6/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.
9180 - val_loss: 1.6651
Epoch 7/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.
4981 - val_loss: 1.3021
Epoch 8/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.
2148 - val_loss: 1.0729
Epoch 9/10
1875/1875 [==============================] - 7s 4ms/step - loss: 1.
0436 - val_loss: 0.9386
Epoch 10/10
1875/1875 [==============================] - 7s 4ms/step - loss: 0.
9375 - val_loss: 0.8510
```

In [43]:
```
# Define training loss from history of model

# Define validation loss from history of model


# create a list of number of epochs done while training


# plot line plot of Training Loss
```

```
# plot line plot of Validation Loss
```



For more details refer the official tensorflow document of FTRL

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ftrl (https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Ftrl)

Assignment Summary

When creating a neural network we must select a apropriate optimizer:

following are the optimizers discussed in this assignment

1. Gradient Descent
2. Stochastic Gradient Descent
3. Momentum
4. AdaGrad
5. AdaDelta
6. Adam
7. Mini-Batch Gradient Descent
8. Root Mean Squared Propagation (RMSprop)
9. Nesterov Accelerated Gradient (NAG)
10. Follow The Regularized Leader (FTRL)

we have learned all the optimizers used in neural network and their properties with code

Please fill the below feedback form about this assignment

https://forms.zohopublic.in/cloudyml/form/CloudyMLDeepLearningFeedbackForm/formperma/VCFbldnXAnbcgAIl0lWv2blgHdSldheO4RfktMdgK7s (https://forms.zohopublic.in/cloudyml/form/CloudyMLDeepLearningFeedbackForm/formperma/VCFbldnXAnbcgAIl0lWv2blgHdSldheO4RfktMdgK7s)