Assignment 1

Konstantinos Pasatas (2803568)

November 2024

1 Answers

question 1 The softmax function for the output y_i:

$$y_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$$

Here y_i is the probability for class i, and o_j is the output logit for class i.

We want to compute the derivative of each y_i with respect to each o_j, $\frac{\partial y_i}{\partial o_j}$. In order to do this we should consider 2 different cases.

Case 1:

i=j (the derivative is taken with respect to the logit of the same class).

$$\frac{\partial y_i}{\partial o_j} = \frac{\partial y_i}{\partial o_i} = \frac{\partial \left(\frac{e^{o_i}}{\sum_k e^{o_k}}\right)}{\partial o_i} \tag{1}$$

Let S be the sum of exponentials over all logits:

$$S = \sum_k e^{o_k}$$

Using this, we can rewrite (1) as:

$$\frac{\partial \left(\frac{e^{o_i}}{S}\right)}{\partial o_i} = \frac{\frac{\partial}{\partial o_i}(e^{o_i}) \cdot S - e^{o_i} \cdot \frac{\partial S}{\partial o_i}}{S^2} \tag{2}$$

Now we just have to calculate the derivative of S with respect to o_i.

$$\frac{\partial S}{\partial o_i} = \frac{\partial \sum_k e^{o_k}}{\partial o_i} = \frac{\partial}{\partial o_i}\left(e^{o_1} + \cdots + e^{o_i} + \cdots + e^{o_K}\right) = e^{o_i}$$

Using this, we can rewrite (2) as:

$$\frac{\partial \left(\frac{e^{o_i}}{S}\right)}{\partial o_i} = \frac{e^{o_i} \cdot S - e^{o_i} \cdot e^{o_i}}{S^2} = \frac{e^{o_i} \cdot (S - e^{o_i})}{S^2} = y_i \cdot \frac{(S - e^{o_i})}{S} = y_i \cdot (1 - y_i)$$

So,

$$\frac{\partial y_i}{\partial o_j} = y_i \cdot (1 - y_i)$$

Case 2: $i \neq j$ (the derivative is taken with respect to the logit of a different class).

Here we can adjust (2) as:

$$\frac{\partial \left(\frac{e^{o_i}}{S}\right)}{\partial o_j} = \frac{\frac{\partial}{\partial o_j}(e^{o_i}) \cdot S - e^{o_i} \cdot \frac{\partial S}{\partial o_j}}{S^2} \tag{3}$$

As $i \neq j$ the derivative of e^{o_i} with respect to o_j is 0. So for (3) we have:

$$\frac{\partial \left(\frac{e^{o_i}}{S}\right)}{\partial o_j} = \frac{0 \cdot S - e^{o_i} \cdot e^{o_j}}{S^2} = -\left(\frac{e^{o_i}}{S} \cdot \frac{e^{o_j}}{S}\right) = -y_i \cdot y_j$$

So,

$$\frac{\partial y_i}{\partial o_j} = -y_i \cdot y_j$$

Combining both cases, we get:

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i(1 - y_i), & \text{if } i = j, \\ -y_i y_j, & \text{if } i \neq j \end{cases}$$

Now we will calculate the derivative of cross-entropy with respect to y_j, $\frac{\partial l}{\partial y_j}$.

$$l = \log(y_c)$$

Here y_c is the propability of the correct class c, computed using softmax. The true class is given as an integer, so we can rewrite our function as:

$$\text{loss} = \sum_i l_i$$

$$l_i = \begin{cases} -\log(y_i), & \text{if } c = i, \\ 0, & \text{otherwise} \end{cases}$$

Now we can use this and calculate the derivative with respect to y_i:

$$\frac{\partial l}{\partial y_j} = \begin{cases} -\frac{\partial}{\partial y_j} \log(y_j) = -\frac{1}{y_j}, & \text{if } c = j, \\ \frac{\partial}{\partial y_j}(0) = 0, & \text{otherwise} \end{cases}$$

question 2 We want to compute the derivative of the loss l with respect to each logit o_i including both the correct class logit o_c and the logits for the other classes, $\frac{\partial y_i}{\partial o_i}$. In order to this, we are going to use the Chain Rule.

$$\frac{\partial y_i}{\partial o_i} = \frac{\partial l}{\partial y_c} \cdot \frac{\partial y_c}{\partial o_i} \tag{4}$$

For a neural network, is not necessary to calculate $\frac{\partial y_i}{\partial o_i}$ directly to compute the desired gradient, as backpropagation acts as a middle ground between the symbolic and numeric approaches. Initially, we can compute the derivatives of each part symbolically, and then use the Chain Rule in order to combine these derivatives numerically to obtain the specific gradient we need.

In addition, we talked about Computation Graphs in the lectures, which help us visualize and organize the flow of calculations in a neural network. In practice, the computation graphs break a given complex function into simple parts and allow us to trace the dependencies between variables. As a result, they provide a mean for the application of the Chain Rule for the computation of gradients across layers. With this method, we compute any intermediate derivatives only once and then reuse them for efficiency.

Furthermore, the chain rule in backpropagation allows us to compute the intermediate derivatives one time, and then reuse them as we back propagate through the network. As a result, we avoid recalculating the same derivative multiple times. Using the computation graphs I mentioned above, we can create map for the dependencies and apply the chain rule layer by layer, which makes gradient calculations for each parameter efficient.

In order to compute $\frac{\partial y_i}{\partial o_i}$, we will use the results we found in question 1.

$$\frac{\partial l}{\partial y_j} = \begin{cases} -\frac{1}{y_j}, & \text{if } c = j, \\ 0, & \text{otherwise} \end{cases}$$

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i(1 - y_i), & \text{if } i = j, \\ -y_i y_j, & \text{if } i \neq j \end{cases}$$

Combining these two result, we get:

$$\frac{\partial y_i}{\partial o_j} = \begin{cases} y_i - 1, & \text{if } i = c, \\ y_i, & \text{if } i \neq c \end{cases}$$

question 3 In this section, the implementation of one forward pass and one backward pass of the fully connected Neural Network in Figure 1 is presented.

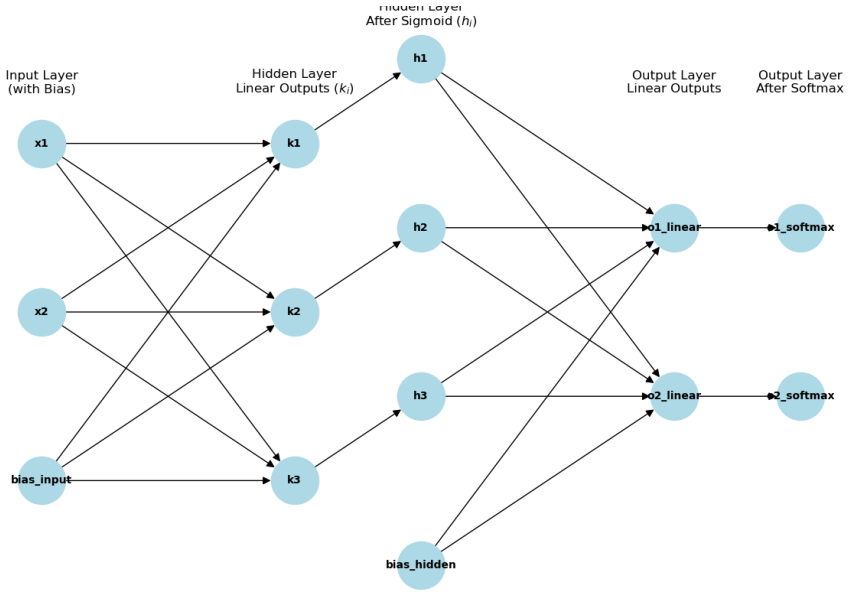

Figure 1: Neural network architecture showing input, hidden, and output layers with Sigmoid and Softmax transformations.

In the following code, we begin with the input vector x = [1., -1.], which represents the values we are going to feed into the network. Additionally, we set the given values for the two sets of weights w and v. The first set contains the weights for the connections between the input layer and the hidden layer. The second set of weights, v, links the hidden layer to the output layer. Furthermore, each layer has biases set to 0.0, with b_in for hidden neurons and b_hidden for output neurons. Finally, the target output is created.

```
# Input
x = [1., -1.]

# Weights
w = [[1., 1., 1.],
     [-1., -1., -1.]]
v = [[1., 1.],
     [-1., -1.],
     [-1., -1.]]

# biases
b_in = [0., 0., 0.]
```

```
b_hidden = [0., 0.]

# target
t = [1, 0]
# Activation functions
def sigmoid(x):
  return 1 / (1 + m.exp(-x))

def softmax(x):
    exp_values = [m.exp(i) for i in x]
    total = sum(exp_values)
    prob = [i / total for i in exp_values]
    return prob
```

The code below implements the forward pass through the network. It calculates the hidden layer weighted sums k and activations h, followed by the output layer weighted sums (o) and final probabilities (y). Finally, the cross-entropy loss is calculated based on the predicted probabilities and the target.

```
# Forward Pass
def forwardpass(x, w, v, b_in, b_hidden, t):
    k = [0.0, 0.0, 0.0]
    h = [0.0, 0.0, 0.0]
    o = [0.0, 0.0]
    y = [0.0, 0.0]

    for j in range(len(k)):
        for i in range(len(x)):
            k[j] += w[i][j] * x[i]
        k[j] += b_in[j]

    for j in range(len(k)):
        h[j] = sigmoid(k[j])

    # Calculate o and y
    for j in range(len(o)):
        for i in range(len(h)):
            o[j] += v[i][j] * h[i]
        o[j] += b_hidden[j]

    softmax_values = softmax(o)
    for j in range(len(o)):
        y[j] = softmax_values[j]

    # Calculate cross-entropy loss
    loss = sum(-t[j] * m.log(y[j]) for j in range(len(y)))
```

```
    return k, h, o, y, loss
```

Variable	Values
Hidden layer weighted sums (**k**)	[2.0, 2.0, 2.0]
Hidden layer activations (**h**)	[0.8808, 0.8808, 0.8808]
Output layer logits (**o**)	[-0.8808, -0.8808]
Output probabilities (**y**)	[0.5, 0.5]
Cross-entropy loss (**loss**)	0.6931

Table 1: Forward Pass Results

The code below, uses backpropagation to compute the gradients of the loss with respect to each weight and bias through the backward_pass function. The function calculates the output layer error terms (delta) and propagates them backward to compute the gradients for the weights (v_prime, w_prime) and biases (b_prime_hidden, b_prime_input).

```
# Backward Pass
def backward_pass(y, t, h, x, v, b_hidden, b_in):
    # Initialize derivatives
    delta = [y_j - t_j for y_j, t_j in zip(y, t)]
    v_prime = [[0.0 for _ in range(len(y))] for _ in range(len(h))]
    w_prime = [[0.0 for _ in range(len(k))] for _ in range(len(x))]
    h_prime = [0.0] * len(h)
    k_prime = [0.0] * len(k)
    b_prime_hidden = [0.0] * len(b_hidden)
    b_prime_input = [0.0] * len(b_in)

    for i in range(len(y)):
        for j in range(len(h)):
            v_prime[j][i] = delta[i] * h[j]
            h_prime[j] += delta[i] * v[j][i]
    for j in range(len(b_hidden)):
        b_prime_hidden[j] = delta[j]
    for i in range(len(k)):
        k_prime[i] = h_prime[i] * h[i] * (1 - h[i])
    for j in range(len(k)):
        for i in range(len(x)):
            w_prime[i][j] = k_prime[j] * x[i]
        b_prime_input[j] = k_prime[j]

    return v_prime, w_prime, b_prime_hidden, b_prime_input
```

Gradient	Values
(**delta**)	[-0.5, 0.5]
(**v_prime**)	[[-0.4404, 0.4404], [-0.4404, 0.4404], [-0.4404, 0.4404]]
(**b_prime_hidden**)	[-0.5, 0.5]
(**h_prime**)	[0.0, 0.0, 0.0]
(**w_prime**)	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
(**b_prime_input**)	[0.0, 0.0, 0.0]

Table 2: Backward Pass Results

question 4 In order to train the neural network using the given data, a few preprocessing steps were needed. Initially, we convert the xtrain type from an np.array to a list, so there are no conflicts with the code for the forward and backward pass. After that, we normalize each feature in training data to be withing the range [0, 1], based on the maximum values of each feature. Following that, we initialize the weights for a neural network with two input units, three hidden units, and two output units. The random weights were generated using a Gaussian distribution with a mean of -1 and standard deviation of 1.

To perform a training loop for the neural network, the two passes from question 3 were used along with stochastic gradient descent. During training, each epoch it one-hot encodes our target variable, calculates the loss within the forwardpass, and then computes the gradient with the backwardpass. Weights and biases are updated using a learning rate of 0.01.

```
for epoch in range(epochs):
    epoch_loss = 0
    for x_instance, target in zip(xtrain, ytrain):
        # One-hot encode the target
        num_classes = 2
        target_one_hot = [1 if i == target else 0 for i in range(num_classes)]
        k, h, o, y, loss = forwardpass(x_instance, w_train,
        v_train, b_in, b_hidden, target_one_hot)
        epoch_loss += loss
        v_prime, w_prime, b_prime_hidden, b_prime_input = backward_pass(y,
        target_one_hot, h, x_instance, v_train, b_hidden, b_in)

        # Updates
        for i in range(len(w_train)):
            for j in range(len(w_train[i])):
                w_train[i][j] -= learning_rate * w_prime[i][j]

        for i in range(len(v_train)):
            for j in range(len(v_train[i])):
                v_train[i][j] -= learning_rate * v_prime[i][j]

        for i in range(len(b_in)):
```

```
        b_in[i] -= learning_rate * b_prime_input[i]

    for i in range(len(b_hidden)):
        b_hidden[i] -= learning_rate * b_prime_hidden[i]
```

Table 3 shows the final weights and biases after training our network for 50 epochs. Figure 2 illustrates a steady decrease over 50 epochs, which indicates that our model adjusts its parameters effectively and it's learning. By the end of the training, the loss converges to zero, which means that the network has minimized the error on the training data.

Variable	**Values**
(**w_train**)	[[-21.1988, -17.1302, 3.6726], [-6.0326, -14.5095, -7.6118]]
(**v_train**)	[[-21.5055, 16.8380], [17.8249, -20.5707], [-16.8816, 17.5766]]
(**b_in**)	[-2.7076, 2.7085, -1.7968]
(**b_hidden**)	[-3.3304, 3.3304]

Table 3: Network Parameters: Weights and Biases

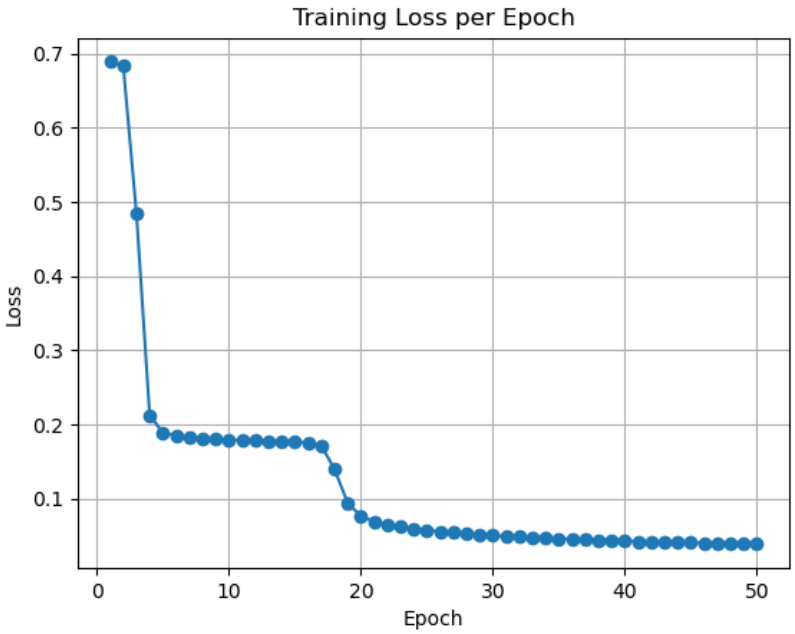

Figure 2: The train loss during training.

question 5 In this question, we want to implement a fully connected neural network using NumPy. In the implementation we are using the vectorized operations, in order to process the data in batches. The new network differs from the one in questions 3 and 4 in the size of inputs (before 2, now 784), hidden layers (before 3 now 300), and number of output nodes (as we want to classify the number from 0 to 9).

As in question 3 and 4, we start by initializing our wights and biases. This time, W has a shape of (784, 300), as we want to connect the 784 input features to the 300 hidden layer units. On the other hand, V has a shape of (300,10), connecting the 300 hidden layer units to the 10 output classes (numbers 0–9). Both sets of weights are initialized with small random values drawn from a normal distribution with mean 0 and standard deviation 0.1. The biases b_in and b_hidden are initialized to zero, with shapes of (1, 300) and (1, 10) respectively.

For the forward pass, we use fully vectorized operations in order to process entire batches of data simultaneously. The linear transformation or the hidden layer (k), is computed as the dot product of input X and W, adding the bias term b_in. Following that, we apply sigmoid activation to (k), element-wise, which results to our new (h). Then, linear transformation for the output layer (o) is applied as before, using the dot product of the activated hidden layer (h) and set of weights (V), adding the bias term b_hidden. Finally, the softmax activation is applied to (o), producing the predicted class probabilities.

```
def forwardpass(X, W, V, b_in, b_hidden):

    k = np.dot(X, W) + b_in
    h = sigmoid(k)

    o = np.dot(h, V) + b_hidden
    y = softmax(o)

    return k, h, o, y
```

For the backward pass, we compute the gradients of loss function with respect to the weights and biases, as we did before. The error at the output layer (delta) is calculated as the difference between the predicted probabilities y and the true labels t. The gradients for the weights in the output layer are obtained by taking the dot product of the transpose of the hidden layer activations (h) and the error term (delta). The gradient for the biases in the hidden layer is calculated by summing the values in delta across all instances in the batch. For the gradient of the hidden layer error, we compute the product of (delta) and the transpose of the output weights (V). Then we multiply this with the derivative of the sigmoid function (h*(1-h)) element-wise. Finally, the gradients for the weights in the hidden layer are computed as the dot product of the transpose of the input data (X) and the hidden layer error (h_prime). The gradients for the biases in the hidden layer are computed by summing the values of (h_prime across all instances in the batch.

```
def backward_pass(y, t, h, X, W, V):
    delta = y - t
    v_prime = np.dot(h.T, delta) / X.shape[0]
    b_prime_hidden = np.sum(delta, axis=0, keepdims=True) / X.shape[0]

    h_prime = np.dot(delta, V.T) * h * (1 - h)

    w_prime = np.dot(X.T, h_prime) / X.shape[0]
    b_prime_input = np.sum(h_prime, axis=0, keepdims=True) / X.shape[0]

    return v_prime, w_prime, b_prime_hidden, b_prime_input
```

In the above code, we divide by X.shape[0], the number of instances in the batch, in order to calculate the average gradient per batch instance. This scaling helps to stabilize training, as it prevents large weights updates when we have large batch sizes. Additionally, averaging the gradients across the batch, it gives us a better estimate for the true gradient direction.

question 6 In this question, we aimed to implement a vectorize approach (numpy matmul) to train the neural network by processing batches of data simultaneously. The first step of the implementation, consists of a batch preparation. We want to make sure that the data fits into batches. For this reason, we must apply padding in case the data size is not divisible by the batch size.

```
pad_size = (num_batches * batch_size) - num_instances
if pad_size > 0:
    xtrain_padded = np.vstack([xtrain, np.zeros((pad_size,
                xtrain.shape[1]))])
    ytrain_one_hot_padded = np.vstack([ytrain_one_hot,
                np.zeros((pad_size, num_classes))])
else:
    xtrain_padded = xtrain
    ytrain_one_hot_padded = ytrain_one_hot

X_batches = xtrain_padded.reshape(num_batches, batch_size, -1)
Y_batches = ytrain_one_hot_padded.reshape(num_batches, batch_size, -1)
```

The next step in our process to adjust the forward pass we created in question 5, in order to be able to feed the network with a batch of images in a single tensor. The adjustments can be seen in the code below.

```
# Forward pass for all batches
k = np.matmul(X_batches, W) + b_in
h = 1 / (1 + np.exp(-k))
o = np.matmul(h, V) + b_hidden
y_pred = np.exp(o - np.max(o, axis=2, keepdims=True))
y_pred /= np.sum(y_pred, axis=2, keepdims=True)
```

Following that, we should calculate the average cross-entropy between the predicted probabilities and the true labels.

```
loss = -np.sum(Y_batches * np.log(y_pred + 1e-8)) / num_instances
```

Additionally, the next adjustment that needs to be made is in the backward pass. In order to do this we will keep the backward pass from question 5 and we will replace the dot product with the matmul operation for tensors. Then as we did before, we backpropagate the error through the hidden layer using the sigmoid derivative. At that point, we us this gradients to adjust the weights and biases. Finally, the weights and biases are updated using gradient descent.

```
# Backward pass
delta = y_pred - Y_batches
```

```
v_prime = np.matmul(h.transpose(0, 2, 1), delta).mean(axis=0)
b_prime_hidden = np.sum(delta, axis=(0, 1)) / num_instances

h_prime = np.matmul(delta, V.T) * h * (1 - h)
w_prime = np.matmul(X_batches.transpose(0, 2, 1), h_prime).mean(axis=0)
b_prime_input = np.sum(h_prime, axis=(0, 1)) / num_instances

# Update weights and biases
W -= learning_rate * w_prime
V -= learning_rate * v_prime
b_in -= learning_rate * b_prime_input.reshape(b_in.shape)
b_hidden -= learning_rate * b_prime_hidden.reshape(b_hidden.shape)
```

Table 4 presents the training and validation loss, along with validation accuracy, for selected epochs. The initial epochs (1-4) show rapid improvements as the model begins learning, while the final epochs (26-30) demonstrate stabilization. However, these results shows that for more epochs the models would have improved. Additionally, in Figure 3 we have a graphical representation of the training and validation loss, along with the accuracy of the model after training for 30 epochs with learning rate 0.01 and batch size 64.

Epoch	**Training Loss**	**Validation Loss**	**Validation Accuracy (%)**
1	1.6005	1.2256	80.66
2	1.5283	1.1966	81.20
3	1.4917	1.1773	81.36
4	1.4668	1.1621	81.52
		...	
26	1.1949	0.9688	83.88
27	1.1866	0.9629	84.00
28	1.1785	0.9572	84.04
29	1.1705	0.9516	84.10
30	1.1628	0.9462	84.26

Table 4: Training and Validation Loss and Accuracy over Selected Epochs

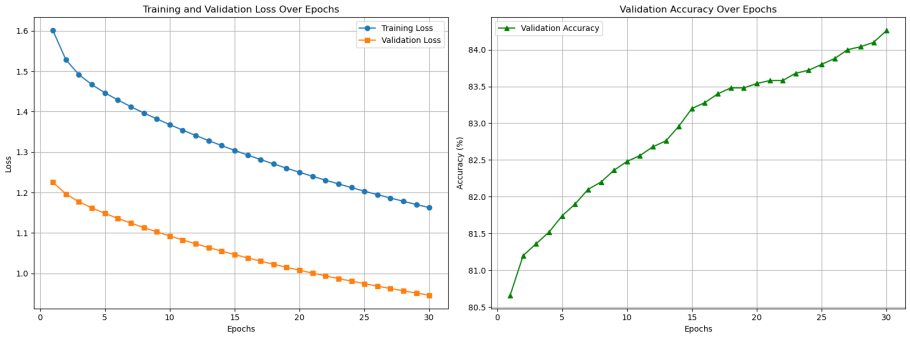

Figure 3: The train/validation loss and accuracy during training.

question 7 In this experiment, we trained the fully connected neural network on the MNIST dataset. We aim to investigate the network's learning behavior under different training conditions, including varying initialization, learning rates, and limited epochs.

The network architecture is consistent with the one we used in the previous two questions. The input layer consist of 784 units, 300 hidden units with sigmoid activation and an output layer of 10 units with softmax activation for the 10 classes. The data loading and preprocessing do not differ from those in the previous question. The only change was the inclusion of test data for generating the final predictions. Additionally, the forward and backward passes remain unchanged from question 6, using the version with np.matmul for tensors. Finally, to ensure a consistent training process and effectively evaluate the differences in the network's behavior, a training function was created. The implementation of this function is shown below.

```
# Training function
def train_model(xtrain, ytrain, xval, yval, W, V, b_in, b_hidden,
learning_rate, epochs, batch_size):
    train_losses, val_losses = [], []
    num_instances = xtrain.shape[0]

    for epoch in range(epochs):
        # Forward pass
        _, h, _, y_pred = forwardpass(xtrain, W, V, b_in, b_hidden)

        train_loss = -np.sum(ytrain * np.log(y_pred + 1e-8)) / num_instances
        train_losses.append(train_loss)

        # Validation forward pass and loss
        _, _, _, y_val_pred = forwardpass(xval, W, V, b_in, b_hidden)
        val_loss = -np.sum(yval * np.log(y_val_pred + 1e-8)) / yval.shape[0]
        val_losses.append(val_loss)
```

```
        # Backward pass
        v_prime, w_prime, b_prime_hidden, b_prime_input = backward_pass(y_pred,
        ytrain, h, xtrain, W, V)

        # Update weights
        W -= learning_rate * w_prime
        V -= learning_rate * v_prime
        b_in -= learning_rate * b_prime_input
        b_hidden -= learning_rate * b_prime_hidden


    return train_losses, val_losses
```

- **Training vs Validation Loss per Epoch** The first experiment was conducted to track the training and validation losses over 5 epochs, aiming to observe the model's learning progression and its generalization ability.

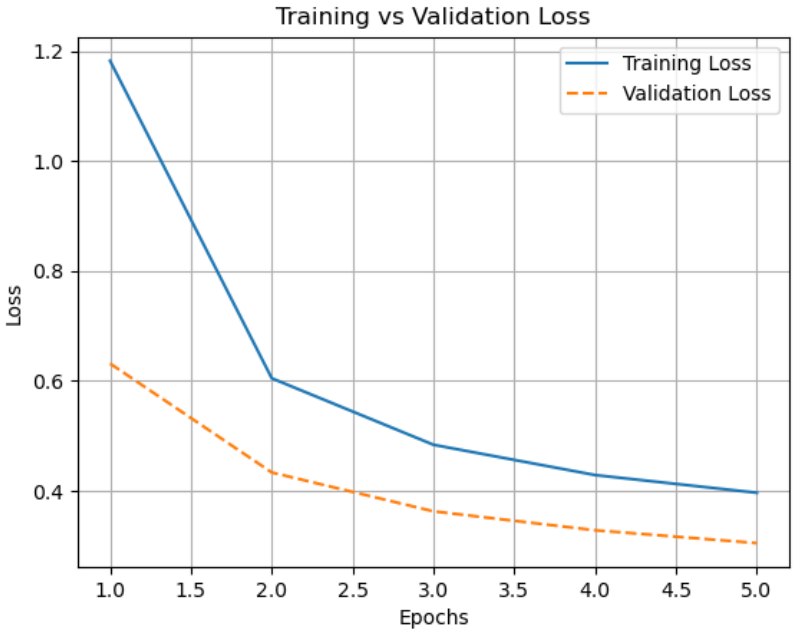

Figure 4: The train/validation loss..

Interestingly, the validation loss is consistently lower than the training loss across all epochs. This suggests that either validation set is easier for the model to classify compared to the train set, or it could imply small variations in data distributions between the sets. Overall, both losses decrease

over epochs, which shows that the models is learning from the data. Additionally, we see a significant drop in loss from epoch 1 to epoch 2, which may happen because the model quickly captures the basic data structure.

- **Random Initialization** The second experiment was conducted to examine the consistency of training loss progression in multiple runs with different random initializations. For this purpose, the model was trained three times with different random initializations for the weights and the biases. For each epoch, we have to calculate the mean of the training losses across the 3 runs. Furthermore, we have to compute the std for each epoch to understand the variation among the different runs.

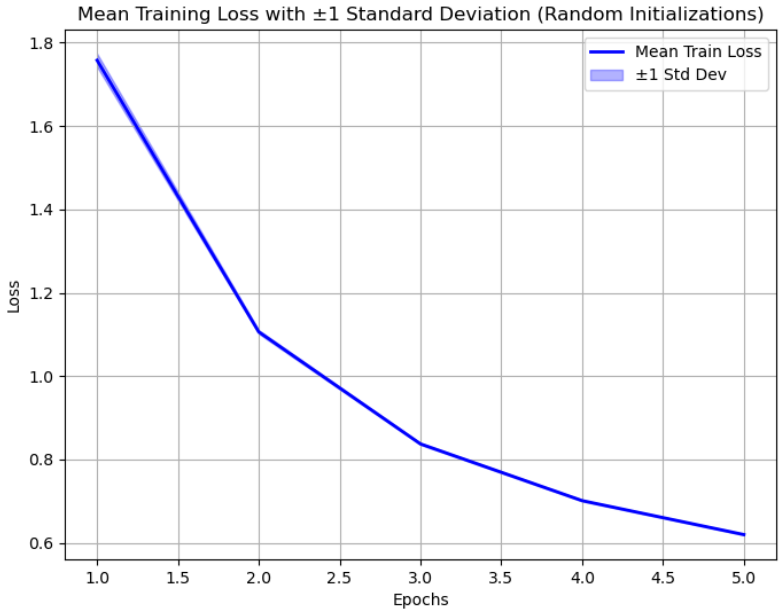

Figure 5: The mean training loss

The plot illustrates a decrease in training loss, which indicates that the model learns, regardless of the initial random weights. Moreover, the narrow std band shows that the models performance is stable with different initialization. To be more precise, the learning trajectory and final loss value do not vary significantly no matter the initialization. The low variability shows that this setup is robust to random initializations, consistently reaching similar training loss levels and boosting confidence in the model's stability.

- **Learning Rate Comparison** The third experiment aimed to investigate

the impact of different learning rates on model performance and act as a hyperparameter tuning process. By rerunning the model with different learning rates, we can determine how the choice of learning rate influences the speed of convergence and the model's ability to generalize. This experiment was conducted by training the model with different learning rates, recording the training and validation losses at each epoch.

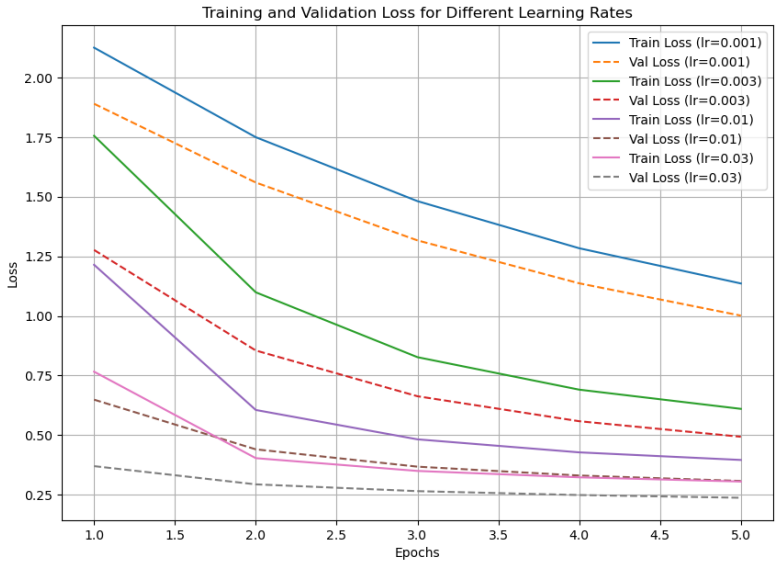

Figure 6: Different Learning Rates

The plot shows a decrease in training and validation losses at varying rates. The smallest decrease was observed with a learning rate of 0.001. A learning rate of 0.003 resulted in faster convergence compared to 0.001, with both training and validation losses decreasing more rapidly. A learning rate of 0.01 exhibited even quicker convergence than the previous two, but the model started to show signs of overfitting, as indicated by the increasing gap between training and validation losses. Finally, a learning rate of 0.03 converged the fastest and achieved the lowest loss values within 5 epochs.

Through the analysis of training and validation losses across different learning rates and multiple random initializations, the best-performing hyperparameters for the model was identified. The model achieved training and validation losses of 0.2965 and 0.2792, respectively in the last run. This indicates a well-trained model with minimal overfitting. The model, trained with learning rate of 0.03, reached a test accuracy of 92.21%, showing strong performance in rec-

ognizing handwritten digits.

Figure 7: Final training and validation loss

The confusion matrix in Figure 8 shows that most digits were correctly classified, with only a few misclassifications. Notably, most of the misclassified digits closely resembled the correct digits in appearance. For instance, the digit 8 was often misclassified as 3 or 5, which suggest that some digits with similar shapes are more challenging for the model to distinguish. This pattern aligns with findings from the exploratory data analysis (EDA) shown in Figure 9 (Appendix), where a PCA projection of the data showed clear clusters for some digits (such as 1s and 3s) but overlap for others (like 3s and 5s).

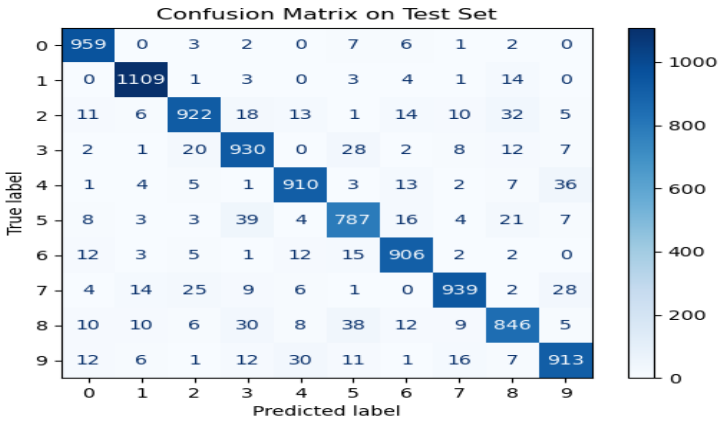

Figure 8: Confusion matrix of model with lr=0.03

A Appendix

Code for data preprocessing in question 4:

```
xtrain = [list(i) for i in xtrain]
X_1 = []
X_2 = []
for i in xtrain:
    X_1.append(i[0])
    X_2.append(i[1])

print(max(X_1), max(X_2))
print(min(X_1), min(X_2))

for i in xtrain:
    i[0] = i[0]/max(X_1)
    i[1] = i[1]/max(X_2)


# Initialize weights
num_inputs = 2
num_hidden_units = 3
num_outputs = 2

w_train = []
for i in range(num_inputs):
    row = []
    for j in range(num_hidden_units):
        weight = random.gauss(-1, 1)
        row.append(weight)
    w_train.append(row)
print(w_train)


v_train = []
for i in range(num_hidden_units):
    row = []
    for j in range(num_outputs):
        v_weights = random.gauss(-1, 1)
        row.append(v_weights)
    v_train.append(row)
print(v_train)
```

Code for data normalizations and initialization of weights and biases in question 5:

```
# Normalization
xtrain = xtrain / 255.0
```

```
xval = xval / 255.0


# Initialization

num_inputs = 784
num_hidden_units = 300
num_outputs = 10

W = np.random.normal(0, 0.1, (num_inputs, num_hidden_units))
b_in = np.zeros((1, num_hidden_units))

V = np.random.normal(0, 0.1, (num_hidden_units, num_outputs))
b_hidden = np.zeros((1, num_outputs))
```

Code for EDA in question 5:

```
import matplotlib.pyplot as plt

# Plot a grid of sample images
num_samples = 4
plt.figure(figsize=(8, 8))
for i in range(num_samples):
    plt.subplot(4, 4, i+1)
    plt.imshow(xtrain[i].reshape(28, 28), cmap='gray')
    plt.title(f"Label: {ytrain[i]}")
    plt.axis('off')
plt.show()

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

# Use PCA or t-SNE for dimensionality reduction
reduced_data = PCA(n_components=2).fit_transform(xtrain[:1000])
plt.scatter(reduced_data[:, 0], reduced_data[:, 1], c=ytrain[:1000], cmap='tab10', s=5)
plt.colorbar()
plt.title("2D Projection of MNIST Data")
plt.savefig('pca')
plt.show()


print("Max pixel value:", xtrain.max())
print("Min pixel value:", xtrain.min())
print("Mean pixel value:", xtrain.mean())
print("Standard deviation of pixel values:", xtrain.std())
```

```python
import matplotlib.pyplot as plt

plt.hist(xtrain.flatten(), bins=50)
plt.title("Pixel Value Distribution")
plt.xlabel("Pixel Intensity")
plt.ylabel("Frequency")
plt.show()
```