SAP ID: 6009210033

SHRI VILEPARLE KELAVANI MANDAL'S
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC ACCREDITED with "A" GRADE (CGPA : 3.18)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING (DATA SCIENCE)

COURSE CODE: DJ19DSC501 **DATE: 02/10/2023**

COURSE NAME: Machine Learning - II **CLASS: AY 2023-24**

60009210033

Jhanvi Parekh

D11

LAB EXPERIMENT NO.3

AIM :

Evaluate and analyze Prediction performance using appropriate optimizers for deep learning models.

THEORY:

Optimizers: Optimizers are algorithms or methods used to change the attributes of your neural network such as weights and learning rate in order to reduce the losses. It finds the value of parameters(weights) that minimize the error when mapping inputs to outputs. These optimization algorithms or optimizers widely affect the accuracy of deep learning model and the speed of training of the model.

Types:

1. **Gradient Descent** - most basic but most used optimization 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 altered so that the function can reach a minima. Through backpropagation, 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.

2. **Stochastic Gradient Descent** - 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

3. **Stochastic Gradient descent with momentum** - Momentum was invented for reducing high variance in SGD and softens the convergence. It accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

4. **Mini-Batch Gradient Descent** - 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.

5. **Adagrad** - This optimizer changes the learning rate - uses different learning rates for each iteration. It changes the learning rate 'η' for each parameter and at every time step 't'. It's a type second order optimization algorithm. It works on the derivative of an error function.

6. **RMSProp** - The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minima. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

7. **AdaDelta** - It is an extension of AdaGrad which tends to remove the decaying learning Rate problem of it. Instead of accumulating all previously squared gradients, Adadelta limits the window of accumulated past gradients to some fixed size w. In this exponentially moving average is used rather than the sum of all the gradients.

8. **Adam** - adaptive moment estimation - adam optimizer updates the learning rate for each network weight individually. Adam optimizers inherit the features of both Adagrad and RMS prop algorithms. The intuition behind the Adam is that 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. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, Adam also keeps an exponentially decaying average of past gradients M(t).

Tasks to be performed:

a) **Take the MNIST dataset**
b) **Initialize a neural network basic layers with random weights.**
c) **Perform practical analysis of optimizers on MNIST dataset keeping batch size, and epochs same but with different optimizers.**
d) **Compare the results by choosing 8 different optimizers on a simple neural network**

[gradient desecent, Stochastic Gradient Descent, Stochastic Gradient descent with momentum, Mini-Batch Gradient Descent, Adagrad, RMSProp, AdaDelta, Adam]

e) **List Advantages and Disadvantages of each Optimizer.**

1. **Gradient Descent (GD):**

- Advantages:
  - i. Guaranteed convergence to the global minimum for convex loss functions.
  - ii. Simplicity and ease of implementation.
  - iii. Deterministic updates, which can be useful for debugging.
  - iv. Works well with small and well-behaved datasets.
  - v. No additional hyper parameters to tune.
- Disadvantages:
  - i. Slow convergence, especially for large datasets.
  - ii. Prone to getting stuck in local minima for non-convex functions.
  - iii. Sensitive to the choice of learning rate.
  - iv. Inefficient for high-dimensional spaces.
  - v. Doesn't handle noisy gradients well.

2. **Stochastic Gradient Descent (SGD):**

- Advantages:
  - i. Faster convergence than GD, especially for large datasets.
  - ii. Escape local minima more easily due to the randomness in updates.
  - iii. Can handle non-convex loss functions.
  - iv. Enables online learning (updating the model with each new data point).
  - v. Can generalize well with noisy gradients.
- Disadvantages:
  - i. High variance in updates, making convergence noisy.
  - ii. Might not converge to a fixed point, but oscillate around it.
  - iii. Less deterministic, which can make it harder to reproduce results.
  - iv. Learning rate tuning is often necessary.
  - v. Convergence can be slow when gradients have varying magnitudes.

3. **Stochastic Gradient Descent with Momentum:**

- Advantages:
  - i. Faster convergence compared to SGD.
  - ii. Dampens oscillations and accelerates convergence in noisy gradients.
  - iii. Helps escape local minima.
  - iv. Provides a smoother trajectory towards the minimum.
  - v. Reduces the sensitivity to learning rate choice.
- Disadvantages:
  - i. Requires tuning of the momentum parameter.
  - ii. Can overshoot the minimum in some cases.
  - iii. Can accumulate momentum towards incorrect directions in highly curved regions.
  - iv. May not work well with certain types of non-convex functions.
  - v. Additional hyper parameter to tune.

4. **Mini-Batch Gradient Descent:**

- Advantages:
  - i. Strikes a balance between GD and SGD, suitable for moderate-sized datasets.
  - ii. Faster convergence and reduced noise compared to SGD.
  - iii. Parallelizable and can take advantage of hardware acceleration.
  - iv. Better generalization than SGD.
  - v. Can benefit from vectorised operations.

- Disadvantages:
  1. Requires tuning of the batch size parameter.
  2. May not converge as quickly as SGD on very large datasets.
  3. Batch size selection can impact convergence and memory requirements.
  4. Not as computationally efficient as pure SGD.

5. Adagrad:

- Advantages:
  1. Automatically adapts learning rates for each parameter.
  2. Suitable for sparse data.
  3. Effective for handling noisy gradients.
  4. Low learning rates for frequently updated parameters.
  5. Minimal hyper parameter tuning required.
- Disadvantages:
  1. Learning rates can become too small over time, leading to slow convergence.
  2. Accumulated squared gradients can become large, causing numerical instability.
  3. Inefficient for deep networks with many parameters.
  4. May not perform well with non-stationary data distributions.
  5. Not suitable for all types of problems, e.g., image classification.

6. RMSProp (Root Mean Square Propagation):

- Advantages:
  1. Addresses the problem of Adagrad's decreasing learning rates by using a moving average of squared gradients.
  2. Adaptive learning rates that converge faster than Adagrad.
  3. Suitable for a wide range of problems, including deep learning.
  4. Effective in handling non-stationary data distributions.
  5. Requires minimal hyper parameter tuning compared to some other optimizers.
- Disadvantages:
  1. Requires tuning of the decay rate hyper parameter.
  2. Can still have issues with very small learning rates in some cases.
  3. Sensitive to hyper parameter choices, especially the decay rate.
  4. May not perform optimally when gradients exhibit large variations across parameters.
  5. May require more memory than SGD or simple momentum methods due to the need to store previous squared gradients.

7. AdaDelta:

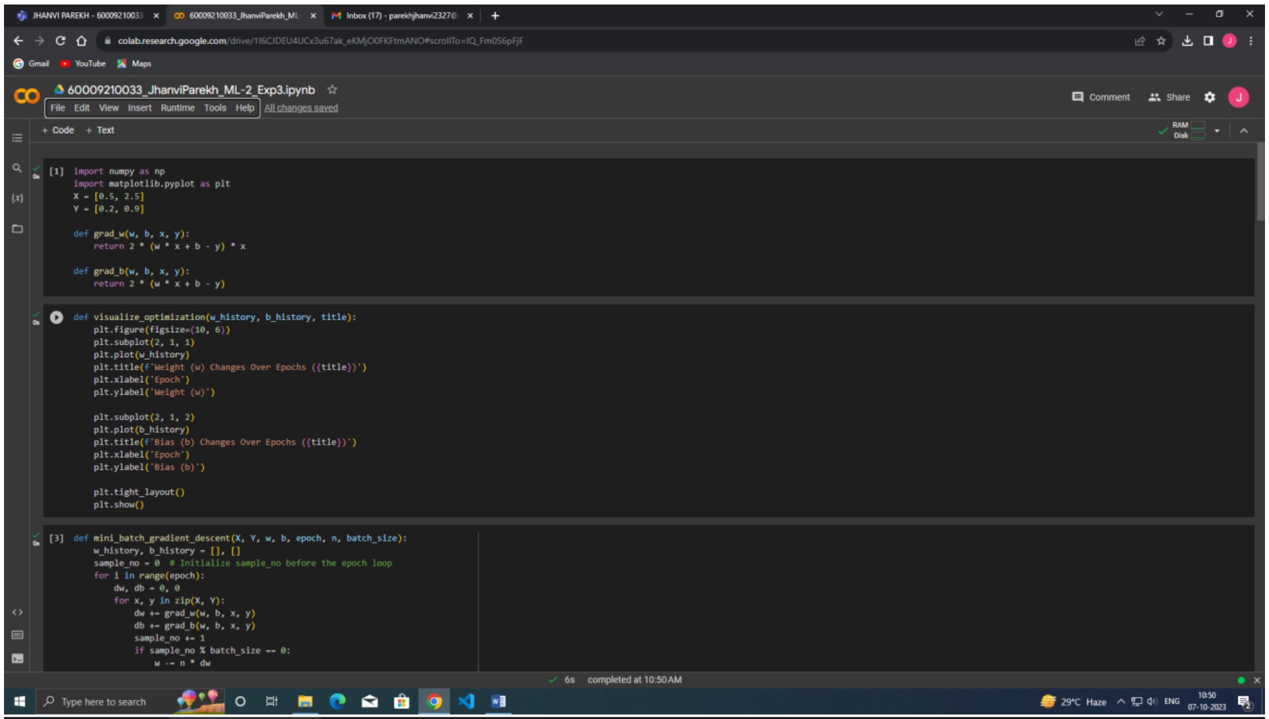
- Advantages:
  1. Eliminates the need for a learning rate hyper parameter entirely.
  2. Adapts learning rates based on recent gradient information.
  3. Robust to the choice of initial learning rate.
  4. Suitable for problems with noisy or sparse gradients.
  5. Generally, converges quickly and effectively.
- Disadvantages:
  1. Requires more memory for storing previous updates compared to some other optimizers.
  2. Less commonly used compared to SGD and Adam, so there may be fewer resources and libraries available.
  3. May not perform as well as Adam on some complex problems with large datasets.

iv. Hyper parameter tuning is still needed for other parameters like the decay rate.
v. May not handle certain non-stationary data distributions as well as RMSProp or Adam.

8. Adam (Adaptive Moment Estimation):

- Advantages:
  i. Combines the benefits of both momentum and adaptive learning rates.
  ii. Fast convergence, widely used, and generally robust.
  iii. Works well in practice with default hyper parameter settings.
  iv. Effective for a wide range of neural network architectures.
  v. Performs well on a variety of tasks, making it a popular choice.
- Disadvantages:
  i. Requires tuning of hyper parameters, including the learning rate, momentum, and decay rates.
  ii. Sensitive to the choice of initial learning rate.
  iii. May exhibit convergence issues on certain types of problems if hyper parameters are not carefully chosen.
  iv. Can converge to suboptimal solutions if the learning rate is not properly tuned.
  v. Can have increased memory requirements compared to simpler optimizers like SGD.

LINK:
https://colab.research.google.com/drive/1I6CJDEU4UCx3u67ak_eKMjO0FKFtmANO?usp=sharing

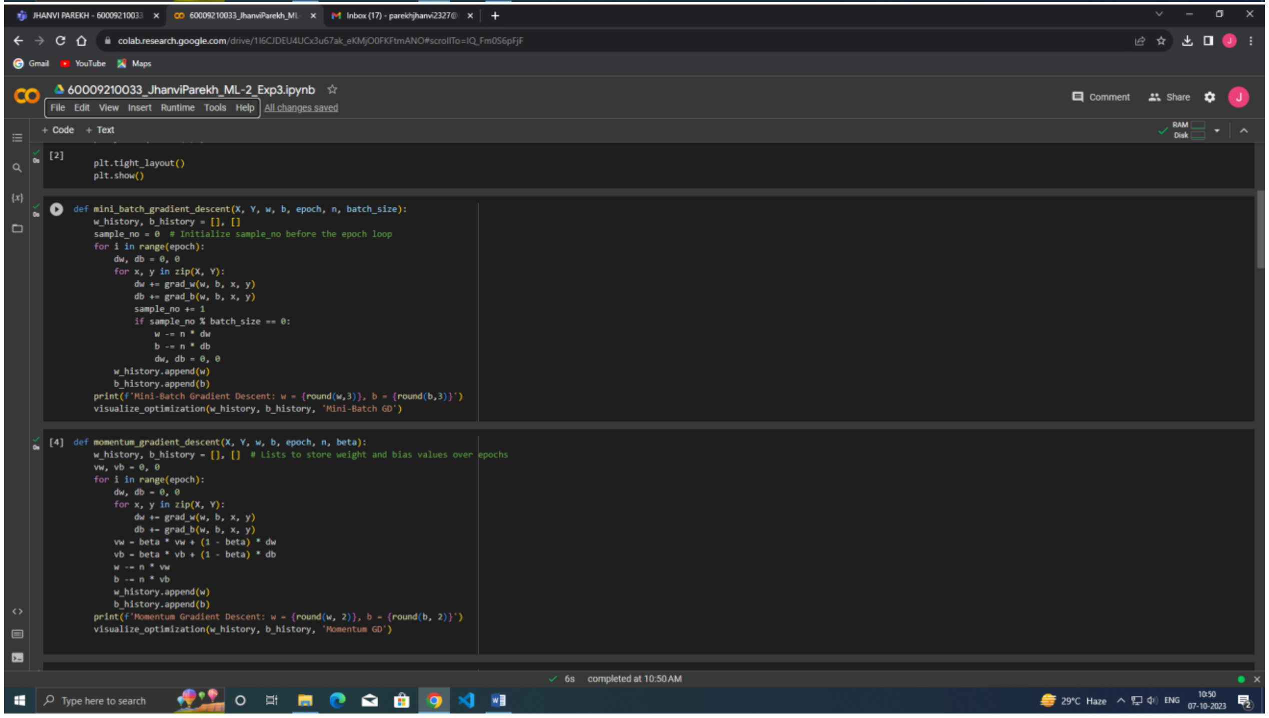

60009210033_JhanviParekh_ML-2_Exp3.ipynb

```
[1] import numpy as np
    import matplotlib.pyplot as plt
    X = [0.5, 2.5]
    Y = [0.2, 0.9]

    def grad_w(w, b, x, y):
        return 2 * (w * x + b - y) * x

    def grad_b(w, b, x, y):
        return 2 * (w * x + b - y)
```

```
def visualize_optimization(w_history, b_history, title):
    plt.figure(figsize=(10, 6))
    plt.subplot(2, 1, 1)
    plt.plot(w_history)
    plt.title(f'Weight (w) Changes Over Epochs ({title})')
    plt.xlabel('Epoch')
    plt.ylabel('Weight (w)')

    plt.subplot(2, 1, 2)
    plt.plot(b_history)
    plt.title(f'Bias (b) Changes Over Epochs ({title})')
    plt.xlabel('Epoch')
    plt.ylabel('Bias (b)')

    plt.tight_layout()
    plt.show()
```

```
[3] def mini_batch_gradient_descent(X, Y, w, b, epoch, n, batch_size):
        w_history, b_history = [], []
        sample_no = 0  # Initialize sample_no before the epoch loop
        for i in range(epoch):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += grad_w(w, b, x, y)
                db += grad_b(w, b, x, y)
                sample_no += 1
                if sample_no % batch_size == 0:
                    w -= n * dw
```

6s completed at 10:50 AM

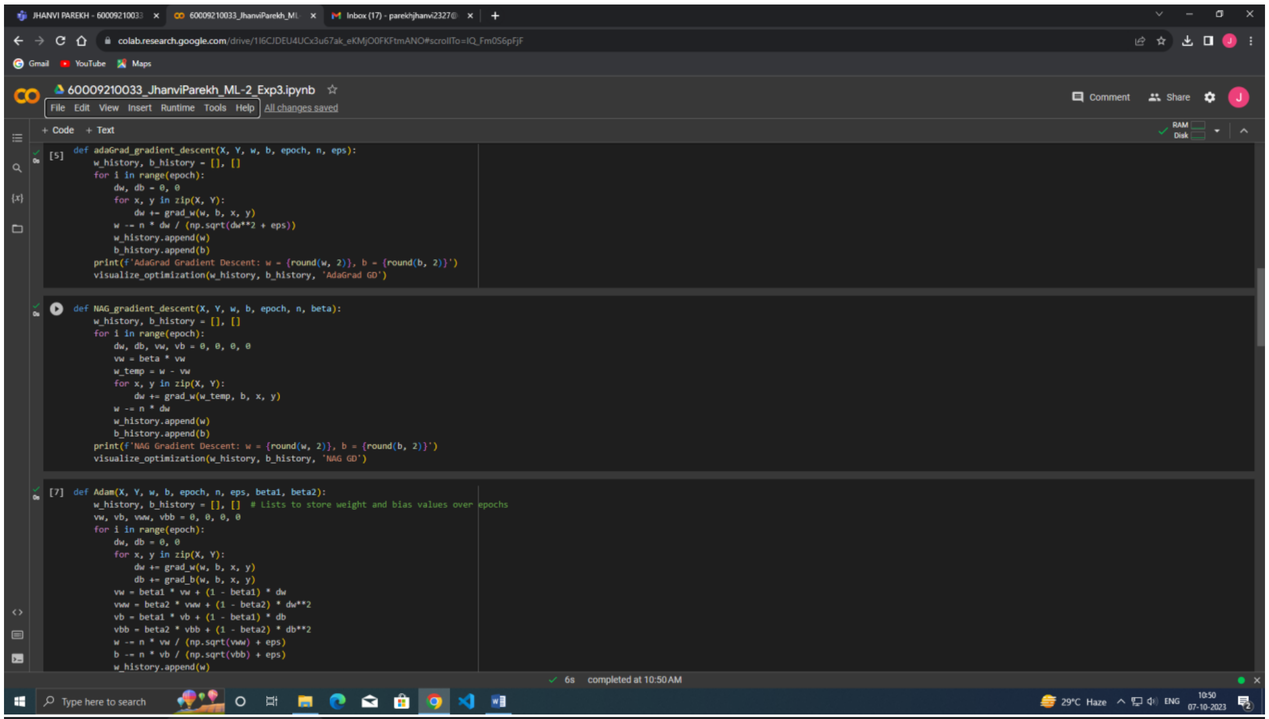

60009210033_JhanviParekh_ML-2_Exp3.ipynb

```
[2]
        plt.tight_layout()
        plt.show()
```

```
def mini_batch_gradient_descent(X, Y, w, b, epoch, n, batch_size):
    w_history, b_history = [], []
    sample_no = 0  # Initialize sample_no before the epoch loop
    for i in range(epoch):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            sample_no += 1
            if sample_no % batch_size == 0:
                w -= n * dw
                b -= n * db
                dw, db = 0, 0
        w_history.append(w)
        b_history.append(b)
    print(f'Mini-Batch Gradient Descent: w = {round(w,3)}, b = {round(b,3)}')
    visualize_optimization(w_history, b_history, 'Mini-Batch GD')
```

```
[4] def momentum_gradient_descent(X, Y, w, b, epoch, n, beta):
        w_history, b_history = [], []  # Lists to store weight and bias values over epochs
        vw, vb = 0, 0
        for i in range(epoch):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += grad_w(w, b, x, y)
                db += grad_b(w, b, x, y)
            vw = beta * vw + (1 - beta) * dw
            vb = beta * vb + (1 - beta) * db
            w -= n * vw
            b -= n * vb
            w_history.append(w)
            b_history.append(b)
        print(f'Momentum Gradient Descent: w = {round(w, 2)}, b = {round(b, 2)}')
        visualize_optimization(w_history, b_history, 'Momentum GD')
```

6s completed at 10:50 AM

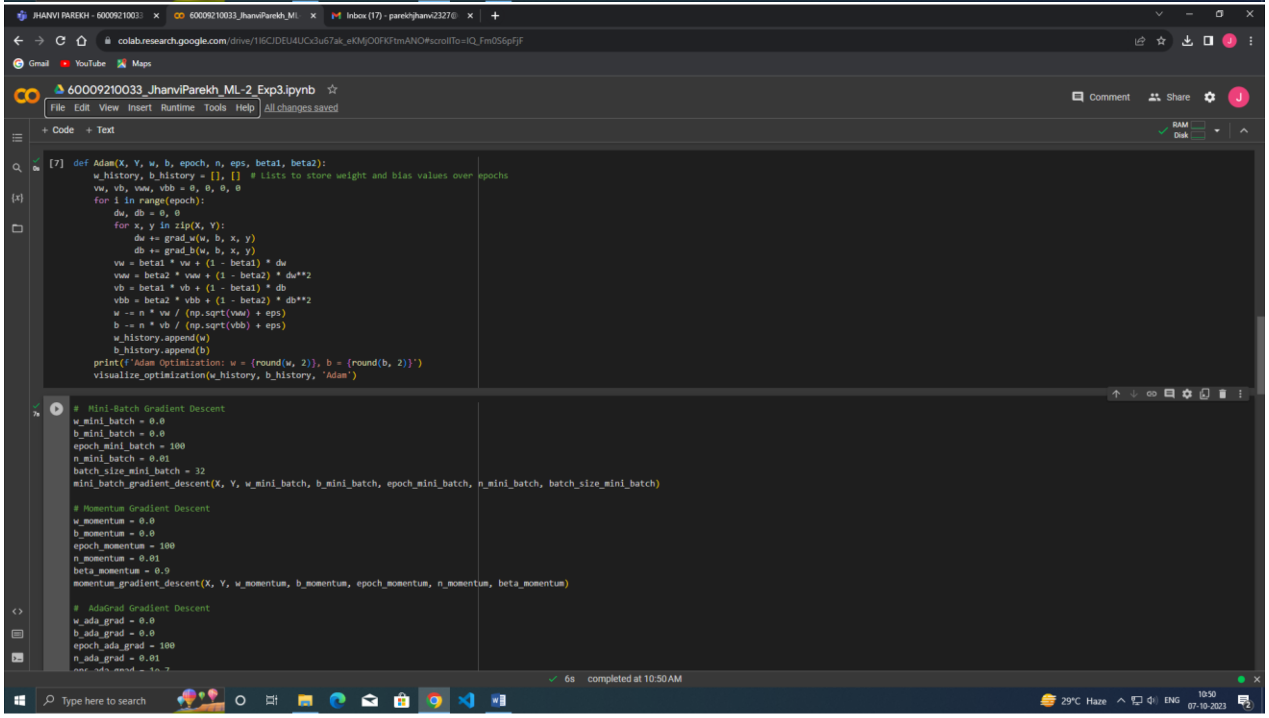


```
[5] def adaGrad_gradient_descent(X, Y, w, b, epoch, n, eps):
        w_history, b_history = [], []
        for i in range(epoch):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += grad_w(w, b, x, y)
            w -= n * dw / (np.sqrt(dw**2 + eps))
            w_history.append(w)
            b_history.append(b)
        print(f'AdaGrad Gradient Descent: w = {round(w, 2)}, b = {round(b, 2)}')
        visualize_optimization(w_history, b_history, 'AdaGrad GD')
```

```
def NAG_gradient_descent(X, Y, w, b, epoch, n, beta):
    w_history, b_history = [], []
    for i in range(epoch):
        dw, db, vw, vb = 0, 0, 0, 0
        vw = beta * vw
        w_temp = w - vw
        for x, y in zip(X, Y):
            dw += grad_w(w_temp, b, x, y)
        w -= n * dw
        w_history.append(w)
        b_history.append(b)
    print(f'NAG Gradient Descent: w = {round(w, 2)}, b = {round(b, 2)}')
    visualize_optimization(w_history, b_history, 'NAG GD')
```

```
[7] def Adam(X, Y, w, b, epoch, n, eps, beta1, beta2):
        w_history, b_history = [], []  # Lists to store weight and bias values over epochs
        vw, vb, vww, vbb = 0, 0, 0, 0
        for i in range(epoch):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += grad_w(w, b, x, y)
                db += grad_b(w, b, x, y)
            vw = beta1 * vw + (1 - beta1) * dw
            vww = beta2 * vww + (1 - beta2) * dw**2
            vb = beta1 * vb + (1 - beta1) * db
            vbb = beta2 * vbb + (1 - beta2) * db**2
            w -= n * vw / (np.sqrt(vww) + eps)
            b -= n * vb / (np.sqrt(vbb) + eps)
            w_history.append(w)
```

6s completed at 10:50 AM

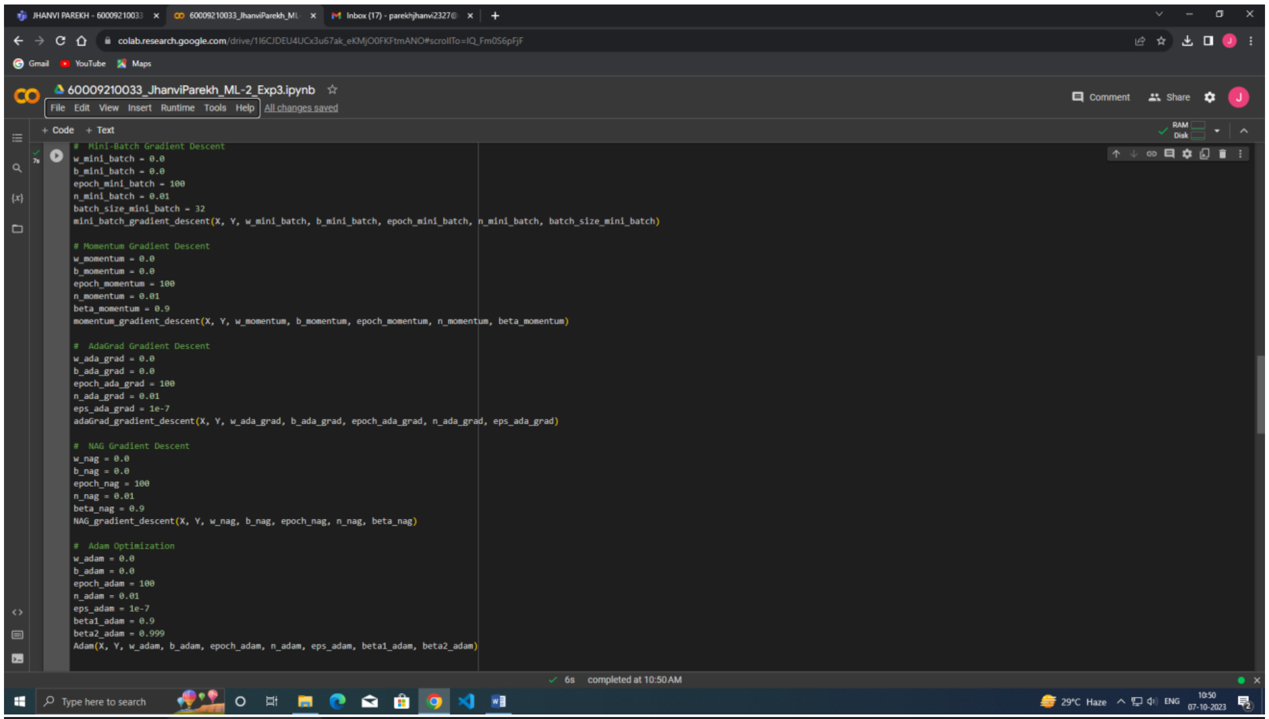


```
[7] def Adam(X, Y, w, b, epoch, n, eps, beta1, beta2):
        w_history, b_history = [], []  # Lists to store weight and bias values over epochs
        vw, vb, vww, vbb = 0, 0, 0, 0
        for i in range(epoch):
            dw, db = 0, 0
            for x, y in zip(X, Y):
                dw += grad_w(w, b, x, y)
                db += grad_b(w, b, x, y)
            vw = beta1 * vw + (1 - beta1) * dw
            vww = beta2 * vww + (1 - beta2) * dw**2
            vb = beta1 * vb + (1 - beta1) * db
            vbb = beta2 * vbb + (1 - beta2) * db**2
            w -= n * vw / (np.sqrt(vww) + eps)
            b -= n * vb / (np.sqrt(vbb) + eps)
            w_history.append(w)
```

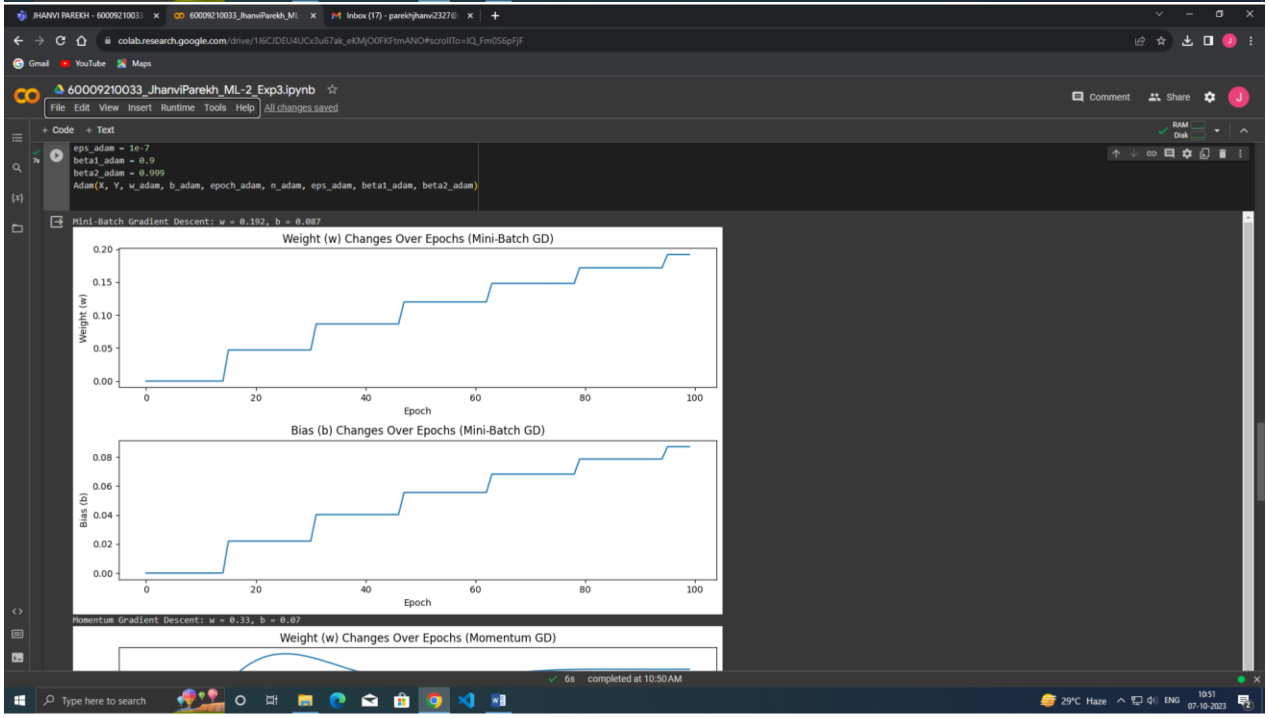
```
            b_history.append(b)
        print(f'Adam Optimization: w = {round(w, 2)}, b = {round(b, 2)}')
        visualize_optimization(w_history, b_history, 'Adam')
```

```
#  Mini-Batch Gradient Descent
w_mini_batch = 0.0
b_mini_batch = 0.0
epoch_mini_batch = 100
n_mini_batch = 0.01
batch_size_mini_batch = 32
mini_batch_gradient_descent(X, Y, w_mini_batch, b_mini_batch, epoch_mini_batch, n_mini_batch, batch_size_mini_batch)

# Momentum Gradient Descent
w_momentum = 0.0
b_momentum = 0.0
epoch_momentum = 100
n_momentum = 0.01
beta_momentum = 0.9
momentum_gradient_descent(X, Y, w_momentum, b_momentum, epoch_momentum, n_momentum, beta_momentum)

#  AdaGrad Gradient Descent
w_ada_grad = 0.0
b_ada_grad = 0.0
epoch_ada_grad = 100
n_ada_grad = 0.01
```

6s completed at 10:50 AM


60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
6s completed at 10:50 AM

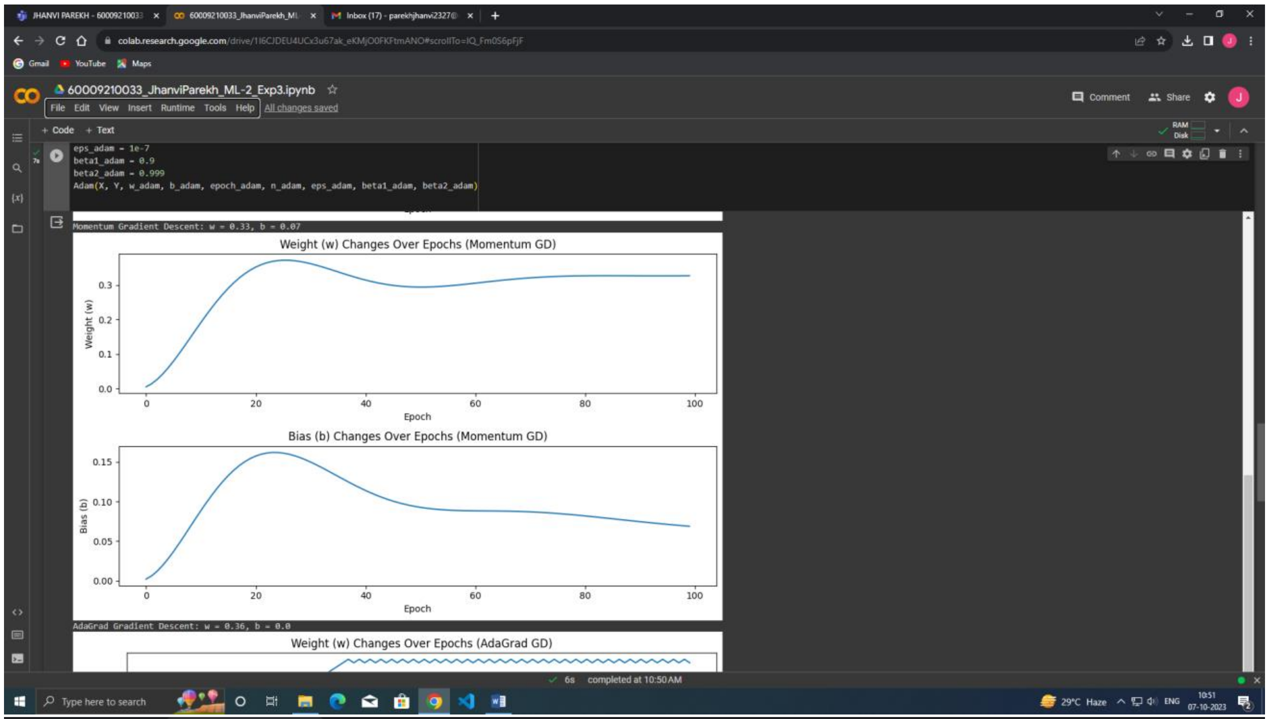
```
#  Mini-Batch Gradient Descent
w_mini_batch = 0.0
b_mini_batch = 0.0
epoch_mini_batch = 100
n_mini_batch = 0.01
batch_size_mini_batch = 32
mini_batch_gradient_descent(X, Y, w_mini_batch, b_mini_batch, epoch_mini_batch, n_mini_batch, batch_size_mini_batch)

# Momentum Gradient Descent
w_momentum = 0.0
b_momentum = 0.0
epoch_momentum = 100
n_momentum = 0.01
beta_momentum = 0.9
momentum_gradient_descent(X, Y, w_momentum, b_momentum, epoch_momentum, n_momentum, beta_momentum)

#  AdaGrad Gradient Descent
w_ada_grad = 0.0
b_ada_grad = 0.0
epoch_ada_grad = 100
n_ada_grad = 0.01
eps_ada_grad = 1e-7
adaGrad_gradient_descent(X, Y, w_ada_grad, b_ada_grad, epoch_ada_grad, n_ada_grad, eps_ada_grad)

#  NAG Gradient Descent
w_nag = 0.0
b_nag = 0.0
epoch_nag = 100
n_nag = 0.01
beta_nag = 0.9
NAG_gradient_descent(X, Y, w_nag, b_nag, epoch_nag, n_nag, beta_nag)

#  Adam Optimization
w_adam = 0.0
b_adam = 0.0
epoch_adam = 100
n_adam = 0.01
eps_adam = 1e-7
beta1_adam = 0.9
beta2_adam = 0.999
Adam(X, Y, w_adam, b_adam, epoch_adam, n_adam, eps_adam, beta1_adam, beta2_adam)
```


60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Weight (w) Changes Over Epochs (Mini-Batch GD)
Weight (w)
Epoch
Bias (b) Changes Over Epochs (Mini-Batch GD)
Bias (b)
Epoch
Weight (w) Changes Over Epochs (Momentum GD)
6s completed at 10:50 AM

```
eps_adam = 1e-7
beta1_adam = 0.9
beta2_adam = 0.999
Adam(X, Y, w_adam, b_adam, epoch_adam, n_adam, eps_adam, beta1_adam, beta2_adam)
```

```
Mini-Batch Gradient Descent: w = 0.192, b = 0.087
```

```
Momentum Gradient Descent: w = 0.33, b = 0.07
```

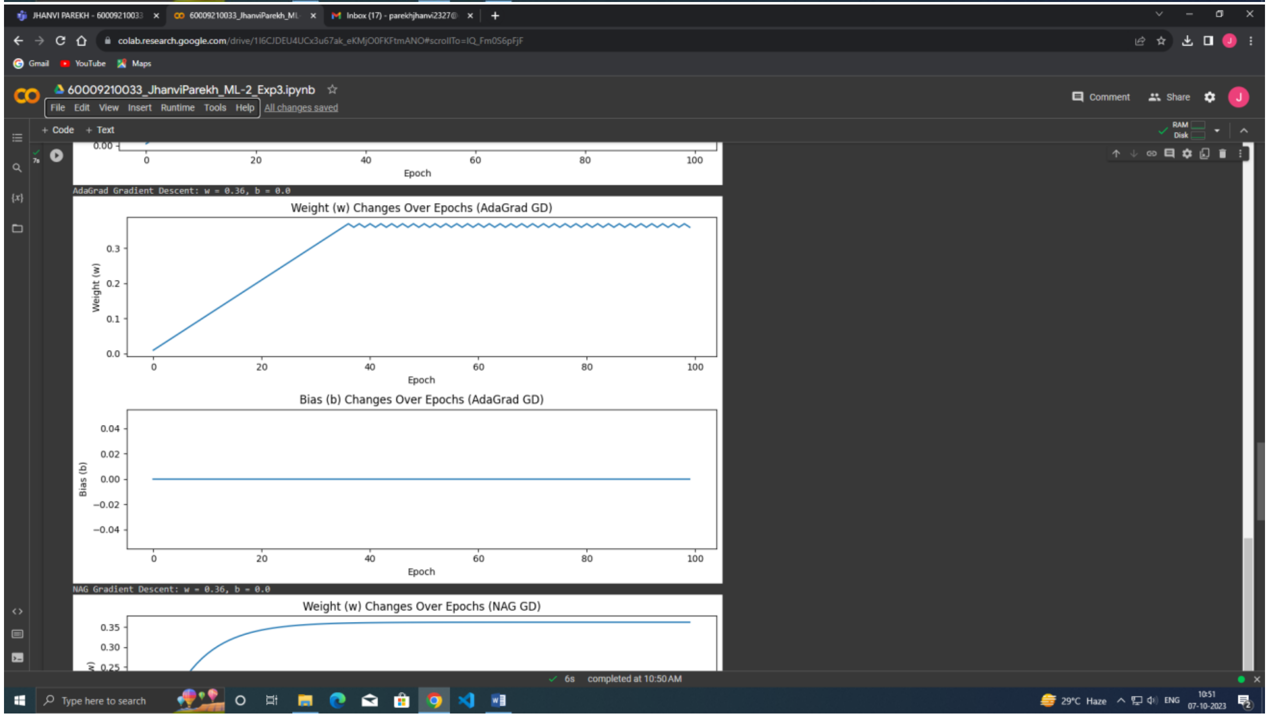


```
eps_adam = 1e-7
beta1_adam = 0.9
beta2_adam = 0.999
Adam(X, Y, w_adam, b_adam, epoch_adam, n_adam, eps_adam, beta1_adam, beta2_adam)
```

```
Momentum Gradient Descent: w = 0.33, b = 0.07
```

60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Weight (w) Changes Over Epochs (Momentum GD)
Weight (w)
Epoch
Bias (b) Changes Over Epochs (Momentum GD)
Bias (b)
Epoch
Weight (w) Changes Over Epochs (AdaGrad GD)
completed at 10:50 AM

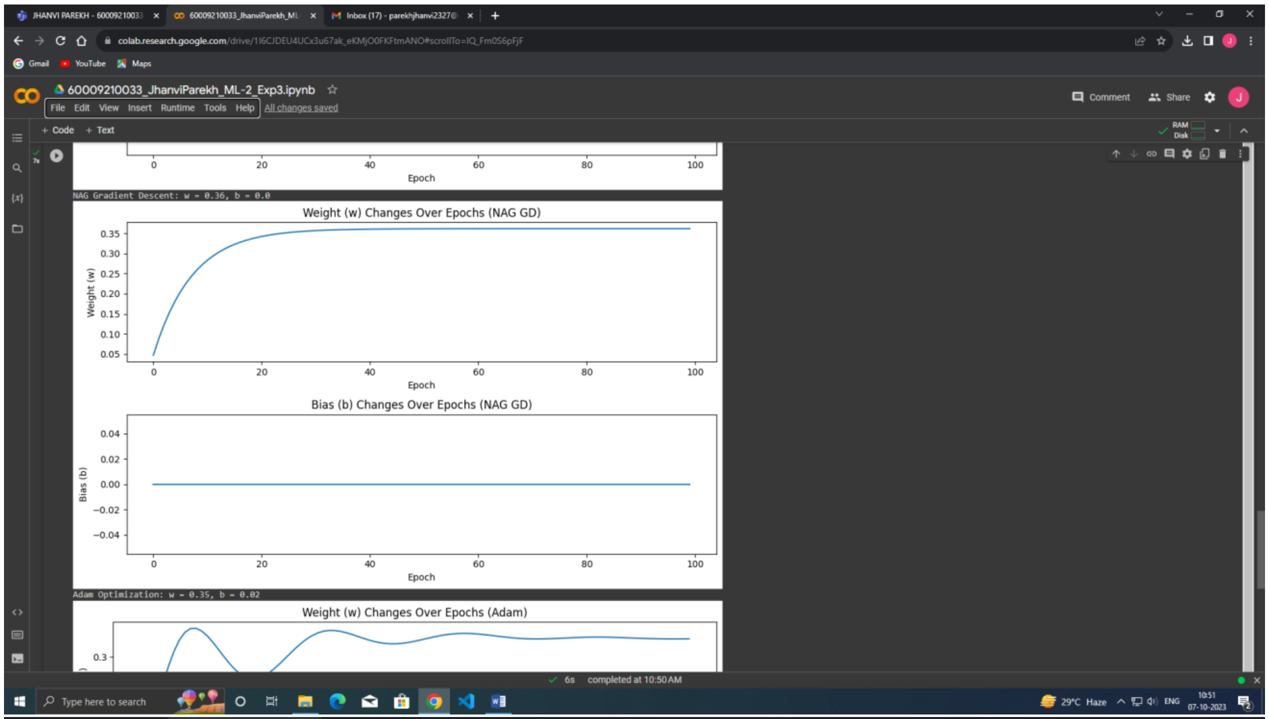
```
AdaGrad Gradient Descent: w = 0.36, b = 0.0
```



```
AdaGrad Gradient Descent: w = 0.36, b = 0.0
```

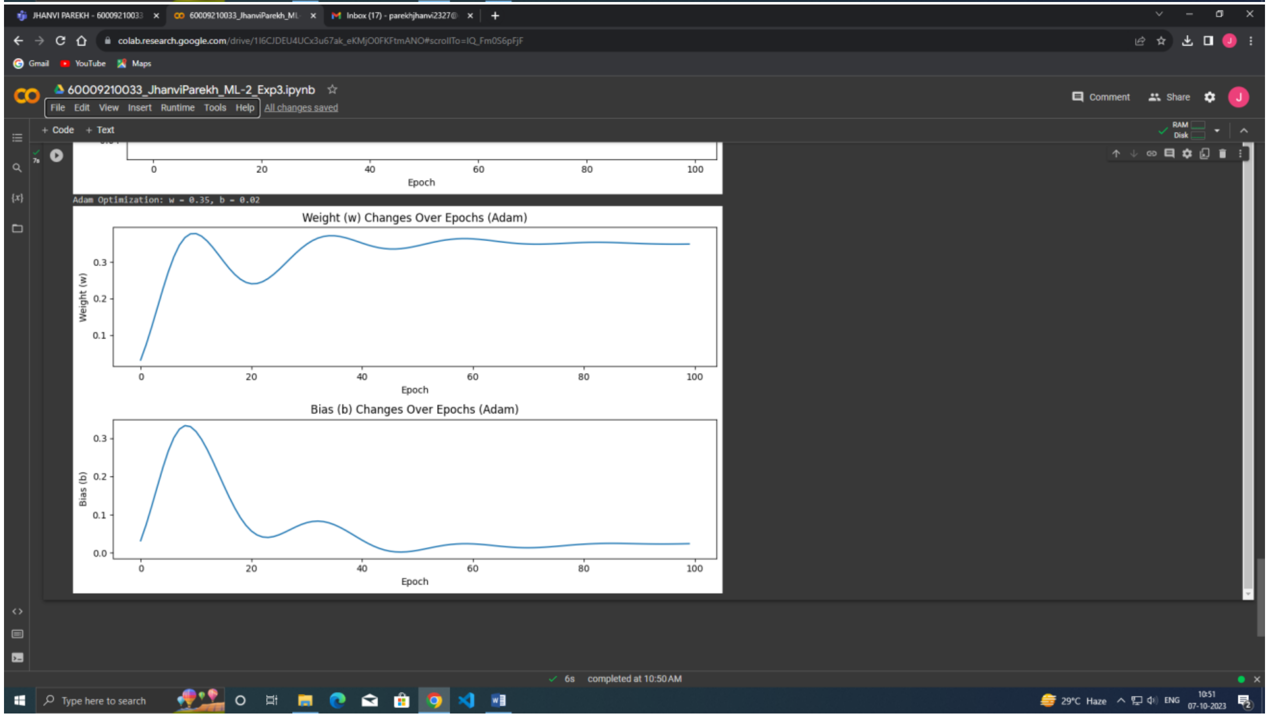
60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Weight (w) Changes Over Epochs (AdaGrad GD)
Weight (w)
Epoch
Bias (b) Changes Over Epochs (AdaGrad GD)
Bias (b)
Epoch
Weight (w) Changes Over Epochs (NAG GD)
completed at 10:50 AM

```
NAG Gradient Descent: w = 0.36, b = 0.0
```


60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
Epoch
Weight (w) Changes Over Epochs (NAG GD)
Weight (w)
Epoch
Bias (b) Changes Over Epochs (NAG GD)
Bias (b)
Epoch
Weight (w) Changes Over Epochs (Adam)
6s completed at 10:50 AM

```
NAG Gradient Descent: w = 0.36, b = 0.0
```

```
Adam Optimization: w = 0.35, b = 0.02
```


60009210033_JhanviParekh_ML-2_Exp3.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment Share
+ Code + Text
Epoch
Weight (w) Changes Over Epochs (Adam)
Weight (w)
Epoch
Bias (b) Changes Over Epochs (Adam)
Bias (b)
Epoch
6s completed at 10:50 AM

```
Adam Optimization: w = 0.35, b = 0.02
```