Comparative Analysis of Optimization Algorithms in Deep Learning

Tong Huang

March 8, 2024

1 Introduction

Stochastic Gradient Descent (SGD) has long been the cornerstone of neural network optimization, due to its simplicity and effectiveness. However, the inherent challenges of training deep neural networks, such as slow convergence, susceptibility to local minima, and the difficulty of choosing an appropriate learning rate, have driven the development of advanced optimization techniques. Among these, SGD with Momentum, RMSprop, and Adam stand out as significant enhancements over plain SGD, each introducing unique mechanisms to address the challenges of deep learning optimization.

This project aims to conduct a comprehensive comparative analysis of these four optimization algorithms: SGD, SGD with Momentum, RMSprop, and Adam. Our comparison is grounded on several key metrics that capture aspects of optimization relevant to practitioners and researchers alike, including convergence speed, robustness to hyperparameter settings, and final model accuracy.

2 Objective

This project evaluates the performance of these algorithms on a churn dataset, to understand their behavior and to analyze their efficiency in terms of convergence speed.

3 Methodology

3.1 Data

This project uses a churn data set on E commerce retail. There are 48 columns including different characteristics of users, and 49358 rows which represent the number of users in the dataset.

3.2 Model Architecture

This project employs a simple neural net structure from scratch, for me to both practice implementing deep learning model and learn how to write the optimization algorithms from scratch. The simple neural net includes one hidden layer with 10 neurons by using ReLu activation function, one output layer with Sigmoid function, and a binary cross-entropy loss function.

3.3 Optimization Methods

1. Gradient Descent
2. SGD(Stochastic Gradient Descent)
3. SGD with Momentum
4. SGD with RMSProp
5. SGD with Adam

3.4 Evaluating Metrics

1. Training Time: Time taken to converge or reach a specific accuracy/loss threshold
2. Convergence Speed: Number of epochs or iterations needed to converge to a threshold.
3. Final Performance: Accuracy, F1 score, precision, recall, or other relevant performance metrics on both the training and validation/test sets.
4. Stability: Variation in performance metrics across epochs (less variation is usually better).

3.5 Deep Learning Structure

This section shows the algorithms that forward and backward propagation follow.

3.5.1 Forward Propagation

$$\begin{aligned} Z_1 &= W_1 A_0 + b_1 \\ A_1 &= g(Z_1) \\ Z_2 &= W_2 A_1 + b_2 \\ A_2 &= \textbf{Sigmoid}(Z_2) \end{aligned}$$

3.5.2 Backward Propagation

$$\begin{aligned} dZ_2 &= A_2 - Y \\ dW_2 &= \frac{1}{m} dZ_2 \cdot A1^T \\ db_2 &= \frac{1}{m} \sum dZ_2 \\ dA_1 &= W_2^T \cdot dZ_2 \\ dZ_1 &= dA_1 \odot (Z_1 > 0) \\ dW_1 &= \frac{1}{m} dZ_1 \cdot X^T \\ db_1 &= \frac{1}{m} \sum dZ_1 \end{aligned}$$

1. A_0 is the initial train data, X
2. W_1 is the initial weight for first layer
3. b_1 is the initial bias for first layer
4. Z_1 is the first layer linear combination
5. A_1 is the first layer output
6. W_2 is the initial weight for second layer
7. b_2 is the initial weight for second layer
8. Z_2 is the second layer linear combination
9. A_2 is the output after apply sigmoid function to Z_2

4 Optimization Algorithms

This section shows that steps that each optimization algorithm follow to execute the backward propagation.

4.1 Stochastic Gradient Descent

In the training phase, the dataset is first randomized to ensure that the order of the data points does not introduce bias into the learning algorithm. Following this initial step, the model's weights and biases are iteratively updated using a single data point at each iteration. This process, which constitutes one pass through the entire dataset, is referred to as an epoch. Upon completion of an epoch, the dataset is shuffled anew, and the iterative update process is repeated for the subsequent epoch.

4.1.1 Update Algorithm

$$\begin{aligned} W_1 &= W_1 - \alpha dW_1 \\ b_1 &= b_1 - \alpha db_1 \\ W_2 &= W_2 - \alpha dW_2 \\ b_2 &= b_2 - \alpha b_2 \end{aligned}$$

This update only using one data point after randomizing the entire data set.

4.2 Stochastic Gradient Descent with Momentum

Stochastic Gradient Descent (SGD) with Momentum is an advanced optimization technique that builds upon the foundational principles of traditional SGD. While adhering to the core SGD protocol of iteratively updating model parameters using individual data points, SGD with Momentum introduces an additional mechanism to enhance the convergence process. This is achieved through the initialization of velocity parameters for each model parameter, enabling the algorithm to retain a memory of previous gradients. The degree of momentum is controlled by a hyperparameter, denoted as β , which determines the weighting of the accumulated gradient history in the parameter update equation.

4.2.1 Update Algorithm

$$\begin{aligned} v_{W1} &= \beta \cdot v_{W1} + (1-\beta) \cdot dW1, \\ v_{b1} &= \beta \cdot v_{b1} + (1-\beta) \cdot db1, \\ v_{W2} &= \beta \cdot v_{W2} + (1-\beta) \cdot dW2, \\ v_{b2} &= \beta \cdot v_{b2} + (1-\beta) \cdot db2, \end{aligned}$$

$$\begin{aligned} W1 &= W1 - \alpha \cdot v_{W1}, \\ b1 &= b1 - \alpha \cdot v_{b1}, \\ W2 &= W2 - \alpha \cdot v_{W2}, \\ b2 &= b2 - \alpha \cdot v_{b2}, \end{aligned}$$

1. The terms v_{W_i} and v_{b_i} are the velocities for the weights and biases of layers $i = 1, 2$
2. The terms dW_i and db_i represent the gradients of the loss function with respect to the weights and biases of layer i
3. The parameter β is the momentum coefficient
4. The parameter α is the learning rate

4.3 Stochastic Gradient Descent with RMSProp

RMSprop operates on the principle of modulating the learning rate of each weight in the model based on the recent magnitudes of its gradients. This is achieved by maintaining a moving average of the squared gradients, which serves to normalize the gradient step.

4.3.1 Update Algorithm

$$\begin{aligned}
s_{W_1} &= \beta \cdot s_{W_1} + (1-\beta) \cdot (dW_1)^2 \\
s_{b_1} &= \beta \cdot s_{b_1} + (1-\beta) \cdot (db_1)^2 \\
s_{W_2} &= \beta \cdot s_{W_2} + (1-\beta) \cdot (dW_2)^2 \\
s_{b_2} &= \beta \cdot s_{b_2} + (1-\beta) \cdot (db_2)^2
\end{aligned}$$

$$\begin{aligned}
W_1 &= W_1 - \frac{\alpha \cdot dW_1}{\sqrt{s_{W_1}} + \epsilon} \\
b_1 &= b_1 - \frac{\alpha \cdot db_1}{\sqrt{s_{b_1}} + \epsilon} \\
W_2 &= W_2 - \frac{\alpha \cdot dW_2}{\sqrt{s_{W_2}} + \epsilon} \\
b_2 &= b_2 - \frac{\alpha \cdot db_2}{\sqrt{s_{b_2}} + \epsilon}
\end{aligned}$$

1. The terms s_{W_1}, s_{b_1}, s_{W_2}, and s_{b_2} represent the moving averages of the squared gradients for the respective weights and biases.
2. The parameter α is the learning rate
3. The parameter β is the decay rate that influences the moving average, blending the influence of past squared gradients and the current squared gradient.
4. ϵ is a small constant added for numerical stability to prevent division by zero.

4.4 Stochastic Gradient Descent with ADAM

The Adam optimization algorithm enhances the stochastic gradient descent process by incorporating momentum and scaling the gradient to achieve an adaptive learning rate for each parameter. This is accomplished through the calculation of the first and second moments of the gradients.

4.4.1 Update Algorithm

$$\begin{aligned}
&\text{For each layer } i \in \{1, 2\} : \\
&m_{W_i} = \beta_1 \cdot m_{W_i} + (1-\beta_1) \cdot dW_i \\
&mb_i = \beta_1 \cdot mb_i + (1-\beta_1) \cdot db_i \\
&v_{W_i} = \beta_2 \cdot v_{W_i} + (1-\beta_2) \cdot (dW_i)^2 \\
&v_{b_i} = \beta_2 \cdot v_{b_i} + (1-\beta_2) \cdot (db_i)^2
\end{aligned}$$

$$\begin{aligned}
m_{W_i}^{corrected} &= \frac{m_{W_i}}{1-\beta_1^t}, \quad & m_{b_i}^{corrected} &= \frac{m_{b_i}}{1-\beta_1^t} \\
v_{W_i}^{corrected} &= \frac{v_{W_i}}{1-\beta_2^t}, \quad & v_{b_i}^{corrected} &= \frac{v_{b_i}}{1-\beta_2^t}
\end{aligned}$$

$$W_i = W_i - \frac{\alpha \cdot m_{W_i}^{corrected}}{\sqrt{v_{W_i}^{corrected}} + \epsilon}$$

$$b_i = b_i - \frac{\alpha \cdot m_{b_i}^{corrected}}{\sqrt{v_{b_i}^{corrected}} + \epsilon}$$

- m_{W_i} and m_{b_i}: estimates of the first moments (the mean of the gradients) for weights and biases, respectively.
- v_{W_i} and v_{b_i}: estimates of the second moments (the uncentered variance of the gradients) for weights and biases, respectively.
- β_1 and β_2: hyperparameters that control the decay rates of these moving averages, influencing how much of the past gradients are retained.
- α: the learning rate, determining the step size in the parameter update process.
- ϵ: a small constant added for numerical stability, preventing division by zero in the update equations.
- t: the timestep, representing the current iteration, used in bias correction to adjust the first and second moment estimates.

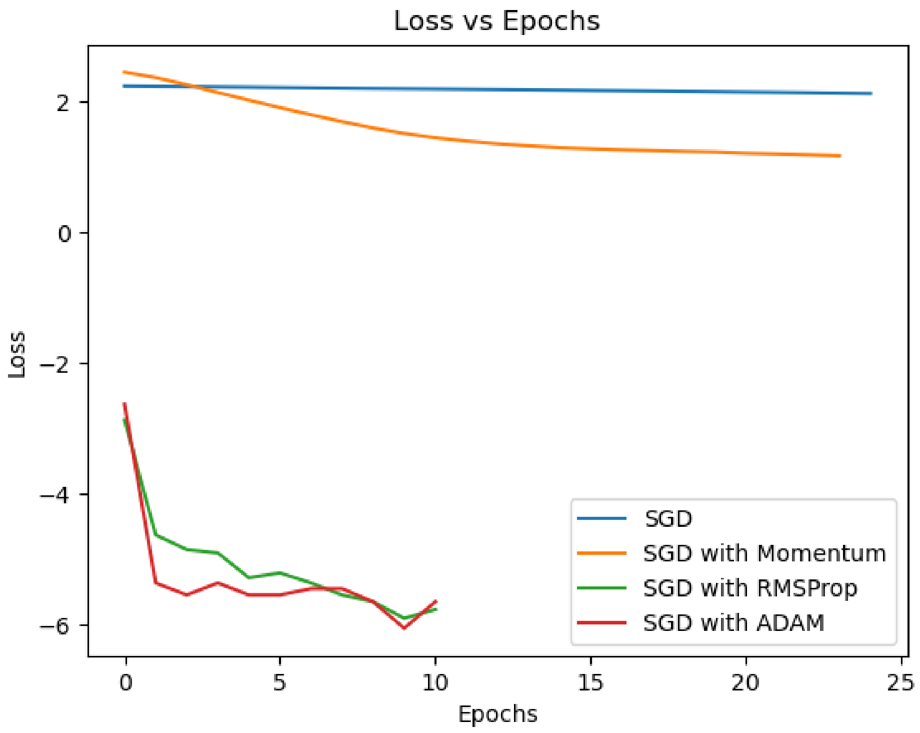
5 Performance

To standardize the parameters, we choose learning rate to be 0.01 for all models, and choose 0.9 for momentum coefficient and decay rate.

5.1 Training Time and Convergence Speed

For training time, we set up a threshold to be the rate of accuracy of 0.90 for algorithms to reach.

Algorithms	Time to reach threshold(0.90 Accuracy)	Epoch to reach threshold (0.90 Accuracy)
SGD	1931 seconds (32 minutes)	247 epochs
SGD with Momentum	66 seconds (1.1 minutes)	24 epochs
SGD with RMSProp	3.58 seconds	1 epoch
SGD with ADAM	9.53 seconds	1 epoch



The loss versus epochs plot shows the convergence efficiency of RMSProp and ADAM over basic SGD and SGD with Momentum. It is worth noting that while ADAM converges rapidly, it does not necessarily lead to the best generalization performance as indicated by the final ROC score. This underlines the importance of evaluating optimization algorithms not solely on their speed but also on their ability to generalize well to unseen data.

5.2 Final Performance

For this section, we will show the out-of-sample testing performance score for each algorithm. For SGD and SGD with momentum, we will pick the number of epochs to be the epoch that they reach 0.9 accuracy threshold. For SGD with RMSProp and ADAM, since they reach 0.99 accuracy after one epoch, we will pick arbitrarily the number of epochs to be 11 for more iterations.

5.2.1 SGD

	Precision	Recall	F1 score	Support
0	0.09	0.56	0.15	1095
1	0.83	0.28	0.41	8776
Accuracy			0.31	
ROC score			0.43	

5.2.2 SGD with momentum

	Precision	Recall	F1 score	Support
0	0.21	0.17	0.19	1095
1	0.90	0.92	0.91	8776
Accuracy			0.84	
ROC score			0.56	

5.2.3 SGD with RMSProp

	Precision	Recall	F1 score	Support
0	0.40	0.28	0.33	1095
1	0.91	0.95	0.93	8776
Accuracy			0.87	
ROC score			0.69	

5.2.4 SGD with ADAM

	Precision	Recall	F1 score	Support
0	0.35	0.68	0.47	1095
1	0.95	0.84	0.93	8776
Accuracy			0.82	
ROC score			0.80	

6 Conclusion

The empirical evaluation of various optimization algorithms on a classification task reveals notable differences in their performance and efficiency. SGD, while foundational, lags significantly in terms of training time and requires considerably more epochs to achieve a comparable level of accuracy to other methods. The addition of momentum to SGD drastically reduces both the time to convergence and the number of epochs required, illustrating the efficacy of incorporating velocity components into the optimization process.

RMSProp outperforms both SGD and SGD with Momentum in terms of accuracy, reaching a near-optimal accuracy in just one epoch. This demonstrates the strength of RMSProp convergence behavior, particularly in settings where the optimization landscape is complex.

SGD with ADAM, however, emerges as a superior optimization strategy in this comparison. It achieves the highest accuracy and ROC score while also converging in the least amount of time. The balanced combination of momentum and adaptive learning rates in ADAM accounts for its robust performance across various metrics, including precision, recall, and F1 score. ADAM's ability to quickly navigate the optimization landscape and find a performance model with minimal iterations underscores its utility in training deep learning models.

In conclusion, the selection of an optimization algorithm should be guided by the specific demands of the task at hand, balancing the need for speed and high convergence with the ultimate goal of achieving a model that generalizes well. The empirical evidence suggests that while SGD with Momentum provides a significant improvement over plain SGD, ADAM and RMSProp offer compelling advantages for faster training and potentially better generalization, making them preferable choices for tasks similar to the one studied here.

7 Appendix(Code)

7.1 Forward Propogation

```
def init_param():
    W1 = np.random.randn(10, 48)
    b1 = np.random.randn(10, 1)
    W2 = np.random.randn(1, 10)
    b2 = np.random.randn(1, 1)

    return W1, b1, W2, b2

def ReLu(Z):
    return np.maximum(0, Z)

def sigmoid(Z):

    return 1/(1 + np.exp(-Z))

def forward_prop(W1, b1, W2, b2, X):
    Z1 = W1.dot(X) + b1
    A1 = ReLu(Z1)
    Z2 = W2.dot(A1) + b2
    A2 = sigmoid(Z2)
    return Z1, A1, Z2, A2
```

7.2 Backward Propagation

```
    def backward_prop(Z1, A1, Z2, A2, W2, X, Y):


    # Calculate gradient of the loss with respect to A2
    dZ2 = A2 - Y

    # Calculate gradient of the loss with respect to W2 and b2
    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)

    # Backpropagate to hidden layer
    dA1 = np.dot(W2.T, dZ2)
    # Calculate gradient of the loss with respect to Z1
    dZ1 = dA1 * (Z1 > 0)  # Since the derivative of ReLU is 1 if Z1 > 0 else 0

    # Calculate gradient of the loss with respect to W1 and b1
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    return dW1, db1, dW2, db2
```

7.3 Functions

```
def update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha):
    W1 = W1 - alpha * dW1
    b1 = b1 - alpha * db1
    W2 = W2 - alpha * dW2
    b2 = b2 - alpha * db2
    return W1, b1, W2, b2

def get_prediction(A2):
    return (A2 > 0.5).astype(int)

def get_accuracy(predictions, Y):
    print(predictions, Y)
    return np.mean(predictions == Y)
def predict(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    predictions = get_prediction(A2)
    return predictions

def predict_score(X, W1, b1, W2, b2):
    _, _, _, A2 = forward_prop(W1, b1, W2, b2, X)
    return A2

def binary_cross_entropy(y_true, y_pred):
    y_pred = np.clip(y_pred, 1e-7, 1- 1e-7)
    left = (1 - y_true) * np.log(1 - y_pred + 1e-7)
    right = y_true * np.log(y_pred + 1e-7)
    return -np.mean(left + right, axis = 0)
```

7.4 SGD

```
    def stochastic_gradient_descent(X, y, epochs, alpha):
    W1, b1, W2, b2 = init_param()

    m = X.shape[1]
    loss_v = np.zeros(epochs)
    for epoch in range(epochs):
        permutation = np.random.permutation(m)
        X_shuffle = X[:, permutation]
        y_shuffle = y[permutation]

        predictions = []

        #alpha = learningRateDecay(alpha, 0.001, epoch)

        for i in range(m):
            X_i = X_shuffle[:, i:i+1]
            y_i = y_shuffle[i:i+1]

            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X_i)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W2, X_i, y_i)
            W1, b1, W2, b2 = update_params(W1, b1, W2, b2, dW1, db1, dW2, db2, alpha)

            predictions.append(get_prediction(A2))

        epoch_predictions = np.concatenate(predictions, axis=1)
        epoch_accuracy = get_accuracy(epoch_predictions, y_shuffle)
        loss = binary_cross_entropy(y_shuffle.reshape(-1,1), epoch_predictions.reshape(-
        loss_v[epoch] = loss
        print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {epoch_accuracy} - loss: {loss}")
        if epoch_accuracy >= 0.90:
            return W1, b1, W2, b2, loss_v

    return W1, b1, W2, b2, loss_v

s = X_train_scaled.shape[1]
p = np.random.permutation(s)
y_train[p]
start_time = time.time()
W1, b1, W2, b2, loss_sgd = stochastic_gradient_descent(X_train_scaled, y_train, 25, 0.01
end_time = time.time()
sgd_time = start_time - end_time
```

7.5 SGD with Momentum

```
def init_v(W1, b1, W2, b2):
    vW1 = np.zeros_like(W1)
    vb1 = np.zeros_like(b1)
    vW2 = np.zeros_like(W2)
    vb2 = np.zeros_like(b2)
    return vW1, vb1, vW2, vb2

def update_params_with_momentum(W1, b1, W2, b2, dW1, db1, dW2, db2, vW1, vb1, vW2, vb2,
    # Update velocities
    vW1 = beta * vW1 + (1 - beta) * dW1
    vb1 = beta * vb1 + (1 - beta) * db1
    vW2 = beta * vW2 + (1 - beta) * dW2
    vb2 = beta * vb2 + (1 - beta) * db2

    # Update parameters
    W1 = W1 - alpha * vW1
    b1 = b1 - alpha * vb1
    W2 = W2 - alpha * vW2
    b2 = b2 - alpha * vb2

    return W1, b1, W2, b2, vW1, vb1, vW2, vb2


def sgd_with_momentum(X, y, epochs, alpha, beta):
    W1, b1, W2, b2 = init_param()
    vW1, vb1, vW2, vb2 = init_v(W1, b1, W2, b2)

    m = X.shape[1]
    loss_v = np.zeros(epochs)
    for epoch in range(epochs):
        permutation = np.random.permutation(m)
        X_shuffle = X[:, permutation]
        y_shuffle = y[permutation]

        predictions = []

        #alpha = learningRateDecay(alpha, 0.01, epoch)

        for i in range(m):
            X_i = X_shuffle[:, i:i+1]
            y_i = y_shuffle[i:i+1]
            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X_i)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W2, X_i, y_i)
            W1, b1, W2, b2, vW1, vb1, vW2, vb2 = update_params_with_momentum(W1, b1, W2,
            dW2, db2, vW1, vb1, vW2, vb2, alpha, beta)

            predictions.append(get_prediction(A2))

        epoch_predictions = np.concatenate(predictions, axis=1)
        epoch_accuracy = get_accuracy(epoch_predictions, y_shuffle)
        print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {epoch_accuracy}")
        loss = binary_cross_entropy(y_shuffle.reshape(-1,1), epoch_predictions.reshape(-
        loss_v[epoch] = loss
        if epoch_accuracy >= 0.90:
            return W1, b1, W2, b2, loss_v

    return W1, b1, W2, b2, loss_v
```

7.6 SGD with RMSProp

```
def init_rmsp(W1, b1, W2, b2):
    sW1 = np.zeros_like(W1)
    sb1 = np.zeros_like(b1)
    sW2 = np.zeros_like(W2)
    sb2 = np.zeros_like(b2)
    return sW1, sb1, sW2, sb2

def update_params_with_rmsprop(W1, b1, W2, b2, dW1, db1, dW2, db2, sW1, sb1, sW2, sb2, a
    # Update squared gradient accumulators
    sW1 = beta * sW1 + (1 - beta) * np.square(dW1)
    sb1 = beta * sb1 + (1 - beta) * np.square(db1)
    sW2 = beta * sW2 + (1 - beta) * np.square(dW2)
    sb2 = beta * sb2 + (1 - beta) * np.square(db2)

    # Update parameters
    W1 -= alpha * dW1 / (np.sqrt(sW1) + epsilon)
    b1 -= alpha * db1 / (np.sqrt(sb1) + epsilon)
    W2 -= alpha * dW2 / (np.sqrt(sW2) + epsilon)
    b2 -= alpha * db2 / (np.sqrt(sb2) + epsilon)

    return W1, b1, W2, b2, sW1, sb1, sW2, sb2

def sgd_with_RMSProp(X, y, epochs, alpha, beta):
    W1, b1, W2, b2 = init_param()
    sW1, sb1, sW2, sb2 = init_rmsp(W1, b1, W2, b2)

    m = X.shape[1]
    loss_v = np.zeros(epochs)
    for epoch in range(epochs):
        permutation = np.random.permutation(m)
        X_shuffle = X[:, permutation]
        y_shuffle = y[permutation]

        predictions = []

        #alpha = learningRateDecay(alpha, 0.01, epoch)
        for i in range(m):
            X_i = X_shuffle[:, i:i+1]
            y_i = y_shuffle[i:i+1]

            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X_i)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W2, X_i, y_i)
            W1, b1, W2, b2, vW1, vb1, vW2, vb2 = update_params_with_rmsprop(W1, b1, W2,
            dW2, db2, sW1, sb1, sW2, sb2, alpha, beta)

            predictions.append(get_prediction(A2))

        epoch_predictions = np.concatenate(predictions, axis=1)
        epoch_accuracy = get_accuracy(epoch_predictions, y_shuffle)

        loss = binary_cross_entropy(y_shuffle.reshape(-1,1), epoch_predictions.reshape(-
        loss_v[epoch] = loss
        print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {epoch_accuracy} - loss:{loss}")
    return W1, b1, W2, b2, loss_v
```

7.7 SGD with ADAM

```
def init_adam(W1, b1, W2, b2):
    # First moment vectors
    mW1 = np.zeros_like(W1)
    mb1 = np.zeros_like(b1)
    mW2 = np.zeros_like(W2)
    mb2 = np.zeros_like(b2)

    # Second moment vectors
    vW1 = np.zeros_like(W1)
    vb1 = np.zeros_like(b1)
    vW2 = np.zeros_like(W2)
    vb2 = np.zeros_like(b2)

    return mW1, mb1, mW2, mb2, vW1, vb1, vW2, vb2

def update_with_ADAM(W1, b1, W2, b2, dW1, db1, dW2, db2, mW1, mb1, mW2, mb2,
                     vW1, vb1, vW2, vb2, t, alpha, beta1, beta2, epsilon=1e-8):

    mW1 = beta1 * mW1 + (1 - beta1) * dW1
    mb1 = beta1 * mb1 + (1 - beta1) * db1
    mW2 = beta1 * mW2 + (1 - beta1) * dW2
    mb2 = beta1 * mb2 + (1 - beta1) * db2

    # Correcting the square of gradients
    vW1 = beta2 * vW1 + (1 - beta2) * (dW1 ** 2)
    vb1 = beta2 * vb1 + (1 - beta2) * (db1 ** 2)
    vW2 = beta2 * vW2 + (1 - beta2) * (dW2 ** 2)
    vb2 = beta2 * vb2 + (1 - beta2) * (db2 ** 2)

    # First moment correction
    mW1_corrected = mW1 / (1 - beta1 ** t)
    mb1_corrected = mb1 / (1 - beta1 ** t)
    mW2_corrected = mW2 / (1 - beta1 ** t)
    mb2_corrected = mb2 / (1 - beta1 ** t)
    # Second moment correction
    vW1_corrected = vW1 / (1 - beta2 ** t)
    vb1_corrected = vb1 / (1 - beta2 ** t)
    vW2_corrected = vW2 / (1 - beta2 ** t)
    vb2_corrected = vb2 / (1 - beta2 ** t)

    # Parameter updates
    W1 -= alpha * mW1_corrected / (np.sqrt(vW1_corrected) + epsilon)
    b1 -= alpha * mb1_corrected / (np.sqrt(vb1_corrected) + epsilon)
    W2 -= alpha * mW2_corrected / (np.sqrt(vW2_corrected) + epsilon)
    b2 -= alpha * mb2_corrected / (np.sqrt(vb2_corrected) + epsilon)

    return W1, b1, W2, b2, mW1, mb1, mW2, mb2, vW1, vb1, vW2, vb2

def sgd_with_adam(X, y, epochs, alpha, beta1, beta2):
    W1, b1, W2, b2 = init_param()
    mW1, mb1, mW2, mb2, vW1, vb1, vW2, vb2 = init_adam(W1, b1, W2, b2)

    m = X.shape[1]
    t = 1
    loss_v = np.zeros(epochs)
    for epoch in range(epochs):
        permutation = np.random.permutation(m)
        X_shuffle = X[:, permutation]
        y_shuffle = y[permutation]

        predictions = []
        #alpha = learningRateDecay(alpha, 0.01, epoch)

        for i in range(m):
            X_i = X_shuffle[:, i:i+1]
            y_i = y_shuffle[i:i+1]

            Z1, A1, Z2, A2 = forward_prop(W1, b1, W2, b2, X_i)
            dW1, db1, dW2, db2 = backward_prop(Z1, A1, Z2, A2, W2, X_i, y_i)
            W1, b1, W2, b2, mW1, mb1, mW2, mb2, vW1, vb1, vW2, vb2 = update_with_ADAM(W1
                     vW1, vb1, vW2, vb2, t, alpha, beta1, beta2)

            predictions.append(get_prediction(A2))

            t += 1

        epoch_predictions = np.concatenate(predictions, axis=1)
        epoch_accuracy = get_accuracy(epoch_predictions, y_shuffle)
        loss = binary_cross_entropy(y_shuffle.reshape(-1,1), epoch_predictions.reshape(-
        loss_v[epoch] = loss
        print(f"Epoch {epoch + 1}/{epochs} - Accuracy: {epoch_accuracy} - loss: {loss}")

    return W1, b1, W2, b2, loss_v
```