In [1]:
```
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
import pandas as pd
import seaborn as sns
import random
from matplotlib.pyplot import figure
random.seed(0)
figure(figsize=(15, 6), dpi=80)
%config Completer.use_jedi=False
```

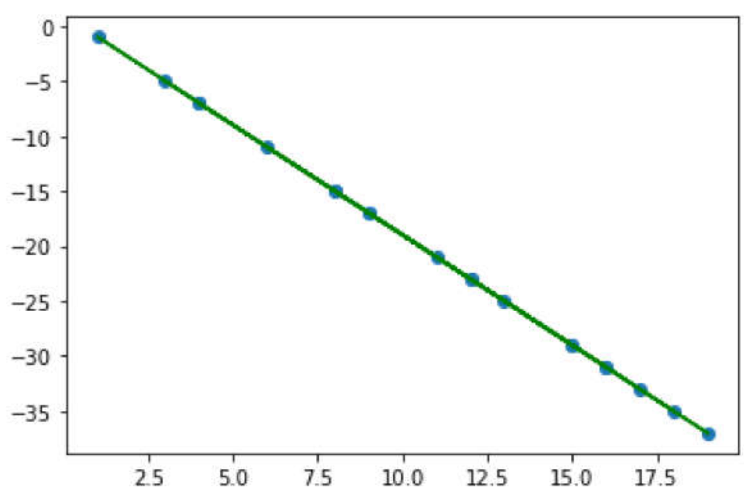
```
<Figure size 1200x480 with 0 Axes>
```

In [2]:
```
from sklearn import linear_model
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
```

In [3]:
```
x=[random.randrange(0,20,1) for i in range(20)]
x=np.array(x)
y=(-2*x) + 1
print(f'x={x}\ny{y}')
print(f'x_shape={x.shape}\ny_shape{y.shape}')
```

```
x=[12 13  1  8 16 15 12  9 15 11 18  6 16  4  9  4  3 19  8 17]
y[-23 -25  -1 -15 -31 -29 -23 -17 -29 -21 -35 -11 -31  -7 -17  -7  -5 -37
 -15 -33]
x_shape=(20,)
y_shape(20,)
```

In [4]:
```
plt.plot(x,y,color="green")
plt.scatter(x,y)
plt.show()
```



In [ ]:
```
#np.column_stack((np.ones(len(x),dtype=int) , x))
```

In [ ]:
```
#y.reshape(-1,1).shape
```

In [ ]:
```
#(x.shape)[0]
```

In [ ]:
```
#np.zeros((np.zeros((x.shape[1],1),1)) # x is matrix inside it x0 , x1
```

In [ ]:
```
#y.reshape(-1,1).shape
```

Batch GD Problems

- ***Standard Gradient descent*** updates the *parameters* only after each epoch i.e. after calculating the *derivatives* for all the observations it updates the *parameters*. This phenomenon may lead to the following ***problems:***
  - It can be very slow for very large datasets because only one-time update for each epoch. Large number of **epochs** is required to have a substantial number of updates.
  - For large datasets, the vectorization of data doesn't fit into **memory**.
  - For non-convex surfaces, it may only find the **local minimums.**

In [ ]:
```

```

1) BATCH GRADIENT DESCENT

In [8]:
```
def Batch_GD(x,y,maxEpochs,learningRate ,convergence):
    loss=[]
    thetaList0=[]
    thetaList1=[]
    ypredictedEpochs=[]
    X=np.column_stack((np.ones(len(x),dtype=int),x)) #more columns x0 ,x1
    y=y.reshape(-1,1)     #(shape(20,1))
    m=(X.shape)[0]        #m=20
    thetas=np.zeros((X.shape[1],1))
    count=0
    epoch=0
    while epoch < maxEpochs:
        count +=1

        ypredicted = X @ thetas # (20,2) @ (2,1) ===> (20,1)
        costOld=(np.sum(np.square(ypredicted - y)))/ (2*m) #Mean Square Error (ol

        Gradient = (np.transpose(X) @ (ypredicted - y) ) / m # (2,20) @ (20,1) ==
        thetas =thetas - (learningRate * Gradient) #(2,1)
        thetaList0.append(thetas[0])
        thetaList1.append(thetas[1])


        ypredicted = X @ thetas # (20,2) @ (2,1) ===> (20,1)
        costNew=(np.sum(np.square(ypredicted - y)))/ (2*m) #Mean Square Error (Ne

        loss.append(costNew) #loss list
        ypredictedEpochs.append(ypredicted)

        if abs(costOld - costNew) < convergence:
            print(f'convergence occur after ({count}) iterations')
            return r2_score(y,ypredicted) ,thetas ,ypredicted ,loss ,thetaList0 ,

        epoch+=1

    print(f'sorru=y Max_epochs ({maxEpochs}) have occured')
    return r2_score(y,ypredicted),thetas ,ypredicted ,loss ,thetaList0 ,thetaList
```

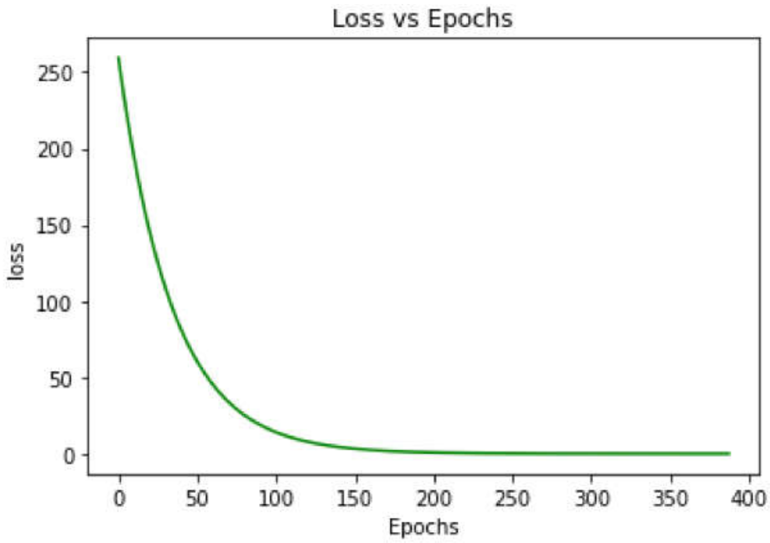
In [9]:
```
R2Score,thetas,ypredicted,loss,thetaList0,thetaList1,ypredictedEpochs=Batch_GD(x,
```

```
convergence occur after (388) iterations
```

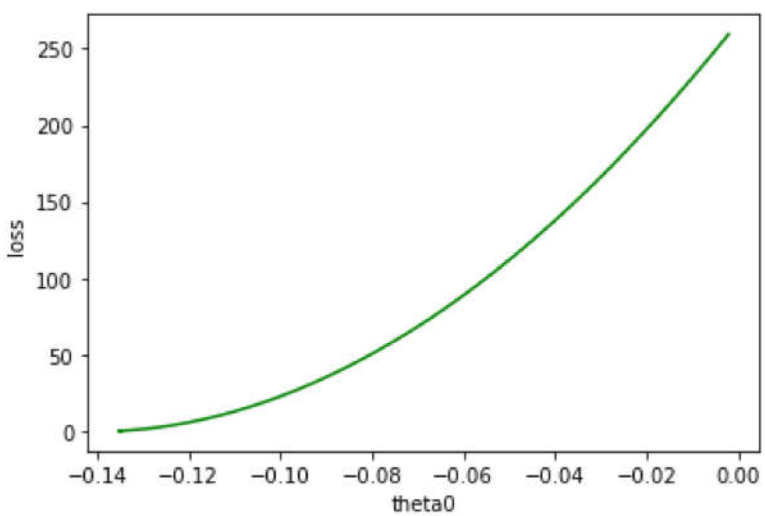
In [10]:
```
R2Score * 100
```

Out[10]:
```
99.77054278129594
```

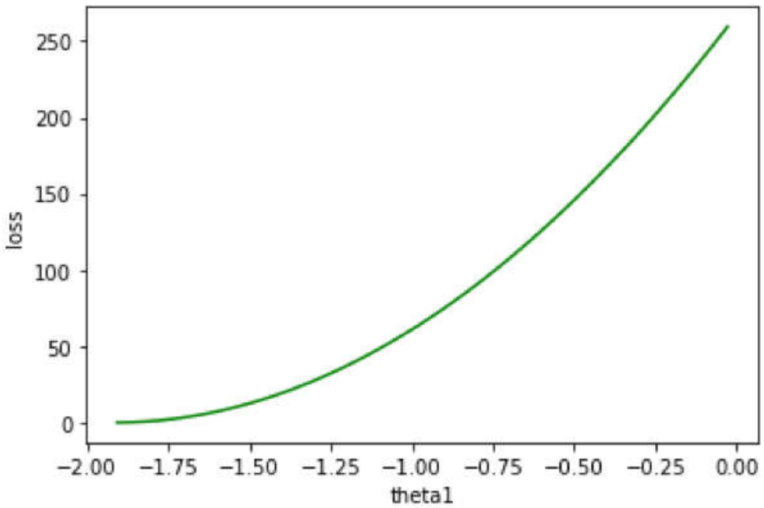
In [11]:
```
plt.plot(loss , color="green")
plt.xlabel("Epochs")
plt.ylabel("loss")
plt.title("Loss vs Epochs")
plt.show()
```



In [12]:
```
plt.plot(thetaList0,loss,color="green")
plt.xlabel("theta0")
plt.ylabel("loss")
plt.show()
```



In [13]:
```
plt.plot(thetaList1,loss,color="green")
plt.xlabel("theta1")
plt.ylabel("loss")
plt.show()
```



In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
    plt.show()
```

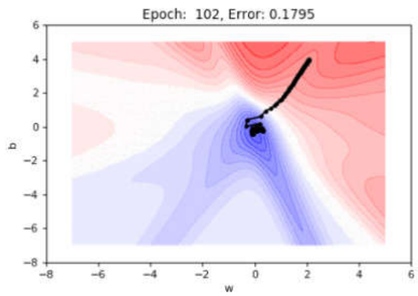
In [ ]:
```
plt.scatter(x,y)
plt.plot(x,ypredicted)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Best regression Line")
plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
```

In [ ]:

Mini Batch GD

- Instead of going over all examples, Mini-batch Gradient Descent sums up over lower number of examples based on the batch size. Therefore, learning happens on each mini-batch of **b** examples:
- $\Theta = \Theta - \boldsymbol{\alpha} \boldsymbol{\nabla}_{\Theta} \boldsymbol{J}\left(\Theta; \boldsymbol{x}^{(i:i+b)}; \boldsymbol{y}^{(i:i+b)}\right)$
- $\boldsymbol{J}(\boldsymbol{\theta_0}, \boldsymbol{\theta_1}) = \frac{1}{2b} \sum_{i=1}^{b} \left(\boldsymbol{h_\theta}\left(\boldsymbol{x}^{(i)}\right) - \boldsymbol{y}^{(i)}\right)^2$
- **Advantages of Mini-batch GD:**
  - Updates are less noisy compared to SGD which leads to better convergence.
  - A high number of updates in a single epoch compared to GD so less number of epochs are required for large datasets.
  - Fits very well to the processor memory which makes computing faster.
- ***Note:*** The batch size is something we can tune. It is usually chosen as power of 2 such as 32, 64, 128, 256, 512, etc.



8

2) MINI BATCH GRADIENT DESCENT

In [ ]:
```
def Mini_Batch_GD(x,y,maxEpochs , batchSize , learningRate , convergence):
    loss=[]
    thetaList0=[]
    thetaList1=[]
    ypredictedEpochs=[]
    X=np.column_stack((np.ones(len(x),dtype=int),x))
    y=y.reshape(-1,1)
    m=(X.shape)[0]
    thetas=np.zeros((X.shape[1],1))
    count=0
    epoch=0

    lossBatch=[]
    ypredictedList=[]
    numberOfBatch=int(m/batchSize)

    while epoch < maxEpochs:
        count +=1
        for i in range(0,m,numberOfBatch):
            ypredicted = X[i:i+numberOfBatch] @ thetas
            ypredictedList.append(ypredicted)

            costOld=(np.sum(np.square(ypredicted - y[i:i+numberOfBatch])))/ (2*nu

            Gradient = (np.transpose(X[i:i+numberOfBatch]) @ (ypredicted - y[i:i+
            thetas =thetas - (learningRate * Gradient)
            thetaList0.append(thetas[0])
            thetaList1.append(thetas[1])


            ypredicted = X[i:i+numberOfBatch] @ thetas
            ypredictedTotal=X@thetas

            costNew=(np.sum(np.square(ypredicted - y[i:i+numberOfBatch])))/ (2*nu
            lossBatch.append(costNew) #loss list


        ypredictedEpochs.append(ypredictedTotal)
        loss.append(costNew)

        if abs(costOld - costNew) < convergence:
            print(f'convergence occur after ({count}) iterations')
            yp=np.concatenate(ypredictedList , axis=0)
            yp=np.reshape(yp[-1*m:],(m,1))
            return r2_score(y,yp) ,thetas[-1] ,yp ,loss ,lossBatch,thetaList0 ,th

        epoch+=1
        yp=np.concatenate(ypredictedList , axis=0)
        yp=np.reshape(y[-1*m:],(m,1))

    print(f'sorry Max_epochs ({maxEpochs}) have occured')
    return r2_score(y,yp) ,thetas[-1] ,yp ,loss ,lossBatch,thetaList0 ,thetaList1
```

In [ ]:
```
R2score ,mthetas ,yp ,mloss ,lossBatch,thetaList0 ,thetaList1 ,ypredictedEpochs =
```

In [ ]:
```
R2score *100
```

In [ ]:
```
plt.plot(mloss,color="green")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("loss vs epochs")
plt.show()
```

In [ ]:
```
plt.plot(thetaList0,lossBatch , color="green")
plt.xlabel("theta0")
plt.ylabel("lossBatch")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
plt.plot(thetaList1,lossBatch , color="green")
plt.xlabel("theta0")
plt.ylabel("lossBatch")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
    plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
```

In [ ]:
```
plt.scatter(x,y)
plt.plot(x,yp)
plt.xlabel("x")
plt.ylabel("y")
plt.title("best lineRegression")
plt.show()
```

In [ ]:

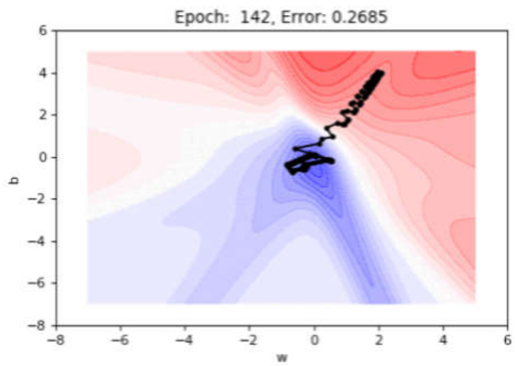
In [ ]:

Stochastic GD (SGD)

- ***Stochastic gradient descent*** updates the parameters for each observation which leads to more number of updates.

$$\Theta = \Theta - \boldsymbol{\alpha}\nabla_{\Theta}\boldsymbol{J}\left(\Theta; \boldsymbol{x}^{(i)}; \boldsymbol{y}^{(i)}\right)$$

$$\boldsymbol{J}(\boldsymbol{\theta_0}, \boldsymbol{\theta_1}) = \left(\boldsymbol{h_\theta}\left(\boldsymbol{x}^{(i)}\right) - \boldsymbol{y}^{(i)}\right)^{\mathbf{2}}$$



- ***Disadvantages of SGD:***
  - Due to frequent fluctuations, it will keep overshooting near to the desired exact minima.
  - Add noise to the learning process i.e. the variance becomes large since we only use 1 example for each learning step.
  - Increase run time.
  - We can’t utilize vectorization over 1 example.

7

3) STOCHASTIC GRADIENT DESCENT

In [ ]:
```
def Stochastic_GD(x,y,maxEpochs , batchSize , learningRate , convergence):
    loss=[]
    thetaList0=[]
    thetaList1=[]
    ypredictedEpochs=[]
    X=np.column_stack((np.ones(len(x),dtype=int),x))
    y=y.reshape(-1,1)
    m=(X.shape)[0]
    thetas=np.zeros((X.shape[1],1))
    count=0
    epoch=0

    lossBatch=[]
    ypredictedList=[]
    numberOfBatch=int(m/batchSize)

    while epoch < maxEpochs:
        count +=1
        for i in range(0,m,numberOfBatch):
            ypredicted = X[i:i+numberOfBatch] @ thetas
            ypredictedList.append(ypredicted)

            costOld=(np.sum(np.square(ypredicted - y[i:i+numberOfBatch])))/ (2*nu

            Gradient = (np.transpose(X[i:i+numberOfBatch]) @ (ypredicted - y[i:i+
            thetas =thetas - (learningRate * Gradient)
            thetaList0.append(thetas[0])
            thetaList1.append(thetas[1])


            ypredicted = X[i:i+numberOfBatch] @ thetas
            ypredictedTotal=X@thetas

            costNew=(np.sum(np.square(ypredicted - y[i:i+numberOfBatch])))/ (2*nu
            lossBatch.append(costNew)


        ypredictedEpochs.append(ypredictedTotal)
        loss.append(costNew)

        if abs(costOld - costNew) < convergence:
            print(f'convergence occur after ({count}) iterations')
            yp=np.concatenate(ypredictedList , axis=0)
            yp=np.reshape(yp[-1*m:],(m,1))
            return r2_score(y,yp) ,thetas[-1] ,yp ,loss ,lossBatch,thetaList0 ,th

        epoch+=1
        yp=np.concatenate(ypredictedList , axis=0)
        yp=np.reshape(yp[-1*m:],(m,1))

    print(f'sorry Max_epochs ({maxEpochs}) have occured')
    return r2_score(y,yp) ,thetas[-1] ,yp ,loss ,lossBatch,thetaList0 ,thetaList1
```

In [ ]:
```
Rscore ,mthetas ,yp ,loss ,lossBatch,thetaList0 ,thetaList1 ,ypredictedEpochs=Stc
```

In [ ]:
```
Rscore *100
```

In [ ]:
```
plt.plot(loss,color="green")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("loss vs epochs")
plt.show()
```

In [ ]:
```
plt.plot(thetaList0,lossBatch , color="green")
plt.xlabel("theta0")
plt.ylabel("lossBatch")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
plt.plot(thetaList0,lossBatch , color="green")
plt.xlabel("theta0")
plt.ylabel("lossBatch")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y,color="green")
    plt.plot(x,h)
    plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
```

In [ ]:
```
plt.scatter(x,y)
plt.plot(x,yp)
plt.xlabel("x")
plt.ylabel("y")
plt.title("best lineRegression")
plt.show()
```

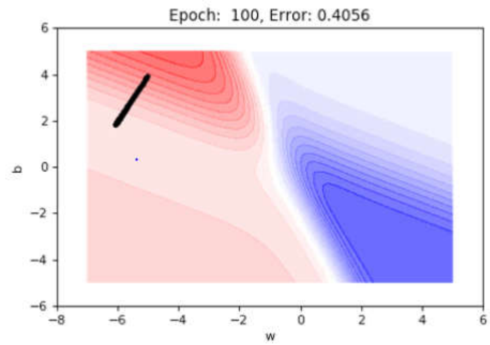
In [ ]:
```

```

MOMENTUM BASED GRADIENT

Better Optimization w.r.t. GD

- Consider a case with initialization in a flat surface where GD is used and the error is not reducing when the gradient is in the flat surface.
- Even after a large number of epochs for e.g. 10000 the algorithm is not converging.
- Due to this issue, the convergence is not achieved so easily and the learning takes too much time.
- To overcome this problem **Momentum based gradient descent** is used.



27

Momentum-based GD

- **Motivation:**
  - Consider a case where in order to reach to your desired destination you are continuously being asked to follow the same direction and once you become confident that you are following the right direction then you start taking ***bigger steps*** and you keep getting ***momentum*** in that same direction.
  - Similar to this if the ***gradient*** is in a ***flat surface*** for long term then rather than taking constant steps it should take ***bigger steps*** and keep the ***momentum*** continue. This approach is known as ***momentum based gradient descent***.

Momentum-based GD

Update your Batch GD for one variable implementation to be Momentum-Based GD and check your results

Momentum based Gradient Descent Update Rule

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

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

In [ ]:
```
def MomentumBased_Batch_GD(x,y,maxEpochs,gama ,learningRate ,convergence):
    loss=[]
    thetaList0=[]
    thetaList1=[]
    ypredictedEpochs=[]
    X=np.column_stack((np.ones(len(x),dtype=int),x))
    y=y.reshape(-1,1)
    m=(X.shape)[0]        #m=20
    thetas=np.zeros((X.shape[1],1))
    count=0
    epoch=0

    v=0
    while epoch < maxEpochs:
        count +=1

        ypredicted = X @ thetas
        costOld=(np.sum(np.square(ypredicted - y)))/ (2*m)

        Gradient = (np.transpose(X) @ (ypredicted - y) ) / m

        v= ( gama*v ) + (learningRate * Gradient)
        thetas =thetas - v  #where v for speed up the update

        thetaList0.append(thetas[0])
        thetaList1.append(thetas[1])


        ypredicted = X @ thetas # (20,2) @ (2,1) ===> (20,1)
        costNew=(np.sum(np.square(ypredicted - y)))/ (2*m) #Mean Square Error (Ne

        loss.append(costNew) #loss list
        ypredictedEpochs.append(ypredicted)

        if abs(costOld - costNew) < convergence:
            print(f'convergence occur after{count} iterations')
            return r2_score(y,ypredicted) ,thetas ,ypredicted ,loss ,thetaList0 ,

        epoch+=1

    print(f'sorry Max_epochs {maxEpochs} have occured')
    return r2_score(y,ypredicted),thetas ,ypredicted ,loss ,thetaList0 ,thetaList
```

In [ ]:
```
R2score,thetas ,ypredicted ,loss ,thetaList0 ,thetaList1 ,ypredictedEpochs = Mome
```

In [ ]:
```
R2score *100
```

In [ ]:
```
plt.plot(loss,color="green")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("loss vs epochs")
plt.show()
```

In [ ]:
```
plt.plot(thetaList0,loss , color="green")
plt.xlabel("theta0")
plt.ylabel("loss")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
plt.plot(thetaList1,loss , color="green")
plt.xlabel("theta0")
plt.ylabel("loss")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
    plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
```

In [ ]:
```
plt.scatter(x,y)
plt.plot(x,ypredicted)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Best regression Line")
plt.show()
```

In [ ]:

In [ ]:

In [ ]:

Nesterov Accelerated GD (NAG)

- In the standard momentum method:
  - **first** computes the gradient at the current position;
  - **then** takes a big jump in the direction of the accumulated gradient.
- In **NAG:**
  - **first** make a big jump in the direction of the previous accumulated gradient;
  - **then** measure the gradient where you end up and make a correction.

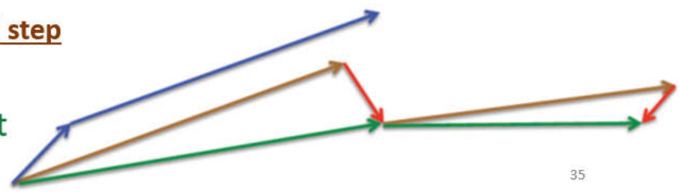
It is always better to correct a mistake after you have made it.

brown vector = jump **looking ahead step**

red vector = correction

green vector = accumulated gradient

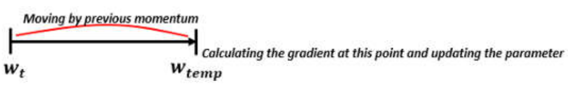
blue vectors = standard momentum



35

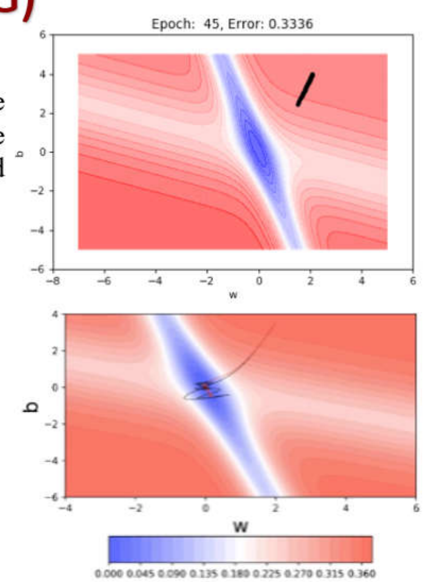
Nesterov Accelerated GD (NAG)

- This looking ahead helps **NAG** in finishing its job (finding the minima) quicker than **momentum-based GD**. Hence the **oscillations** are **less** compared to **momentum based GD** and also there are fewer chances of missing the **minima**.



NAG Update Rule

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



NESTROV ACCELERATED GD(NAG)

Nesterov Accelerated GD (NAG)

Update your Batch GD for one variable implementation to be NAG and check your results

γ takes values between 0 and 1.

- γ takes values between 0 and 1.

NAG Update Rule

$$w_{temp} = w_t - \gamma * v_{t-1}$$

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

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

In [ ]:
```
def NestrovAccelerated_Batch_GD(x,y,maxEpochs,gama ,learningRate ,convergence):
    loss=[]
    thetaList0=[]
    thetaList1=[]
    ypredictedEpochs=[]
    X=np.column_stack((np.ones(len(x),dtype=int),x))
    y=y.reshape(-1,1)
    m=(X.shape)[0]
    thetas=np.zeros((X.shape[1],1))
    count=0
    epoch=0

    v=0
    while epoch < maxEpochs:
        count +=1

        ypredicted = X @ thetas
        costOld=(np.sum(np.square(ypredicted - y)))/ (2*m)

        Gradient = (np.transpose(X) @ (ypredicted - y) ) / m

        theta_temp = thetas - (gama * v)
        ypredicted_temp = X @ theta_temp
        Gradient_temp = (np.transpose(X) @ (ypredicted_temp - y) ) / m
        thetas = theta_temp - (learningRate * Gradient_temp)
        v =  (gama * v) + (learningRate * Gradient_temp)


        thetaList0.append(thetas[0])
        thetaList1.append(thetas[1])


        ypredicted = X @ thetas
        costNew=(np.sum(np.square(ypredicted - y)))/ (2*m)

        loss.append(costNew)
        ypredictedEpochs.append(ypredicted)

        if abs(costOld - costNew) < convergence:
            print(f'convergence occur after ({count}) iterations')
            return r2_score(y,ypredicted) ,thetas ,ypredicted ,loss ,thetaList0 ,

        epoch+=1

    print(f'sorry Max_epochs ({maxEpochs}) have occured')
    return r2_score(y,ypredicted),thetas ,ypredicted ,loss ,thetaList0 ,thetaList
```

In [ ]:
```
R2Score,thetas ,ypredicted ,loss ,thetaList0 ,thetaList1 ,ypredictedEpochs = Nest
```

In [ ]:
```
R2Score
```

In [ ]:
```
plt.plot(loss,color="green")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("loss vs epochs")
plt.show()
```

In [ ]:
```
plt.plot(thetaList0,loss , color="green")
plt.xlabel("theta0")
plt.ylabel("loss")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
plt.plot(thetaList1,loss , color="green")
plt.xlabel("theta0")
plt.ylabel("loss")
plt.title("theta0 vs lossBatch")
plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
    plt.show()
```

In [ ]:
```
for h in ypredictedEpochs:
    plt.scatter(x,y)
    plt.plot(x,h)
```

In [ ]:
```
plt.scatter(x,y)
plt.plot(x,ypredicted)
plt.xlabel("x")
plt.ylabel("y")
plt.title("Best regression Line")
plt.show()
```

In [ ]:
```

```