Mini-Project: Deep Learning from scratch

Tamer, Abu-shakra 319147260
Tomer, Varsanno 206727471
Naor, Guetta 316017359

April 21, 2024

Contents

Part 1 : The classifier and optimizer

1 Part 1.1 :

1.1 Softmax Regression

We implemented the Soft-Max-Regression function to transform the neural network's final layer into the probability of an instance belonging to a particular class.

Softmax (Z) = $\frac{\exp(xw_j - \eta)}{\sum_{i=1}^{N} \exp(xw_i - \eta)}$

We added the η as recommended to ensure safe computation.

Z : The weights matrix (W) time Data matrix (X)

X : Input data matrix where each row represents a feature and each column represents a sample.
Shape: (number of features, number of samples)

W : Weight matrix representing the parameters of the model.
Shape : (number of classes, number of features)

Returns: Output of the softmax regression, representing the predicted probabilities for each class.
Shape: (number of classes, number of samples)

Code implementation :
$Z = X \cdot W$
Exp = $\exp(\mathbf{Z} - \max(\mathbf{Z}))$
Softmax Output$_i$ = $\frac{\exp(\mathrm{Z}_i)}{\sum_{j=1}^{N} \exp(\mathrm{Z}_j)}$

Here is an example for a 3 data features and 4 data samples:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \\ a_{41} & a_{42} & a_{43} \end{bmatrix}$$

row = Feature
column = Data sample

1.2 Softmax-Regression-Loss : cross-entropy function

softmax regression loss (X,W,C) = $-\frac{1}{m} \sum_{k=1}^{l} c_k \log \left(diag(\sum_{j=1}^{\ell} \exp(XW_j))^{-1} \cdot \exp(XW_k) \right)$

C : One-hot encoded class labels for each sample.
Shape: (number of samples, number of classes)

Returns : Softmax regression loss, calculated as the negative log-likelihood of the true class probabilities.

Code implementation :
Probabilities = soft_max_regression(X, W)
Log Probs = $-\log$(Probabilities)
Loss = $\sum_{i=1}^{m}$ Log Probs$_i$

1.3 Gradients by W and X for SM

Softmax regression gradient by W

softmax regression gradient by W(X,W,C) = $\frac{1}{m} \left[diag(\sum_j \exp(W_j X))^{-1} \cdot \exp(W_k X) - C_p \right] X^T$

Returns : Gradient by W of the softmax regression loss function.
Shape: (number of classes, number of features)

Code implementation :
Probabilities = soft max regression(X, W)
Probabilities = Probabilities $- 1$
Gradient = $\frac{1}{m} \cdot$ Probabilities $\cdot X^T$

```python
def gradient_test_byW(function, gradient):
  # Gradient Check Setup
  batchSize = 2
  EPSILON = 0.1
  outputnum = 2

  X , C =pickRandomBach(data_training_X,data_training_C,batchSize)
  X = np.vstack([X, np.ones((1, X.shape[1]))])

  W = np.random.randn(outputnum, X.shape[0])


  d = np.random.randn(*W.shape)

  def F(W):
      Z = W.dot(X)
      return function(Z, C)

  def g_F(W):
      return gradient(X, W, C)


  F0 = F(W)
  g0 = g_F(W)
  y0 = np.zeros(8)
  y1 = np.zeros(8)

  print("k\terror order 1\t\terror order 2")

  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      Fk = F(W + epsk * d)
      F1 = F0 + epsk * np.sum(g0 * d)
      y0[k-1] = abs(Fk - F0)
      y1[k-1] = abs(Fk - F1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Gradient Test for Softmax Regression By W")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()

gradient_test_byW(compute_loss_for_testing,softmax_regression_gradient_by_W)
```

Figure 1: SM gradient test by W code

```python
def softmax_regression_gradient_by_W(X ,W, C):
    Z = W.dot(X)
    m = Z.shape[1]
    probabilities = softmax(Z)
    # for i in range(m):
    #   probabilities[:, i] -= C[:, i]
    probabilities -= C
    gradient = np.dot(probabilities, X.T) / m
    return gradient
```

```
k       error order 1           error order 2
1       0.093862618502540 52    0.00245180768203 5234
2       0.047550851886846934    0.000606361205440 8862
3       0.023927850725005 03    0.000150755821138 88153
4       0.012001719064543503    3.758420852850719e-05
5       0.006010268629803472    9.383006732477384e-06
6       0.003007481657832 4892  2.344160435541 1982e-06
7       0.001504327047288 2009  5.858618458143283e-07
8       0.000752310001010 1702  1.464535568374 0036e-07
```

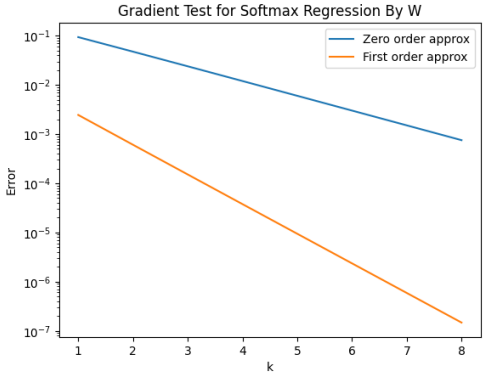

Figure 2: SM gradient by W code + results

Softmax regression gradient by X

softmax regression gradient by X(X,W,C) = $\frac{1}{m} W^T \left[\exp(W_k X) \oslash diag(\sum_j \exp(W_j X))^{-1} - C \right]$

Returns : Gradient by X of the softmax regression loss function.
Shape: (number of features, number of samples)

Code implementation :
Probabilities = soft max regression(X, W)
Probabilities = Probabilities $- C$
Gradient = $\frac{1}{m} \cdot W^T \cdot$ Probabilities

```python
def gradient_test_byX(function, gradient):
  # Gradient Check Setup
  batchSize = 2
  EPSILON = 0.1
  outputnum = 2

  X , C =pickRandomBach(data_training_X,data_training_C,batchSize)

  W1 = np.random.randn(2, 2)

  d = np.random.randn(*X.shape)

  def F(X):
      Z = W1.dot(X)
      return function(Z , C)

  def g_F(X):
      return gradient( X , W1 , C)


  F0 = F(X)
  g0 = g_F(X)
  y0 = np.zeros(8)
  y1 = np.zeros(8)

  print("k\terror order 1\t\terror order 2")

  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      Fk = F(X + epsk * d)
      F1 = F0 + epsk * np.sum(g0 * d)
      y0[k-1] = abs(Fk - F0)
      y1[k-1] = abs(Fk - F1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Gradient Test for Softmax Regression By X")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()

gradient_test_byX(compute_loss_for_testing,softmax_regression_gradient_by_X)
```

Figure 3: SM gradient test by X code

```python
def softmax_regression_gradient_by_X(X, W, C):
    m = X.shape[1]
    A = np.dot(W, X)
    prob = softmax(A)
    P = prob - C
    grad_X = np.dot(W.T, P) / m
    return grad_X
```

```
k       error order 1           error order 2
1       0.010403109720281245    0.00018141415968997432
2       0.005156031466468036    4.518368617240087e-05
3       0.0025666984188740805   1.1274528726235022e-05
4       0.0012805278873428971   2.815942268974414e-06
5       0.0006395596179034624   7.036453665287823e-07
6       0.0003196038533473944   1.7586707889982733e-07
7       0.0001597579535174698   4.396038322251883e-08
8       7.986798543879736e-05   1.0988871701478331e-08
```

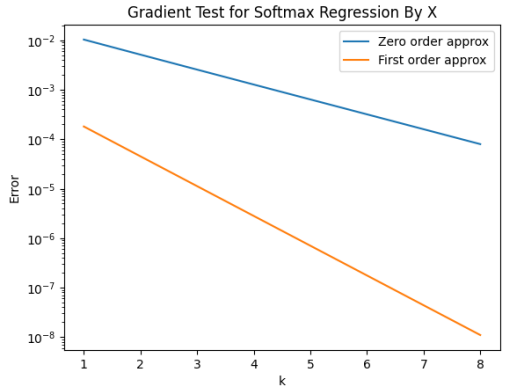

Figure 4: SM gradient by X code + results

2 Part 1.2 :

2.1 SGD on Least Squares

In this section, we used Least Squares in order to check our SGD with Momentum function we wrote the code for LS and LS gradient and checked with Gradient test for correctness.

```
def gradient_test_LS(function, gradient):
  # Gradient Check Setup
  batchSize = 2
  EPSILON = 0.1
  outputnum = 2

  X , C =pickRandomBach(data_training_X,data_training_C,batchSize)
  X = np.vstack([X, np.ones((1, X.shape[1]))])

  W = np.random.randn(outputnum, X.shape[0])


  d = np.random.randn(*W.shape)



  def F(W):
      return function(X, W, C)

  def g_F(W):
      return gradient(X, W, C)


  F0 = F(W)
  g0 = g_F(W)
  y0 = np.zeros(8)
  y1 = np.zeros(8)

  print("k\terror order 1\t\terror order 2")

  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      Fk = F(W + epsk * d)
      F1 = F0 + epsk * np.sum(g0 * d)
      y0[k-1] = abs(Fk - F0)
      y1[k-1] = abs(Fk - F1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Gradient Test for Least Squares")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()

gradient_test_LS(least_squares_loss, least_squares_gradient)
```

Figure 5: LS gradient test code

```
def least_squares_loss(X, W, C):
    # Calculate the least squares loss
    N = len(C)
    loss = (1/N) * np.sum((np.dot(W, X) - C)**2)
    return loss

def least_squares_gradient(X, W, C):
    # Calculate the gradient for least squares regression
    N = len(C)
    gradient = (2/N) * np.dot((np.dot(W, X) - C),X.T)
    return gradient
```

```
k       error order 1           error order 2
1       0.20001922201421962     0.004440367965265772
2       0.09890951901579381     0.0011000919913168872
3       0.049179736510066796    0.0002750229978278895
4       0.024521112505575537    6.875574945652829e-05
5       0.012243367315423193    1.7188937363243895e-05
6       0.006117386423371229    4.297234341699152e-06
7       0.003057618903100412    1.0743085852027434e-06
8       0.0015285408744034612   2.6857714630068585e-07
```

Figure 6: LS code + Results

2.2 Stochastic gradient descent with Momentum

learning_rate : The learning determines the step size in the weight updates during optimization.
momentum : The momentum controls the contribution of the previous velocity to the current update.
epochs : Represents how often the model will iterate over the entire training dataset.

Output:

network : The final network containing the weights after the optimization process.
We choose to use gradient descent with Momentum to try to avoid local minimum.
Code implementation :
$w := w - \eta \nabla Q_i(w) + \alpha \Delta w$
$\Delta w := \alpha \Delta w - \eta \nabla Q_i(w)$

```
def SGD_with_momentum(network, gradients, velocity, learning_rate, momentum):
    for key in network.keys():
        velocity[key] = momentum * velocity[key] - learning_rate * gradients[key]
        network[key] += velocity[key]
```

Figure 7: SGD optimizer code

```python
def testSGD(lossFunction , gradientLossFunction):
  batchSize = 2
  EPSILON = 0.1
  outputnum = 2
  network = {}

  X , C =pickRandomBach(data_training_X,data_training_C,batchSize)
  X = np.vstack([X, np.ones((1, X.shape[1]))])
  W1 = np.random.randn(outputnum, X.shape[0])
  network = {'W1' : W1}
  velocity = {'W1' :np.zeros_like(W1)}

  success_percentages_list = []
  success_percentages = 0
  losses = []

  # Set hyperparameters
  learning_rate = 0.05
  momentum = 0.8
  epochs = 80
  batch_Size = 20

  for i in range(epochs):
    W1 = network['W1']
    X , C =pickRandomBach(data_training_X,data_training_C,batchSize)
    X = np.vstack([X, np.ones((1, X.shape[1]))])
    gradients = {'W1': least_squares_gradient(X, W1, C)}

    SGD_with_momentum(network, gradients, velocity, learning_rate, momentum)

    losses.append(least_squares_loss(X, W1, C))

    success_percentages_list.append(calculate_success_percentage(W1.dot(X), C))


  # Plot the loss over epochs
  plt.plot(range(epochs), losses)
  plt.title("Losses over Epochs (SGD with Momentum) with LS")
  plt.xlabel("Epoch")
  plt.ylabel("losses")
  plt.show()

    # Plot the success percentages over epochs
  plt.plot(range(epochs), success_percentages_list)
  plt.title("success percentages over Epochs (SGD with Momentum) with LS")
  plt.xlabel("Epoch")
  plt.ylabel("success")
  plt.show()
```

Figure 8: SGD LS test code

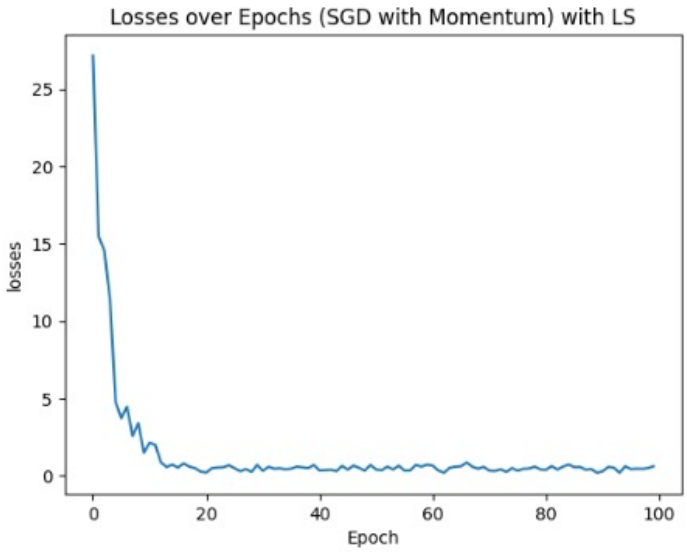

Figure 9: SGD LS test

3 Part 1.3:

3.1 Stochastic gradient descent with Momentum on SoftMax

On the left side is the graph for the Training Data .
And on the right side the graph for the Validation Data .

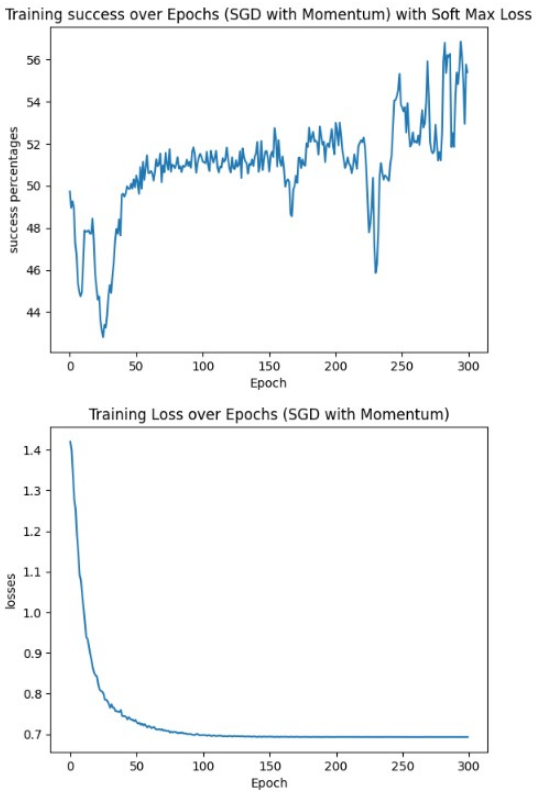

Figure 10: Loss and Success-Percentage - Training Data

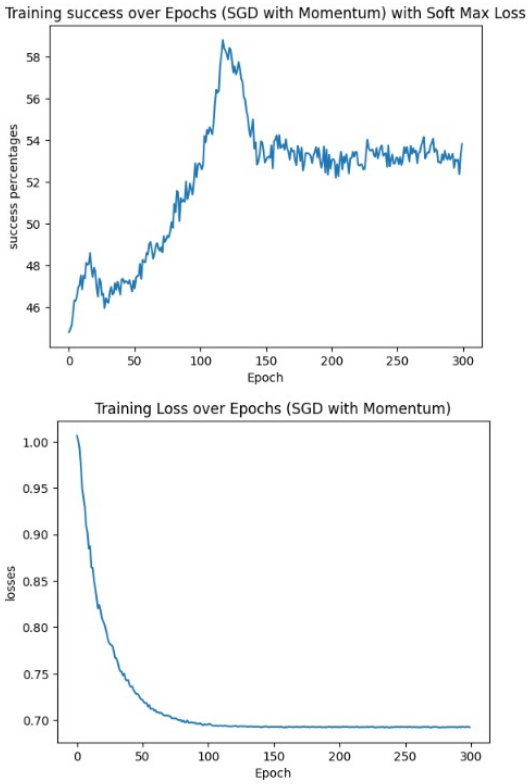

Figure 11: Loss and Success-Percentage - Validation Data

3.2 Batch size

In our testing, we observed that increasing the batch size resulted in smoother and less noisy trends in our results, with a gradual upward or downward movement. Conversely, using a smaller batch size led to more pronounced changes with each epoch. Despite both scenarios resulting in lower losses, the success percentage remained consistently between 45 percent to 60 percent. This outcome was anticipated, considering we haven't yet implemented additional layers in our model.

examples :

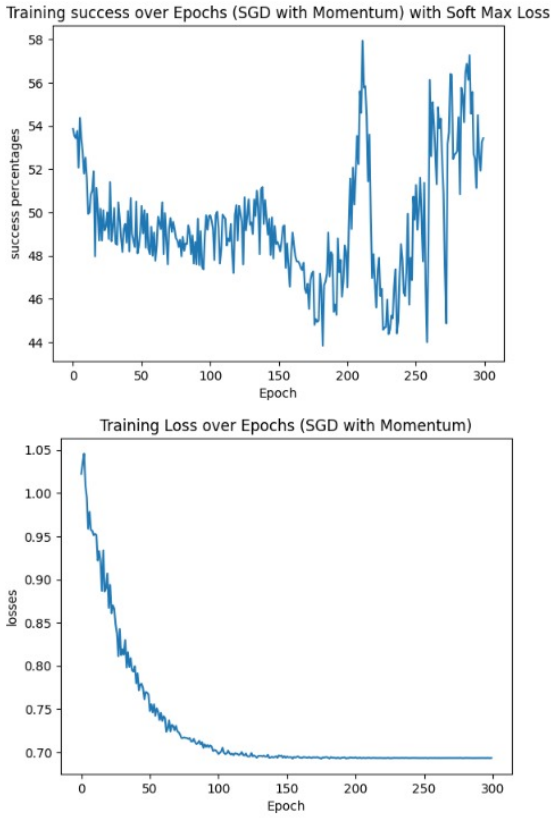

Figure 12: Lower batch size

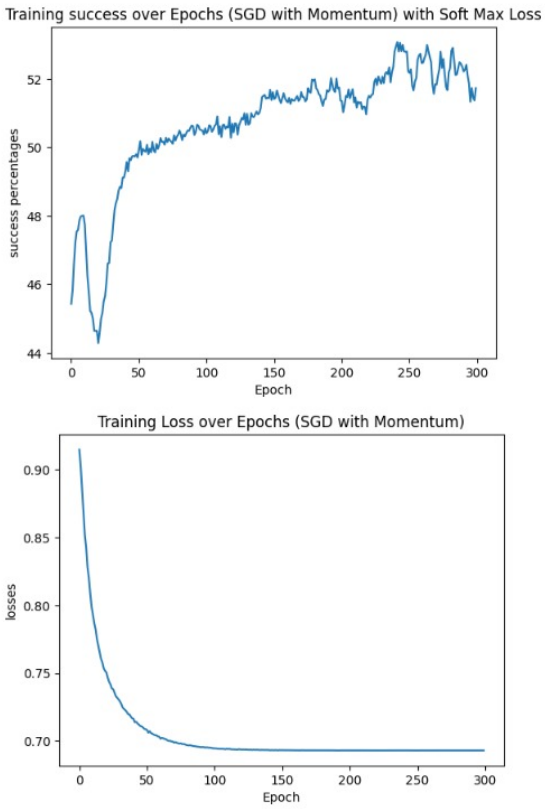

Figure 13: Higher batch size

3.3 Learning rate

Furthermore, Increasing the learning rate caused a substantial increase or decrease in the success percentage, with each epoch producing notable changes. The losses also decreased rapidly. On the other hand, decreasing the learning rate led to more subtle shifts in the success percentage graph. In the loss graph, we observed a slower decline.

examples :

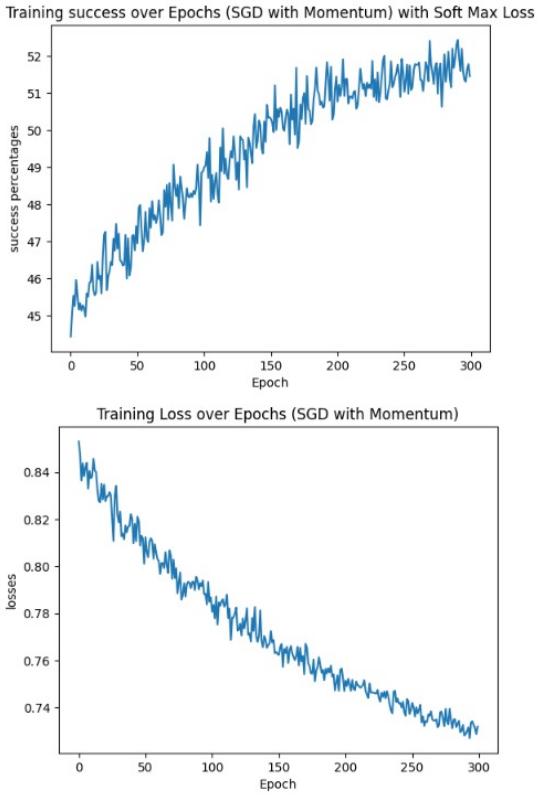

Figure 14: Lower learning rate

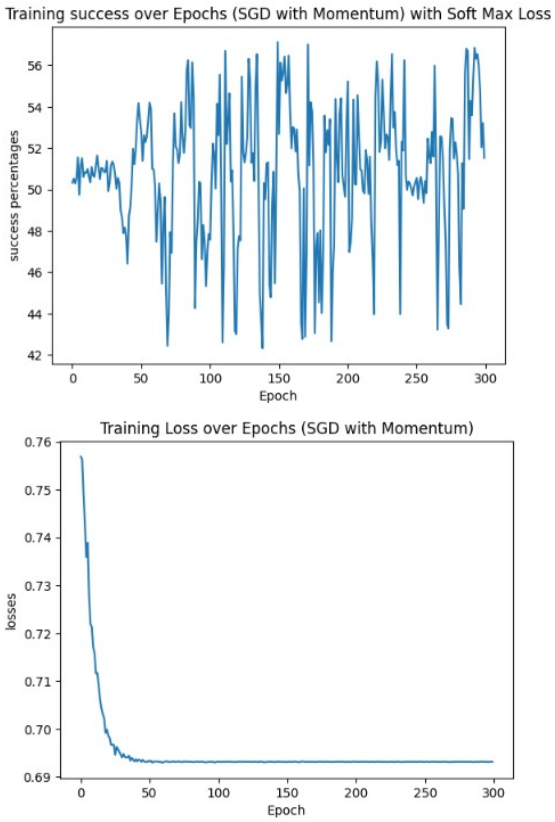

Figure 15: Higher learning rate

Part 2 : the neural network

4 Part 2.1 :

4.1 activation functions

we implemented two activation functions, hyperbolic tangent (tanh) and Rectified Linear Unit (ReLU), along with their derivatives.

4.2 One step forward for the standard neural network

The `single_step_forward_class` function computes a forward step in a standard neural network using a selected activation function.

1. **Select the activation function:** The activation function is chosen based on the input parameter.
$$\text{activation\_func} = \begin{cases} \text{relu} & \text{if activation} == \text{'relu'} \\ \tanh & \text{otherwise} \end{cases}$$
2. **Compute and return the activated output:** The output is computed as follows:
$$\text{activation}(\mathbf{W} \cdot \mathbf{X})$$

```
def single_step_forward_class(X, W, activation='relu'):
    activation_func = relu if activation == 'relu' else tanh

    Z = W.dot(X)
    A = activation_func(Z)
    return A
```

Figure 16: Classic single step forward code

4.3 Verification of Derivatives via Jacobian Test for standard network

This section demonstrates that the computed derivatives with respect to both $\mathbf{X}$ and $\mathbf{W}$ in the neural network model satisfy the Jacobian test.

Derivative with respect to W

$\frac{\partial y}{\partial \mathbf{W}}^T \mathbf{v} = (\sigma'(\mathbf{WX})\mathbf{v})\mathbf{X}^T$

X : Matrix representing the outputs of the previous layer.

W : Weight matrix associated with the current layer, used to transform the input matrix X before applying the activation function.

u: matrix containing the outputs from the forward pass of the current layer.

activation ('relu' by default): Specifies the type of activation function to use.

Returns: Computes and returns the Jacobian transpose matrix time u, This matrix is used in gradient computations for optimizing the weights during the backpropagation process.

Code implementation :

$Z = W \cdot X$

A = activation(Z)

return : $= (\mathbf{A} \circ \mathbf{V})\mathbf{X}^T$

```python
def jacobian_test_by_W_for_classic():
  # Jacobian Check Setup
  num_classes = 3
  input_size = 2
  num_features = 5
  EPSILON = 0.1
  W = np.random.randn(num_classes, num_features + 1)
  X = np.random.randn(num_features, input_size)
  X = np.vstack([X, np.ones((1, X.shape[1]))])
  d = np.random.randn(num_classes, num_features + 1)
  u = np.random.randn(num_classes, input_size)

  def F(W):
      return single_step_forward_class(X, W, tanh)

  def g_F(W):
      return np.dot(u.flatten(), F0.flatten())

  F0 = F(W)
  g0 = g_F(W)
  grad_g0 = JacobianTByW(X, W, u, tanh)

  y0 = np.zeros(8)
  y1 = np.zeros(8)


  print("k\terror order 1 \t\t\t error order 2")
  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      fk = F(W + (epsk * d))
      gk = np.dot(fk.flatten(), u.flatten())
      g1 = g0 + (epsk * (np.dot(d.flatten(),grad_g0.flatten())))
      y0[k - 1] = abs(gk - g0)
      y1[k - 1] = abs(gk - g1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Jacobian test By W for classic")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()
```

Figure 17: Jacobian transposed test code for W'

```python
def JacobianTByW(X, W, u, activation='relu'):

    activationD_func = relu_derivative if activation == 'relu' else tanh_derivative

    Z = W.dot(X)
    A = activationD_func(Z)

    return np.dot(np.multiply(A,u),X.T)
```

```
k       error order 1                   error order 2
1       0.028016602788170086    0.0009390926579160297
2       0.013686160771352762    0.0001474057062256228 8
3       0.006795874998860585    2.6497466297126238e-05
4       0.003390051191240806    5.3624249589656616-06
5       0.0016935291708182199   1.1847876772996813e-06
6       0.0008464490285140513   2.7683694359126326-07
7       0.0004231528922682415   6.679648301144425e-08
8       0.0002115594458693692 6 1.6397976754234378e-08
```

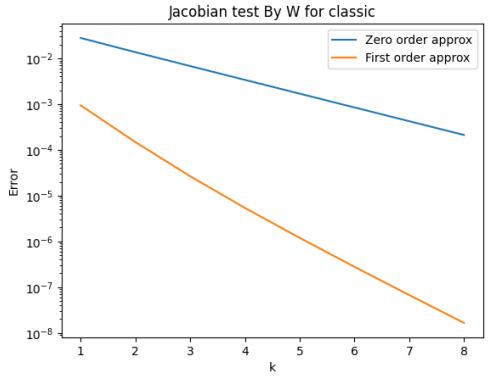

Figure 18: Jacobian by W function + test

Derivative with respect to X

$\frac{\partial y}{\partial \mathbf{X}}^T \mathbf{v} = \mathbf{W}^T(\sigma'(\mathbf{WX})\mathbf{v})$

X : Matrix representing the outputs of the previous layer.

W : Weight matrix associated with the current layer, used to transform the input matrix X before applying the activation function.

u: matrix containing the outputs from the forward pass of the current layer.

activation ('relu' by default): Specifies the type of activation function to use.

Returns: Computes and returns the Jacobian transpose matrix time u.

Code implementation :

$Z = W \cdot X$

A = activation(Z)

return : $\mathbf{W}^T(\mathbf{A} \circ \mathbf{V})$

```python
def jacobian_test_by_X_for_classic():
  # Jacobian Check Setup
  num_classes = 3
  input_size = 2
  num_features = 5
  EPSILON = 0.1
  W = np.random.randn(num_classes, num_features + 1)
  X = np.random.randn(num_features, input_size)
  X = np.vstack([X, np.ones((1, X.shape[1]))])
  d = np.random.randn(num_features + 1, input_size)
  u = np.random.randn(num_classes, input_size)

  def F(X):
      return single_step_forward_class(X, W, tanh)

  def g_F(X):
      return np.dot(u.flatten(), F0.flatten())

  F0 = F(X)
  g0 = g_F(X)
  grad_g0 = JacobianTByX(X, W, u, tanh)

  y0 = np.zeros(8)
  y1 = np.zeros(8)


  print("k\terror order 1 \t\t\t error order 2")
  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      fk = F(X + (epsk * d))
      gk = np.dot(fk.flatten(), u.flatten())
      g1 = g0 + (epsk * (np.dot(d.flatten(),grad_g0.flatten())))
      y0[k - 1] = abs(gk - g0)
      y1[k - 1] = abs(gk - g1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Jacobian test By X for classic")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()
```

Figure 19: Jacobian transposed test code for X'

```python
def JacobianTByX(X, W, u, activation='relu'):

    activationD_func = relu_derivative if activation == 'relu' else tanh_derivative

    Z = W.dot(X)
    A = activationD_func(Z)

    return np.dot(W.T, np.multiply(A,u))
```

```
k       error order 1                   error order 2
1       0.011903876359053434    0.0007912456134713253
2       0.006161838843913525    0.0001857221423487987
3       0.0031287955309935445   4.4984962137673e-05
4       0.0015758205457798713   1.1069700785681924e-05
5       0.0007906995047810561   2.7456185017760015e-06
6       0.0003960388665364966 7 6.836951048638795e-07
7       0.0001981906947143841   1.705861063516778e-07
8       9.913803602878968e-05   4.260438157821511e-08
```

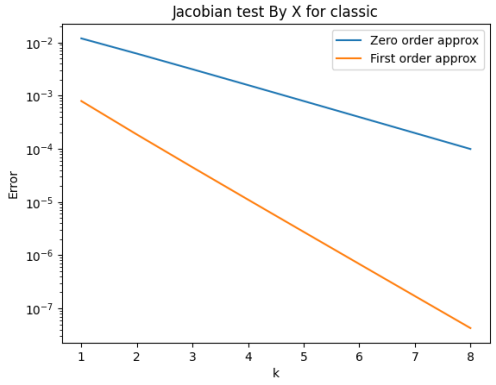

Figure 20: Jacobian by X function + test

5 Part 2.2 :

5.1 One step forward for the residual neural network

The `single_step_forward_res` function computes a forward step in a residual block using a selected activation function, and using the following formula :

step_forward $= \mathbf{X} + \mathbf{W}_2 \left(\sigma(\mathbf{W}_1 \mathbf{X})\right)$

Code implementation :

$Z_{\text{res1}} = \mathbf{W}_1 \cdot \mathbf{X}$

$A_{\text{res1}} = \text{activation}(Z_{\text{res1}})$

$Z_{\text{res2}} = \mathbf{W}_2 \cdot A_{\text{res1}}$

return: $Z_{\text{res2}} + \mathbf{A}$

```python
def single_step_forward_res(X, W1, W2, activation='relu'):
    activation_func = relu if activation == 'relu' else tanh

    #step 1 :
    Z_res1 = W1.dot(X)
    A_res1 = activation_func(Z_res1)

    #step 2 :
    Z_res2 = W2.dot(A_res1)

    A = X[:-1, :]

    Z_res2 += A

    return Z_res2
```

Figure 21: Resnet single step forward code

5.2 Verification of Derivatives via Jacobian Test for residual network

This section verifies that the derivatives with respect to $\mathbf{X}$, $\mathbf{W}_1$, and $\mathbf{W}_2$ in the ResNet model successfully pass the Jacobian test.

Derivative with respect to $\mathbf{W}_1$

$$\frac{\partial y}{\partial \mathbf{W}_1}^T \mathbf{v} = \left(\sigma'(\mathbf{W}_1\mathbf{X}) \circ (\mathbf{W}_2^T\mathbf{u})\right)\mathbf{X}^T$$

Code implementation :

$A_{\text{res1}} = \text{activation derivative}(\mathbf{W}_1\mathbf{X})$

$Z_{\mathbf{u}} = (\mathbf{W}_2^T\mathbf{u})$

return: $(A_{\text{res1}} \circ Z_{\mathbf{u}})\mathbf{X}^T$

```python
def jacobian_test_res_w1():
  # Jacobian Check Setup
  num_classes = 3
  input_size = 2
  num_features = num_classes
  EPSILON = 0.1
  W1 = np.random.randn(num_classes, num_features + 1)
  W2 = np.random.randn(num_classes, num_features )
  X = np.random.randn(num_features, input_size)
  X = np.vstack([X, np.ones((1, X.shape[1]))])
  d = np.random.randn(num_classes, num_features + 1)
  u = np.random.randn(num_classes, input_size)

  def F(W1):
      return single_step_forward_res(X, W1, W2, tanh)

  def g_F(W1):
      return np.dot(u.flatten(), F0.flatten())

  F0 = F(W1)
  g0 = g_F(W1)
  grad_g0 = JacobianT_W1_Res(X, W1, W2, u, tanh)

  y0 = np.zeros(8)
  y1 = np.zeros(8)

  print("k\terror order 1 \t\t\t error order 2")
  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      fk = F(W1 + (epsk * d))
      gk = np.dot(fk.flatten(), u.flatten())
      g1 = g0 + (epsk * (np.dot(d.flatten(),grad_g0.flatten())))
      y0[k - 1] = abs(gk - g0)
      y1[k - 1] = abs(gk - g1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Jacobian test By W1 for ResNet")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()
```

Figure 22: Jacobian transposed test code for W1'

```python
def JacobianT_W1_Res(X, W1, W2, u, activation='relu'):
    activationD_func = relu_derivative if activation == 'relu' else tanh_derivative

    Z_res1 = W1.dot(X)
    A_res1 = activationD_func(Z_res1)

    Z_u = W2.T.dot(u)
    Z_res2 = np.multiply(A_res1,Z_u)

    return Z_res2.dot(X.T)
```

```
k       error order 1                   error order 2
1       0.113090638071082777    0.002403180039270758
2       0.05595436527765063     0.0006106362617446237
3       0.027825704396100548    0.0001538398881475444 6
4       0.013874536730555905    3.860447658254884e-05
5       0.006927635115197006    9.668988208311191e-06
6       0.003461402532044744    2.41946855084052e-06
7       0.001730096675813364    6.05144066412322e-07
8       0.000864897086460203 5  1.5132058628353207e-07
```

Figure 23: Jacobian by W1 function + test

Derivative with respect to $\mathbf{W}_2$

$$\frac{\partial y}{\partial \mathbf{W}_2}^T \mathbf{v} = \mathbf{u}\left(\sigma(\mathbf{W}_1\mathbf{X})^T\right)$$

Code implementation :

$A = \text{activation}(\mathbf{W}_1\mathbf{X})$

return: $(\mathbf{u}\mathbf{A}^T)$

```python
def jacobian_test_res_w2():
  # Jacobian Check Setup
  num_classes = 3
  input_size = 2
  num_features = num_classes
  EPSILON = 0.1
  X = np.random.randn(num_features, input_size)
  X = np.vstack([X, np.ones((1, X.shape[1]))])
  W1 = np.random.randn(num_features, num_classes + 1)
  W2 = np.random.randn(num_features, num_classes)
  d = np.random.randn(num_features , num_classes)
  u = np.random.randn(num_features, input_size)


  def F(W2):
      return single_step_forward_res(X, W1, W2, tanh)

  def g_F(W2):
      return np.dot(u.flatten(), F0.flatten())

  F0 = F(W2)
  g0 = g_F(W2)
  grad_g0 = JacobianT_W2_Res(X, W1, W2, u, tanh)

  y0 = np.zeros(8)
  y1 = np.zeros(8)


  print("k\terror order 1 \t\t\t error order 2")
  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      fk = F(W2 + (epsk * d))
      gk = np.dot(fk.flatten(), u.flatten())
      g1 = g0 + (epsk * (np.dot(d.flatten(),grad_g0.flatten())))
      y0[k - 1] = abs(gk - g0)
      y1[k - 1] = abs(gk - g1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Jacobian test By W2 for ResNet")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()
```

Figure 24: Jacobian transposed test code for W2'

```python
def JacobianT_W2_Res(X, W1, W2, u, activation='relu'):
    activation_func = relu if activation == 'relu' else tanh

    Z_res1 = W1.dot(X)
    A = activation_func(Z_res1)
    A = A.T

    return u.dot(A)
```

```
k       error order 1                   error order 2
1       0.06631118448239404     1.5543122344752192e-15
2       0.033155592241196796    6.661338147750939e-16
3       0.016577796120599952    1.7763568394002505e-15
4       0.008288898060300642    1.5543122344752192e-15
5       0.004144449030151431    1.9984014443252818e-15
6       0.0020722245150754937   6.661338147750939e-16
7       0.0010361122575393011   1.9984014443252818e-15
8       0.0005180561287700947   1.3322676295501878e-15
```

Figure 25: Jacobian by W2 function + test

Test results : In the test, the differing results can be attributed to the variable $W_2 \cdot Z$ not undergoing a linear process.

Derivative with respect to X

$$\frac{\partial y}{\partial \mathbf{X}}^T \mathbf{v} = \mathbf{u} + \mathbf{W}_1^T \left(\sigma'(\mathbf{W}_1\mathbf{X}) \circ (\mathbf{W}_2^T \mathbf{u})\right)$$

Code implementation :

$A_{\text{res1}} = \text{activation derivative}(\mathbf{W}_1\mathbf{X})$

$Z_{\mathbf{u}} = \mathbf{W}_2^T \mathbf{u}$

$\text{Jac} = \mathbf{W}_1^T \left(A_{\text{res1}} \circ Z_{\mathbf{u}}\right)$

return: $\text{Jac} + \mathbf{u}$

The line Jac = Jac[:-1,:] removes the last row from the Jacobian matrix Jac. This row corresponds to the gradients related to the bias term, which is always set to one and does not change during training.

```python
def jacobian_test_res_X():
  # Jacobian Check Setup
  num_classes = 3
  input_size = 2
  num_features = num_classes
  EPSILON = 0.1
  W1 = np.random.randn(num_classes, num_features + 1)
  W2 = np.random.randn(num_classes, num_features )
  X = np.random.randn(num_features, input_size)
  d = np.random.randn(num_features, input_size)
  u = np.random.randn(num_classes, input_size)


  def F(X):
      A = np.vstack([X, np.ones((1, X.shape[1]))])
      return single_step_forward_res(A, W1, W2, tanh)

  def g_F(X):
      return np.dot(u.flatten(), F0.flatten())

  def jaco(X):
      A = np.vstack([X, np.ones((1, X.shape[1]))])
      return JacobianT_X_Res(A, W1, W2, u, tanh)

  F0 = F(X)
  g0 = g_F(X)
  grad_g0 = jaco(X)


  y0 = np.zeros(8)
  y1 = np.zeros(8)


  print("k\terror order 1 \t\t\t error order 2")
  for k in range(1, 9):
      epsk = EPSILON * (0.5 ** k)
      fk = F(X + (epsk * d))
      gk = np.dot(fk.flatten(), u.flatten())
      g1 = g0 + (epsk * (np.dot(d.flatten(),grad_g0.flatten())))
      y0[k - 1] = abs(gk - g0)
      y1[k - 1] = abs(gk - g1)
      print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")

  # Plotting
  plt.semilogy(range(1, 9), y0, label="Zero order approx")
  plt.semilogy(range(1, 9), y1, label="First order approx")
  plt.legend()
  plt.title("Jacobian test By X for ResNet")
  plt.xlabel("k")
  plt.ylabel("Error")
  plt.show()
```

Figure 26: Jacobian transposed test code for X'

```python
def JacobianT_X_Res(X, W1, W2, u, activation='relu'):
    activationD_func = relu_derivative if activation == 'relu' else tanh_derivative

    #step 1 :
    Z_res1 = W1.dot(X)
    A_res1 = activationD_func(Z_res1)


    #step 2 :
    Z_2 = np.dot(W2.T,u)
    A_res2 = np.multiply(A_res1,Z_2)

    #step 3 :
    Jac = np.dot(W1.T,A_res2)
    Jac = Jac[:-1,:]
    Jac += u

    return Jac
```

```
k       error order 1                   error order 2
1       0.13425010692488937     0.00810507824945117
2       0.06507505468657748     0.0020025403488583837
3       0.032033295827931 91    0.0004970386590723574
4       0.015891903353307812    0.00012377476887803773
5       0.007914945288419517    3.0880996204629696e-05
6       0.0039497444317679875   7.712285660543827e-06
7       0.001972943141266059    1.927068212337346e-06
8       0.000985989677696919    4.816411700581114e-07
```

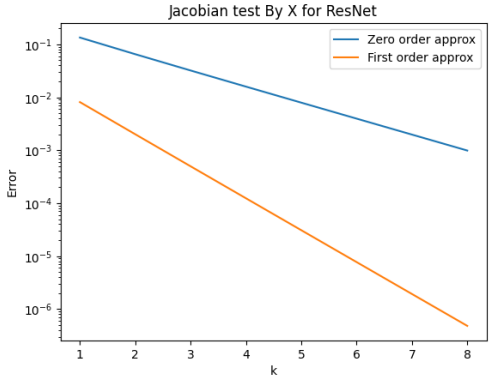

Figure 27: Jacobian by X function + test

6 Part 2.3 :

6.1 Initialize network

The Initialize network function configures a dictionary network with weight matrices for a neural network. It creates standard weights sized by each layer and its previous layer, and for residual layers, it generates two weights ensuring size compatibility between the intermediate and main layers.

6.2 Forward pass

The forward pass function conducts the data propagation through a neural network, applying different processing steps for standard and residual layers, and stores the intermediate outputs.

network : The weight matrices for each layer initialized by initialize network.

X : The initial data matrix fed into the network.

activation : Specifies the activation function (relu or tanh) used.

res places : A list identifying which layers have a residual structure for alternative processing steps.

Outputs:

A : The output from the last layer after applying the softmax function, representing the network's final prediction.

cache: A dictionary storing outputs from each layer, used later in the back propagation process.

```
def forward_pass(network, X, activation='relu', res_places=[]):
    activation_func = relu if activation == 'relu' else tanh
    cache = {'A0': X}
    A = X
    L = len(network) - (len(res_places))
    for i in range(1, L):
        # bias
        A_aug = np.vstack([A, np.ones((1, A.shape[1]))])
        # resNet
        if i-1 in res_places:
            W1 = network['W' + str(i)+'_res1']
            W2 = network['W' + str(i)+'_res2']
            A = single_step_forward_res(A_aug, W1, W2, activation)

        # classic
        else:
            W = network['W' + str(i)]
            A = single_step_forward_class(A_aug, W , activation)

        # caching
        cache['A' + str(i)] = A

    #last layer
    A_aug = np.vstack([A, np.ones((1, A.shape[1]))])
    A = softmax(network['W' + str(L)].dot(A_aug))
    cache['A' + str(L)] = A

    return A, cache
```

Figure 28: Forward pass

6.3 Back propagation

The back propagation function calculates gradients for updating the weights based on the error between the predicted outputs and actual outputs. It utilizes derivatives defined and tested above, operating on both standard and residual layers.

gradients : A dictionary to store the computed gradients for each weight matrix in the network.

Y : The actual output matrix used for training, against which the predicted outputs are compared.

return : The updated dictionary containing gradients for all weight matrices

```
def backpropagation(gradients,network, cache, Y, res_places = [], activation='relu'):
    activation_func = relu if activation == 'relu' else tanh

    L = len(network) - (len(res_places))
    A = cache['A' + str(L - 1)]
    A = np.vstack([A, np.ones((1, A.shape[1]))])
    W_prime = softmax_regression_gradient_by_W(A ,network['W'+str(L)], Y)
    gradients['W'+str(L)] = W_prime
    dA =  softmax_regression_gradient_by_X(A ,network['W'+str(L)], Y)
    dA = dA[:-1,:]

    for i in reversed(range(1, L)):
        A = cache['A'+str(i-1)]
        A_aug = np.vstack([A, np.ones((1, A.shape[1]))])

        #resBack
        if i - 1 in res_places:
            W1 = network['W'+str(i)+'_res1']
            W2 = network['W'+str(i)+'_res2']

            W1_prime = JacobianT_W1_Res(A_aug, W1, W2, dA, activation)
            gradients['W'+str(i)+'_res1'] = W1_prime

            W2_prime = JacobianT_W2_Res(A_aug, W1, W2, dA, activation)
            gradients['W'+str(i)+'_res2'] = W2_prime

            dA = JacobianT_X_Res(A_aug, W1, W2, dA, activation)

        #clasicBack
        else:
            W = network['W'+str(i)]
            W_prime = JacobianTByW(A_aug, W, dA, activation)
            gradients['W'+str(i)] = W_prime
            dA = JacobianTByX(A_aug, W, dA, activation)
            dA = dA[:-1,:]

    return gradients
```

Figure 29: Back propagation

6.4 Gradient test for the complete network

The graph below illustrates the results of our gradient testing across all layers of the neural network, following both the forward pass and back propagation processes.

```python
def Gradient_test_for_complete_network():
    # Gradient test SetUp
    layer_sizes = [2,4,4,4,4,4,4,2]
    res_places = [1,2,3,4,5]
    batchSize = 3
    gradients = {}
    EPSILON = 0.1
    network = initialize_network(layer_sizes, res_places)
    X , C =pickRandomBach(data_training_X,data_training_C,batchSize)

    # Generate a random vector d for each layer
    d = {layer: np.random.randn(network[layer].shape[0], network[layer].shape[1]) for layer in network}

    def F(network):
      A , cache = forward_pass(network, X, 'tanh', res_places)
      return  compute_loss(A,C) , cache

    def g_F(network , cache):
      return backpropagation(gradients, network, cache, C, res_places, 'tanh')

    F0, cache = F(network)
    g0 = g_F(network , cache)
    y0 = np.zeros(8)
    y1 = np.zeros(8)

    for k in range(1, 9):
        epsk = EPSILON * (0.5 ** k)
        # Perturb weights by epsk * d
        W_original = {layer: np.copy(network[layer]) for layer in network}
        for layer in network:
            network[layer] += epsk * d[layer]

        Fk,_ = F(network)
        temp = 0
        for key in g0:
            temp += np.sum(g0[key] * (epsk * d[key]))
        F1 = F0 + temp
        y0[k-1] = abs(Fk - F0)
        y1[k-1] = abs(Fk - F1)
        # Reset weights
        for layer in network:
            network[layer] = W_original[layer]

        print(f"{k}\t{y0[k-1]}\t{y1[k-1]}")
    # Plotting
    plt.semilogy(range(1, 9), y0, label="Zero order approx")
    plt.semilogy(range(1, 9), y1, label="First order approx")
    plt.legend()
    plt.title("Gradient Test for complete network")
    plt.xlabel("k")
    plt.ylabel("Error")
    plt.show()
```

Figure 30: Jacobian transposed test code for complete NN

```
1       0.07757431191911979     0.028203860983774143
2       0.03118679516311662     0.0065015696954437985
3       0.013903009816869316    0.0015603970830329O5
4       0.006553594460344869    0.0003822880934266637
5       0.003180270669744356    9.461748628525335e-05
6       0.0015663630826638908   2.3536490934339405e-05
7       0.0007728277741695 44   5.8694815522342125e-06
8       0.0003871721951691498   1.4655472367897104e-06
```

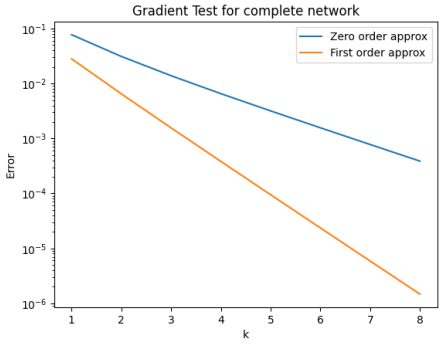

Figure 31: Test results of Jacobian transposed for complete NN

7 Part 2.4 :

7.1 train network

The train network function manages the training of a neural network through multiple iterations, using specified hyper parameters and settings. It performs forward and backward passes, applies stochastic gradient descent with momentum, and calculates losses and success rates for each epoch. The function outputs the trained network along with the success rates and loss values per iteration. These metrics are collected to evaluate and visualize the network's performance.

7.2 data validation test

The data validation test function evaluates the performance of the trained neural network on the validation dataset. The function takes the trained network and processes the validation data through the network. returns the calculated loss and success rate.

7.3 Optimal Network Configurations: Evaluating Different Lengths

We conducted experiments with three different neural network architectures, varying the number of layers to study the impact on training performance and epoch requirements. The details of the hyperparameters used in each configuration are as follows:

Tests on "Swiss Roll"

Network Configurations:

- **Batch Size:** 64
- **Learning Rate:** 0.09
- **Momentum:** 0.2
- **Activation Function:** ReLU
- **Epochs Tested:** 25, 50, 100, 200

Each network configuration also employed a specific architecture setup:

- Each network started with a standard layer that scaled from an input size of 2 units to 8 units.
- The middle layers consisted of residual network (ResNet) blocks with 8 units.
- The final layer was a standard layer that reduced the size back to 2 units.

First Test: 4 layers

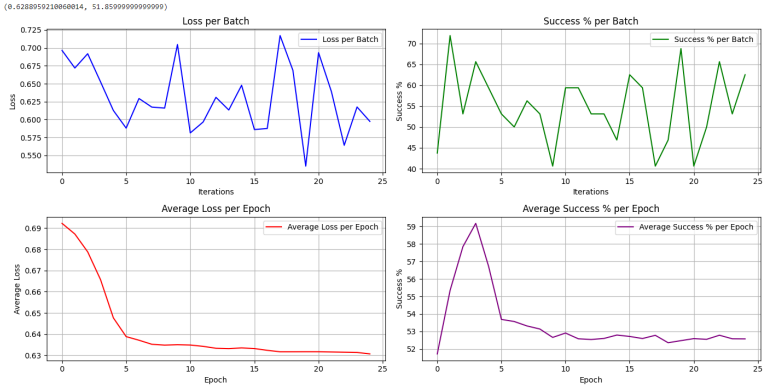

Figure 32: Results for 25 epochs - 51.8% success

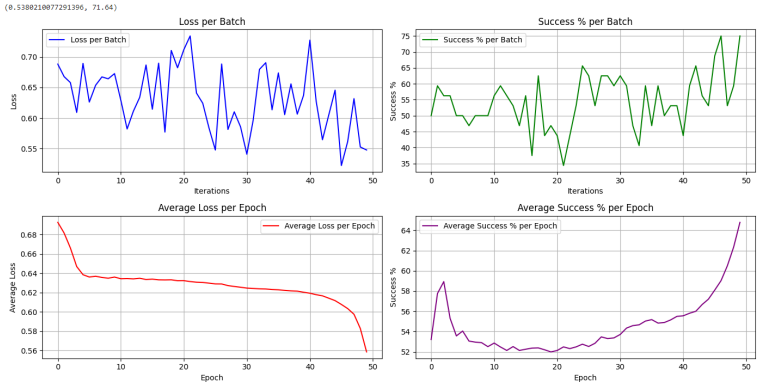

Figure 33: Results for 50 epochs - 71.6% success

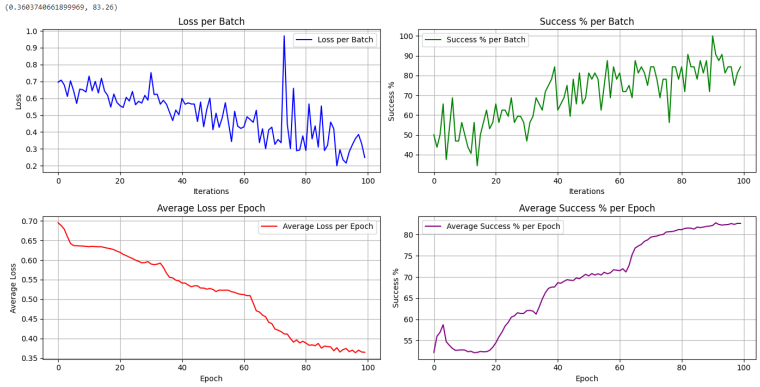

Figure 34: Results for 100 epochs - 83.2% success

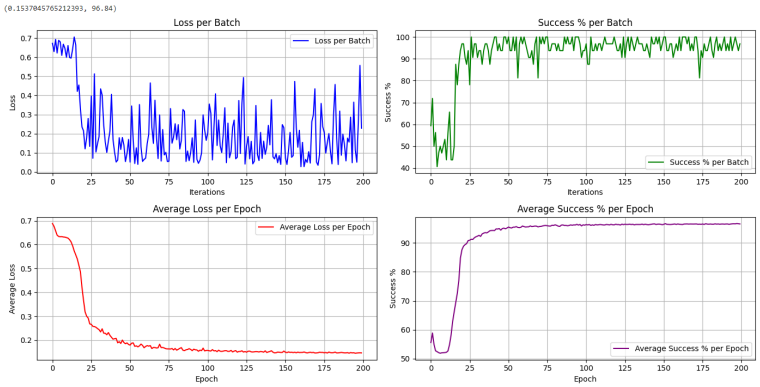

Figure 35: Results for 200 epochs - 96.8% success

Second Test: 7 layers

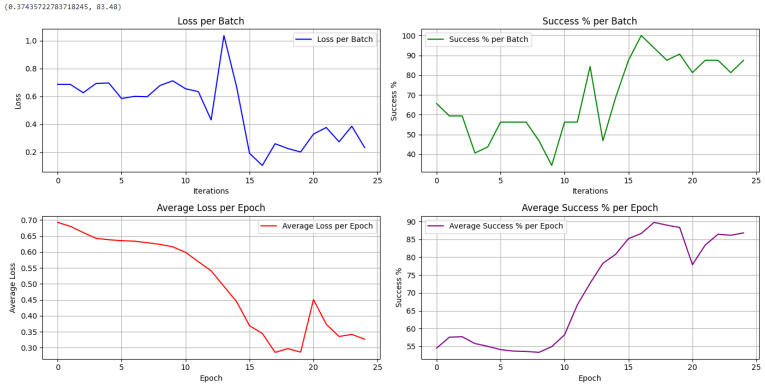

Figure 36: Results for 25 epochs - 83.4% success

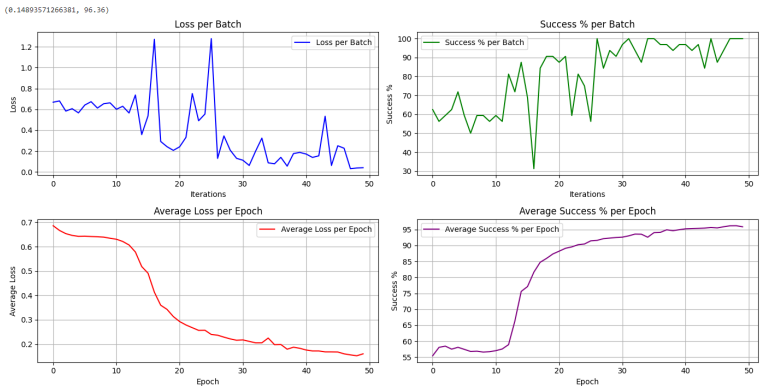

Figure 37: Results for 50 epochs - 96.3% success

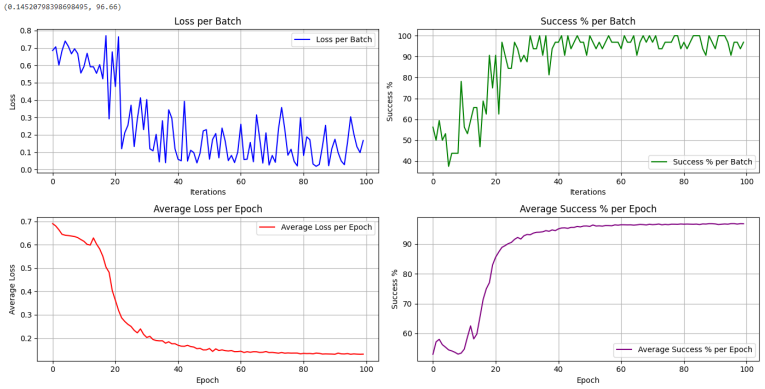

Figure 38: Results for 100 epochs - 96.6% success

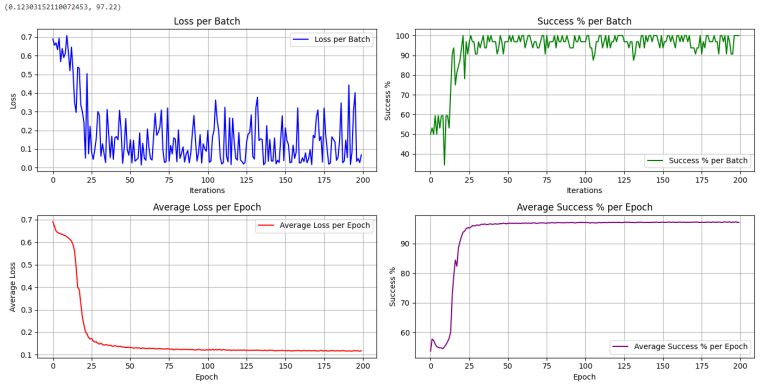

Figure 39: Results for 200 epochs - 97.2% success

Third Test: 12 layers

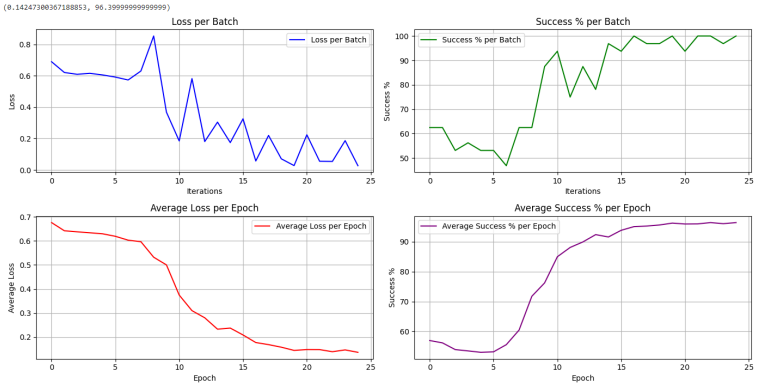

Figure 40: Results for 25 epochs - 96.3% success

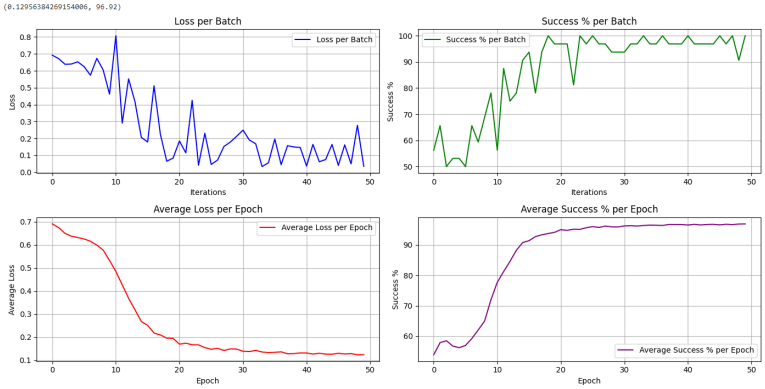

Figure 41: Results for 50 epochs - 96.9% success

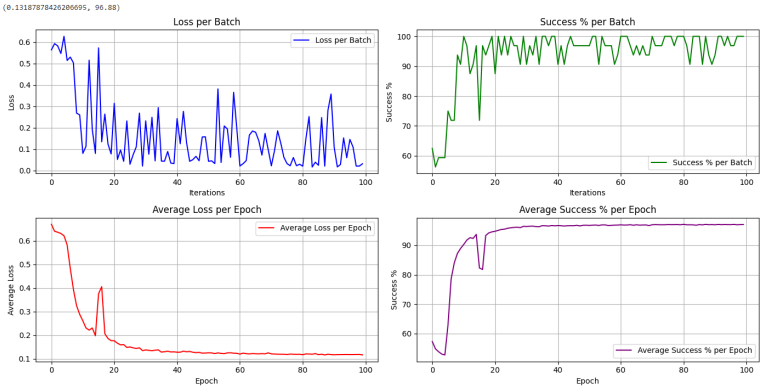

Figure 42: Results for 100 epochs - 96.8% success

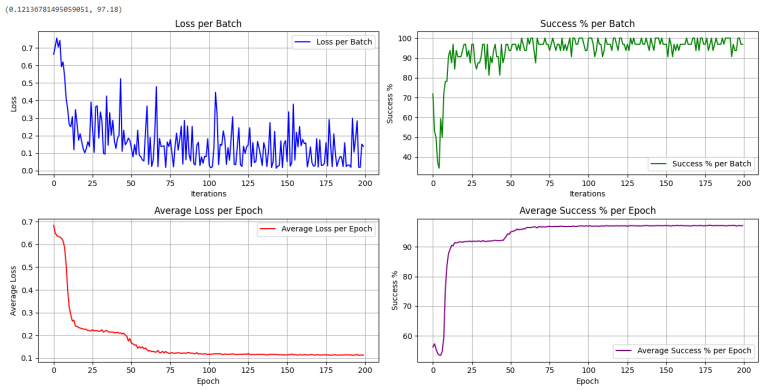

Figure 43: Results for 200 epochs - 97.18% success

In our experiments with varying sizes, we observed that the smaller NN required approximately 200 epochs to achieve satisfactory results. Conversely, the mid-sized layer NN reached comparable outcomes within just 50 epochs. The largest NN configuration completed training in merely 25 epochs. achieving similar results, however, additional training for another 175 epochs only improved performance marginally by approximately 0.4% to 1%.

Conclusion : While larger neural networks are more efficient in terms of epochs needed, they are costlier in terms of time and space. The medium-sized network presents a good compromise, balancing performance, time, and resource demands

The following section presents additional experiments where we tested various layer dimensions and batch sizes.

Tests on "GMM" - layer sizes

Network Configurations:

- **Number of layers:** 5
- **Batch Size:** 64
- **Learning Rate:** 0.09
- **Momentum:** 0.2
- **Activation Function:** ReLU
- **Epochs Tested:** 50, 100

Each network configuration also employed a specific architecture setup:

- Each network started with a standard layer that scaled from an input size of 5 units to 4/8/16 units.
- The middle layers consisted of residual network (ResNet) blocks with 4/8/16 units.
- The final layer was a standard layer that reduced the size back to 5 units.

First Test: 4 units per hidden layer

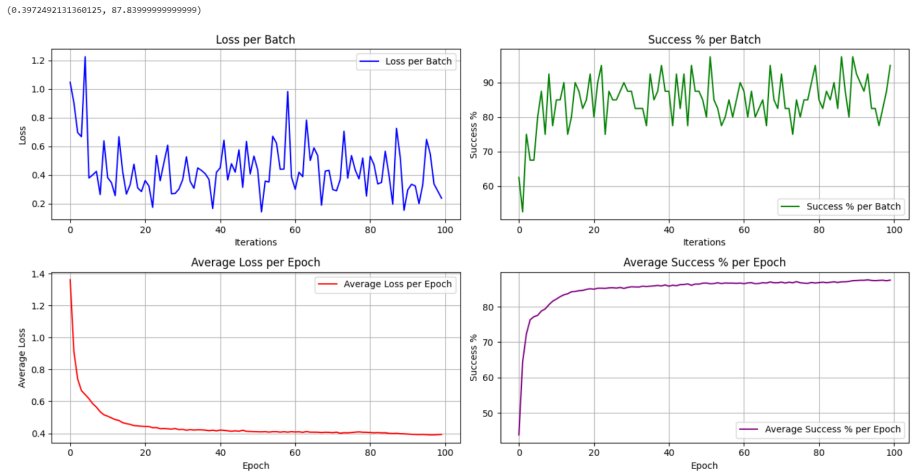

Figure 44: 4 units per hidden layer - 87.8% success

Second Test: 8 units per hidden layer

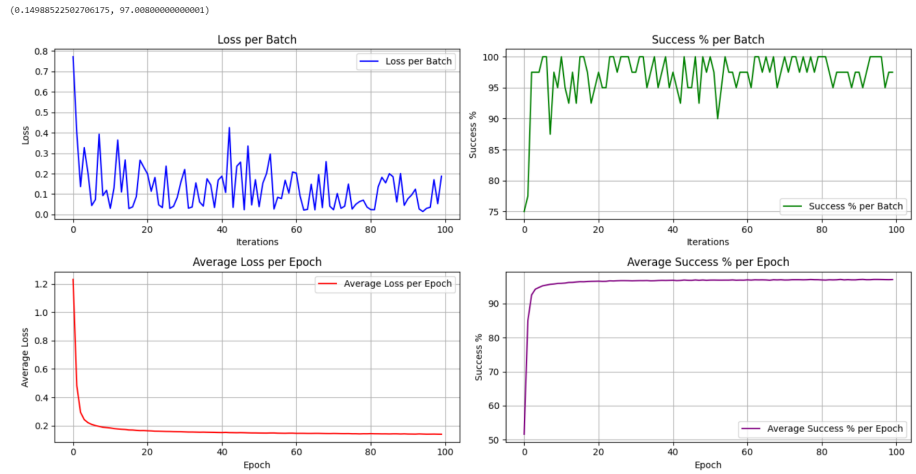

Figure 45: 8 units per hidden layer - 97% success

Third Test: 16 per hidden layer

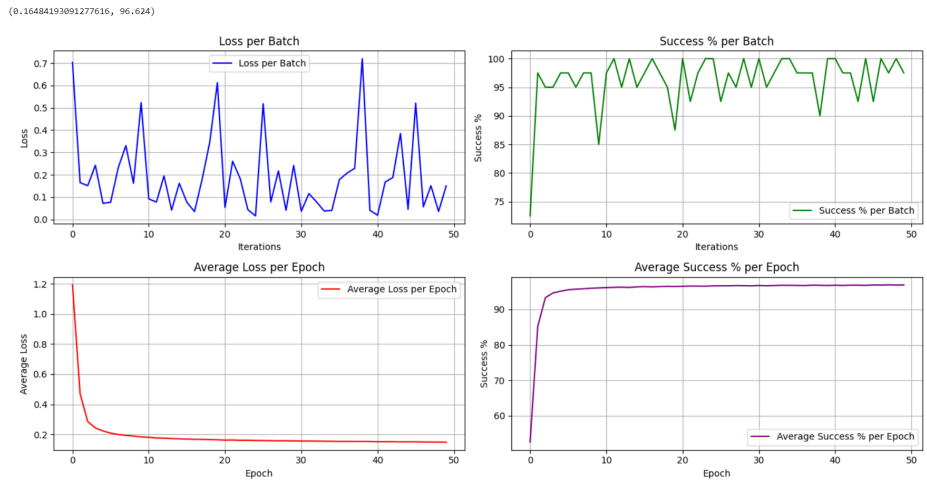

Figure 46: 16 units per hidden layer - 96.6% success

Conclusion : Our findings indicate that larger layers in neural networks yield higher performance outcomes; however, they also demand greater costs in terms of computational time and resource usage. Additionally, beyond a certain point, increasing the layer size does not result in improved results, suggesting a threshold beyond which further enlargement is inefficient.

Tests on "GMM" - batch sizes

Network Configurations:

- **Number of layers:** 5
- **Learning Rate:** 0.09
- **Momentum:** 0.2
- **Activation Function:** ReLU
- **Epochs:** 100

Each network configuration also employed a specific architecture setup:

- Each network started with a standard layer that scaled from an input size of 5 units to 16 units.
- The middle layers consisted of residual network (ResNet) blocks with 16 units.
- The final layer was a standard layer that reduced the size back to 5 units.

First Test: 250 Batch size

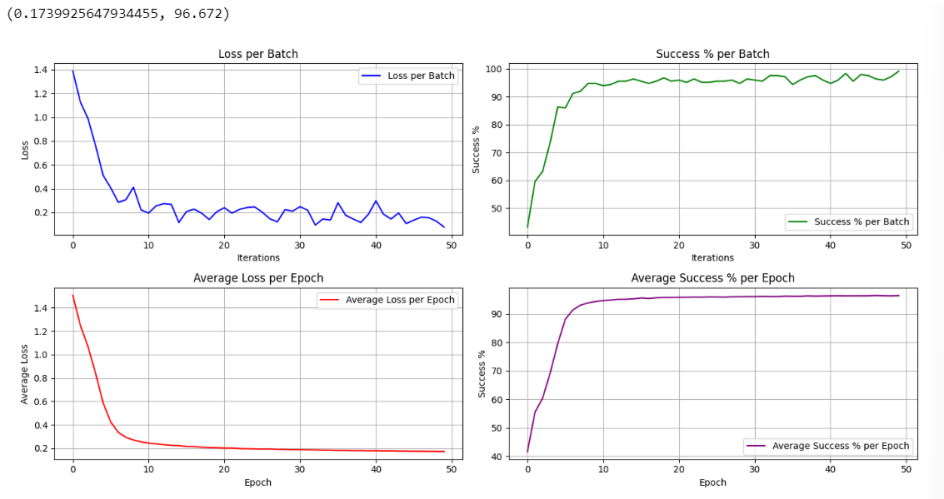

Figure 47: Batch size 250 - 96.6% success

Second Test: 1000 Batch size

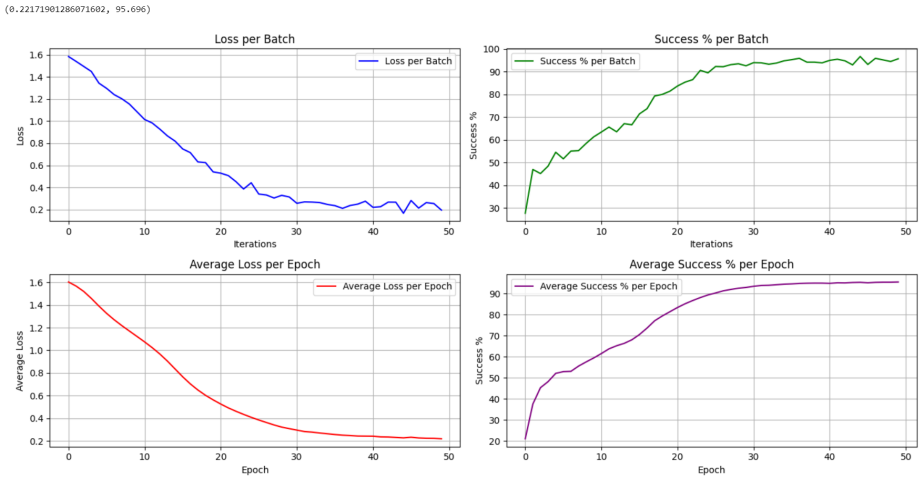

Figure 48: Batch size 1000 - 95.6% success

Third Test: 2500 Batch size

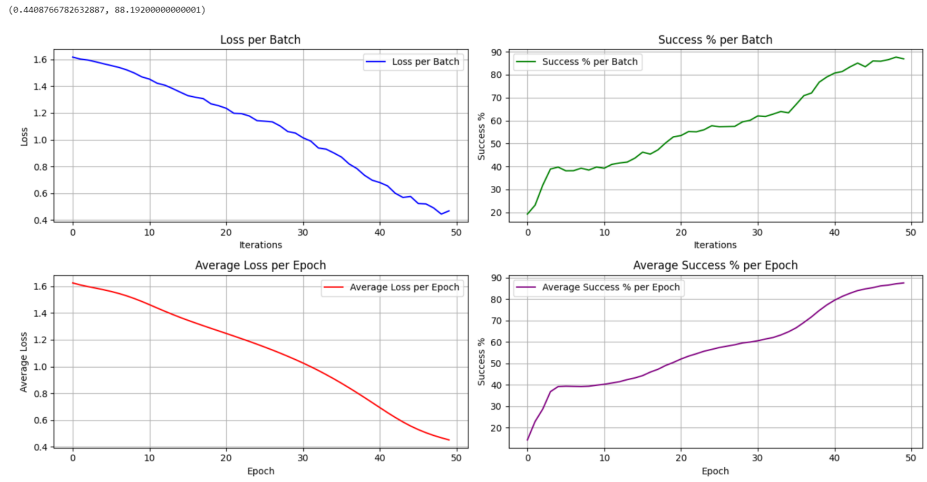

Figure 49: Batch size 2500 - 88.1% success

Fourth Test: 5000 Batch size

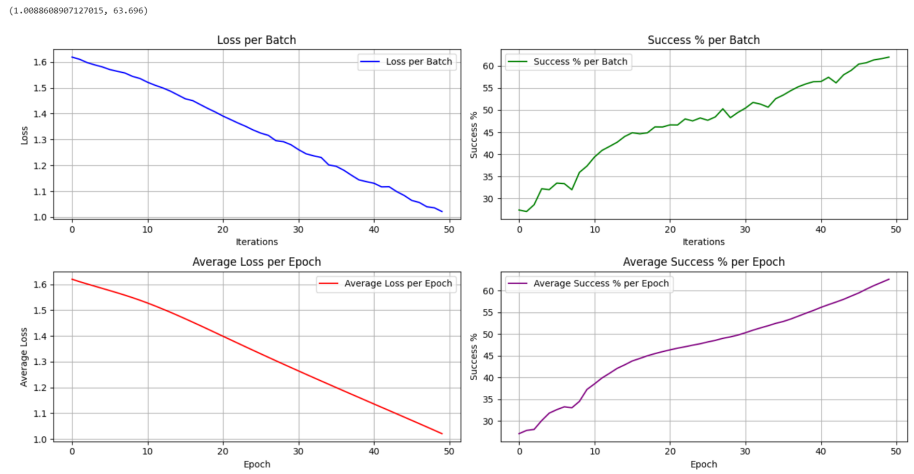

Figure 50: Batch size 5000 - 63.6% success

Conclusion : Our experiments reveal that smaller batch sizes generally lead to better results. Furthermore, excessively large batch sizes require significantly more epochs to train effectively. The graph above illustrates that networks with smaller batch sizes not only minimize loss more effectively but also achieve faster increases in success percentage. Based on these observations, we conclude that a much smaller batch size is preferable for efficient training and enhanced performance.

8 Part 2.5

8.1 Optimizing Neural Network Efficiency - "Swiss Roll"

Next, we focused on maximizing performance while keeping the network size below the threshold of 100C scalar parameters in terms of weight count. We explored various configurations, ranging from 'long' networks with many layers to more compact networks with up to four layers. The optimal configuration emerged as a balance between network length and weight density—neither the longest nor the heaviest in terms of weights.

Optimal Performance Achieved:

- **Layer Sizes:** $[2, 6, 6, 6, 6, 6, 2]$.
- **Batch Size:** 64 .
- **Number of Epochs:** 40 .
- **Learning Rate:** 0.09,
- **Momentum:** 0.2.
- **Activation Function:** 'ReLU'.

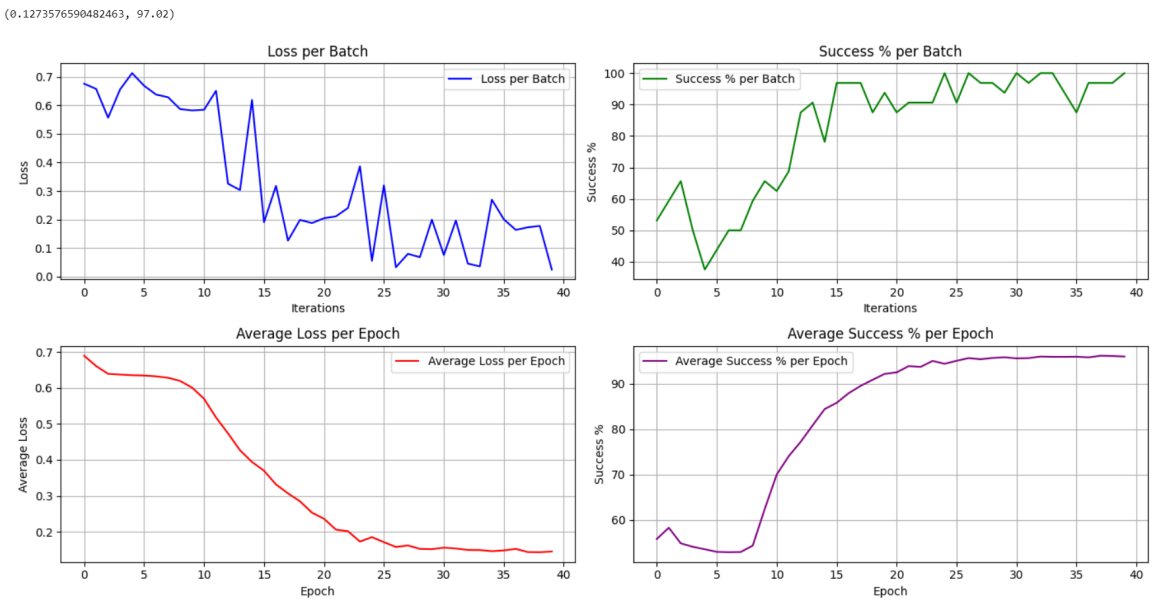

Figure 51: Peak performance achieved with a success rate of 97.0%

To calculate the total number of parameters in our neural network, we multiply the number of neurons in each layer by the number of neurons in the preceding layer to determine the number of weights between them. This calculation is performed for each consecutive layer pair, and the resulting values are then summed to obtain the total parameter count. Our objective was to identify the architecture that delivers the best performance using the fewest epochs, ensuring an efficient training process

Another notable result is :

- **Layer Sizes:** $[2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 2]$.
- **Batch Size:** 64 .
- **Number of Epochs:** 40 .
- **Learning Rate:** 0.09,
- **Momentum:** 0.2.
- **Activation Function:** 'ReLU'.

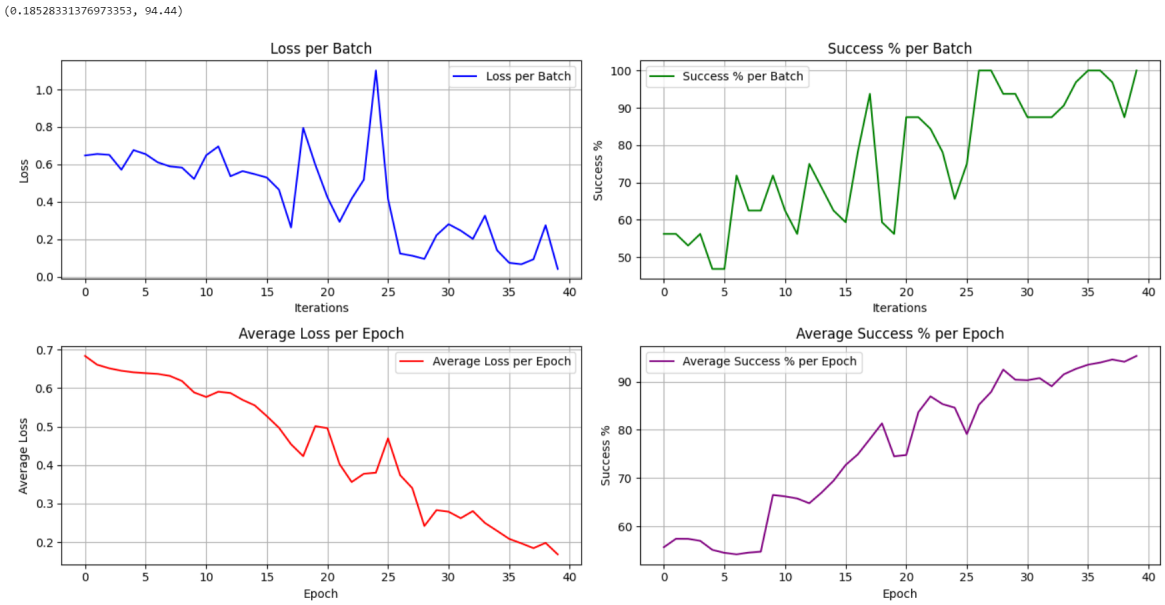

Figure 52: Peak performance achieved with a success rate of 94.4%

9 Part 2.5

9.1 Optimizing Neural Network Efficiency - "GMM"

We conducted the same experiments on the GMM dataset and achieved notable outcomes. Here are the details of the best results :

Optimal Performance Achieved:

- **Layer Sizes:** $[5, 17, 17, 5]$.
- **Batch Size:** 64 .
- **Number of Epochs:** 10 .
- **Learning Rate:** 0.09,
- **Momentum:** 0.2.
- **Activation Function:** 'ReLU'.

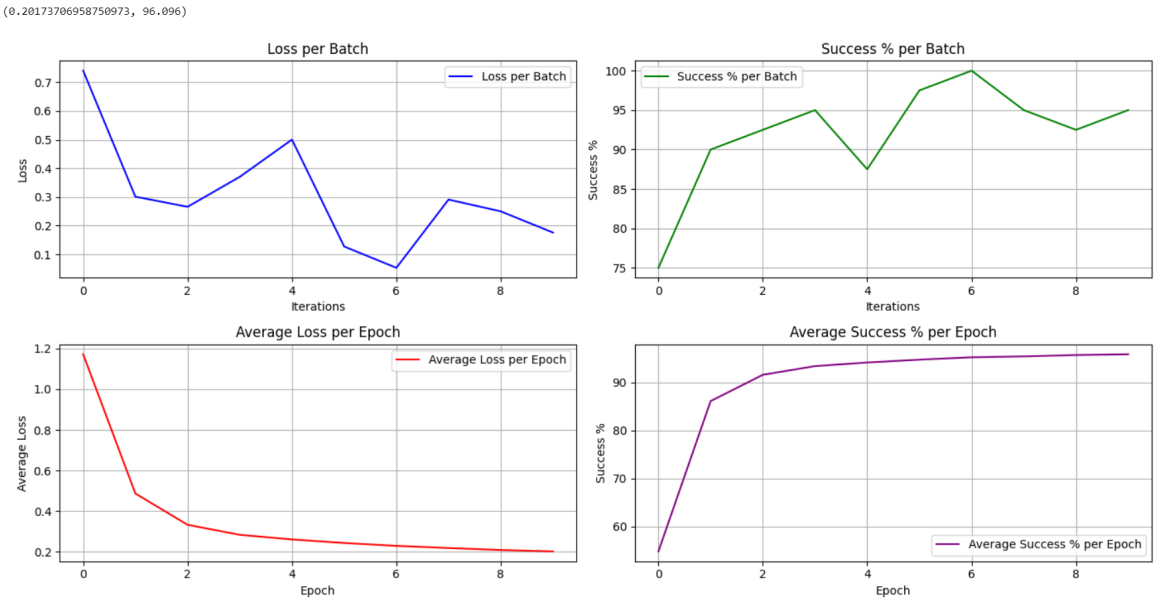

Figure 53: Peak performance achieved with a success rate of 96.0%

Our experiments demonstrated that using the initial settings allowed the network to reach optimal performance within just 10 epochs. Alternative configurations failed to achieve similar success rates and consistently required a significantly higher number of epochs to converge.

Another notable result is :

- **Layer Sizes:** $[5, 13, 13, 13, 5]$.
- **Batch Size:** 64 .
- **Number of Epochs:** 20 .
- **Learning Rate:** 0.09,
- **Momentum:** 0.2.
- **Activation Function:** 'ReLU'.

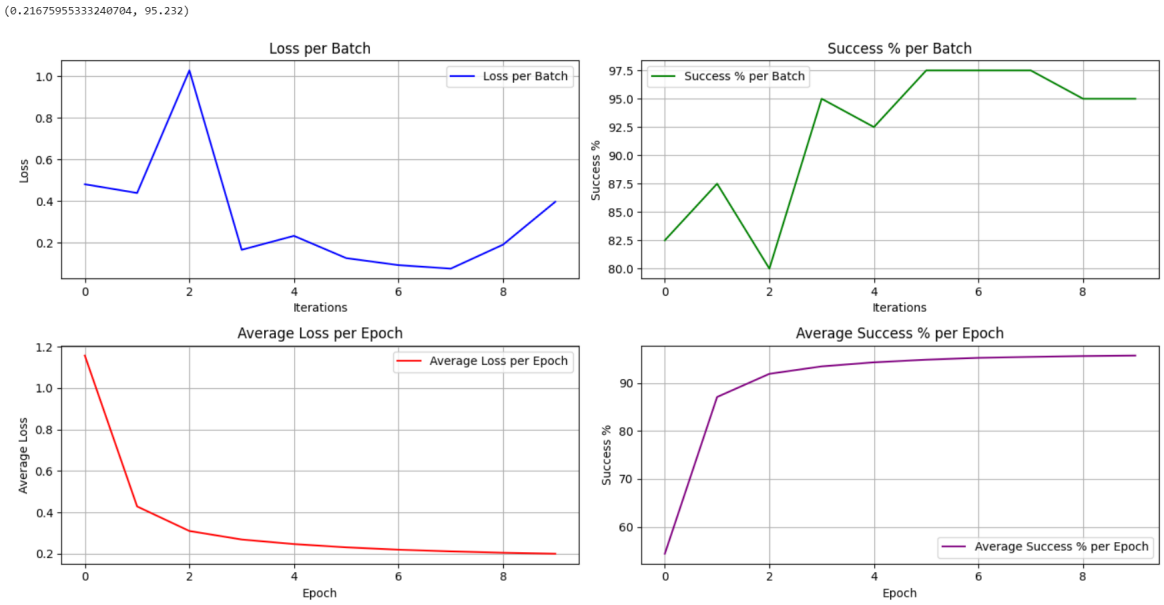

Figure 54: Peak performance achieved with a success rate of 95.2%

10 Code implementation here

Our Code implementation here