Assignment 1

Paul Constantinescu (2830018)

November 2023

1 Answers

Question 1 We derive the gradient of the cross-entropy loss with respect to the softmax inputs for a single example. The cross-entropy loss for a single example is given by:

$$\mathcal{L} = -\sum_{j=1}^{C} t_j \log(y_j)$$

where C is the number of classes, t is a one-hot encoded vector of true labels, and y is the output of the softmax function. The softmax function for class i is:

$$y_i = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}}$$

where z_i is the input to the softmax function for class i. To compute $\frac{\partial \mathcal{L}}{\partial z_i}$, we consider two cases:

- When i is the true class ($t_i = 1$):

$$\frac{\partial \mathcal{L}}{\partial z_i} = y_i - t_i$$

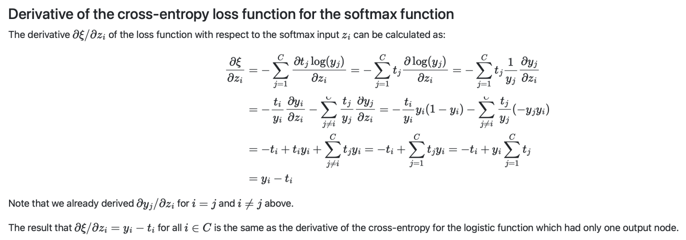
- When i is not the true class ($t_i = 0$):

$$\frac{\partial \mathcal{L}}{\partial z_i} = y_i$$

Combining these, we get:

$$\frac{\partial \mathcal{L}}{\partial z_i} = y_i - t_i$$

This shows that the derivative of the cross-entropy loss with respect to the softmax input is simply the difference between the predicted probability and the truth value.



Derivative of the cross-entropy loss function for the softmax function

The derivative $\partial\xi/\partial z_i$ of the loss function with respect to the softmax input z_i can be calculated as:

$$\begin{aligned}
\frac{\partial\xi}{\partial z_i} &= -\sum_{j=1}^{C}\frac{\partial t_j \log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{\partial \log(y_j)}{\partial z_i} = -\sum_{j=1}^{C} t_j \frac{1}{y_j}\frac{\partial y_j}{\partial z_i} \\
&= -\frac{t_i}{y_i}\frac{\partial y_i}{\partial z_i} - \sum_{j\neq i}^{C}\frac{t_j}{y_j}\frac{\partial y_j}{\partial z_i} = -\frac{t_i}{y_i} y_i(1-y_i) - \sum_{j\neq i}^{C}\frac{t_j}{y_j}(-y_j y_i) \\
&= -t_i + t_i y_i + \sum_{j\neq i}^{C} t_j y_i = -t_i + \sum_{j=1}^{C} t_j y_i = -t_i + y_i \sum_{j=1}^{C} t_j \\
&= y_i - t_i
\end{aligned}$$

Note that we already derived $\partial y_j/\partial z_i$ for $i = j$ and $i \neq j$ above.

The result that $\partial\xi/\partial z_i = y_i - t_i$ for all $i \in C$ is the same as the derivative of the cross-entropy for the logistic function which had only one output node.

Figure 1: The derivative of the cross-entropy loss function for the softmax function.

Question 2 The derivative of the loss $\mathcal{L}$ with respect to an input z_j for non-target classes is part of the gradient computation. However, since the derivative for the target class encapsulates sufficient information for weight updates, this separate derivative is redundant. Contemporary neural network libraries combine softmax and cross-entropy into a single efficient operation, thereby obviating the need for separate calculation of $\frac{\partial\mathcal{L}}{\partial z_j}$ when $j \neq i$.

$$\frac{\partial\mathcal{L}}{\partial z_j} = -\sum_{k=1}^{C} t_k \left(\frac{1}{y_k}\frac{\partial y_k}{\partial z_j}\right)$$

Given that t_i is 1 for the target class and 0 for all other classes, and that y_i depends on all inputs z_k, including z_j, the derivative simplifies to:

$$\frac{\partial\mathcal{L}}{\partial z_j} = y_j$$

Question 3

```
def sigmoid(x):
    return 1 / (1 + math.exp(-x))

def sigmoid_derivative(output):
    return output * (1 - output)

def softmax(x):
    exps = [math.exp(i) for i in x]
    sum_exps = sum(exps)
    return [j / sum_exps for j in exps]

def cross_entropy(output, target):
    loss = -sum(target * math.log(max(output, 1e-15)) for output, target in zip(output,
    ↪  target))
    return loss

def forward_pass(inputs, weights_input_hidden, biases_hidden, weights_hidden_output,
↪   biases_output):
    hidden_layer_input = [sum(input_value * weight for input_value, weight in zip(inputs,
    ↪  weights)) + bias for weights, bias in zip(weights_input_hidden, biases_hidden)]
    hidden_layer_output = [sigmoid(value) for value in hidden_layer_input]
    output_layer_input = [sum(hidden_output * weight for hidden_output, weight in
    ↪  zip(hidden_layer_output, weights)) + bias for weights, bias in
    ↪  zip(weights_hidden_output, biases_output)]
    output_layer_output = softmax(output_layer_input)

    return hidden_layer_input, hidden_layer_output, output_layer_input, output_layer_output

def backward_pass(inputs, hidden_layer_input, hidden_layer_output, output_layer_input,
↪   outputs, target, weights_hidden_output):
    d_loss_d_outputs = [output - target_i for output, target_i in zip(outputs, target)]
    d_weights_hidden_output = [[d_loss * hidden_output for hidden_output in
    ↪  hidden_layer_output] for d_loss in d_loss_d_outputs]
    d_biases_output = d_loss_d_outputs.copy()
    d_loss_d_hidden = [sum(d_loss * weight for d_loss, weight in zip(d_loss_d_outputs, row))
    ↪  for row in zip(*weights_hidden_output)]
    d_hidden_d_inputs = [sigmoid_derivative(value) for value in hidden_layer_input]
    d_weights_input_hidden = [[input_value * d_loss_d_hidden_output * d_hidden_input for
    ↪  input_value in inputs]for d_loss_d_hidden_output, d_hidden_input in
    ↪  zip(d_loss_d_hidden, d_hidden_d_inputs)]
    d_biases_hidden = [d_loss_d_hidden_output * d_hidden_input for d_loss_d_hidden_output,
    ↪  d_hidden_input in zip(d_loss_d_hidden, d_hidden_d_inputs)]

    return d_weights_input_hidden, d_biases_hidden, d_weights_hidden_output, d_biases_output
```

The forward pass computes the linear combination of inputs and weights, adds the bias, and applies the sigmoid activation function for the hidden layer. The output layer receives the activated hidden layer outputs, and the softmax function is then used to predict probabilities. In the backward pass, we calculate the gradients of the loss with respect to the weights and biases by applying the chain rule. These gradients indicate how much a change in each parameter would affect the loss. The 'backward_pass' function calculates these gradients, which are used to update the weights and biases in the direction that reduces the loss.

- Forward pass output: [0.5, 0.5]
- Loss: 0.6931471805599453
- Gradient with respect to weights from input to hidden layer:
  [[-0.0, 0.0], [-0.0, 0.0], [-0.0, 0.0]]
- Gradient with respect to biases in hidden layer:
  [-0.0, -0.0, -0.0]
- Gradient with respect to weights from hidden to output layer:
  [[-0.44039853898894116, -0.44039853898894116, -0.05960146101105877], [0.44039853898894116, 0.44039853898894116, 0.05960146101105877]]
- Gradient with respect to biases in output layer: [-0.5, 0.5]

Question 4

```
def train(x_train, y_train, x_val, y_val, learning_rate, epochs, weights_input_hidden,
↪   weights_output_hidden):
    for epoch in range(epochs):
        total_loss = 0
        for x, y in zip(x_train, y_train):
            target = [1 if y == cls else 0 for cls in range(2)]
            hidden_layer_output, output_layer_output = forward_pass(x, weights_input_hidden,
            ↪   weights_output_hidden)
            weights_input_hidden, weights_output_hidden = backward_pass(
                x, hidden_layer_output, output_layer_output, target, learning_rate,
                weights_input_hidden, weights_output_hidden
            )
            total_loss += cross_entropy(output_layer_output, target)

        tr_accuracy = evaluate(x_train, y_train, weights_input_hidden,
        ↪   weights_output_hidden)
        accuracy = evaluate(x_val, y_val, weights_input_hidden, weights_output_hidden)

        train_losses.append(total_loss / len(y_train))

def normalize_data(data):
    min_val = min(min(row) for row in data)
    max_val = max(max(row) for row in data)
    return [[(2 * (x - min_val) / (max_val - min_val)) - 1 for x in row] for row in data]

(x_train, y_train), (x_val, y_val), num_cls = load_synth()

x_train = normalize_data(x_train)
x_val = normalize_data(x_val)

weights_input_hidden = [[random.gauss(-1, 1) for _ in range(2)] for _ in range(3)]
weights_output_hidden = [[random.gauss(-1, 1) for _ in range(3)] for _ in range(2)]

learning_rate = 0.01
epochs = 10

train(x_train, y_train, x_val, y_val, learning_rate, epochs, weights_input_hidden,
↪   weights_hidden_output)
```

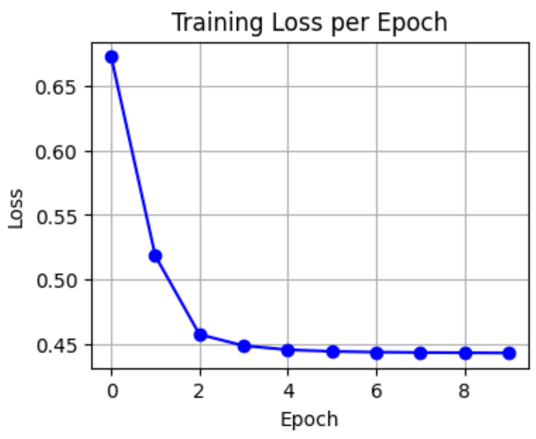


Figure 2: Training Loss per Epoch

Question 5

```
def forward_pass(X, W1, b1, W2, b2):
    Z1 = np.matmul(X, W1) + b1
    A1 = sigmoid(Z1)
    Z2 = np.matmul(A1, W2) + b2
    A2 = softmax(Z2)
    return Z1, A1, Z2, A2

def backward_pass(X, Y, A1, A2, W2):
    m = X.shape[0]
    dZ2 = A2 - Y
    dW2 = np.matmul(A1.T, dZ2) / m
    db2 = np.sum(dZ2, axis=0, keepdims=True) / m
    dA1 = np.matmul(dZ2, W2.T)
    dZ1 = dA1 * sigmoid_derivative(A1)
    dW1 = np.matmul(X.T, dZ1) / m
    db1 = np.sum(dZ1, axis=0, keepdims=True) / m
    return dW1, db1, dW2, db2

def train_and_evaluate(X_train, y_train, X_val, y_val, W1, b1, W2, b2, epochs,
↪   learning_rate):
    training_losses = []
    training_accuracies = []
    validation_accuracies = []

    for epoch in range(epochs):
        Z1, A1, Z2, A2 = forward_pass(X_train, W1, b1, W2, b2)

        train_loss = cross_entropy(A2, y_train)
        training_losses.append(train_loss)

        train_predictions = predict(X_train, W1, b1, W2, b2)
        train_accuracy = get_accuracy(train_predictions, np.argmax(y_train, axis=1))
        training_accuracies.append(train_accuracy)

        dW1, db1, dW2, db2 = backward_pass(X_train, y_train, A1, A2, W2)
        W1, b1, W2, b2 = update_weights(W1, b1, W2, b2, dW1, db1, dW2, db2, learning_rate)

        _, _, _, A2_val = forward_pass(X_val, W1, b1, W2, b2)
        val_predictions = predict(X_val, W1, b1, W2, b2)
        val_accuracy = get_accuracy(val_predictions, np.argmax(y_val, axis=1))
        validation_accuracies.append(val_accuracy)

    return W1, b1, W2, b2, training_losses, training_accuracies, validation_accuracies


(X_train, y_train), (X_test, y_test) , cval = load_mnist()

y_train_one_hot = one_hot_encode(y_train)
y_test_one_hot = one_hot_encode(y_test)

# Xavier initialization - best suited for sigmoid
W1 = np.random.randn(784, 300) * np.sqrt(1. / 784)
b1 = np.zeros((1, 300))
W2 = np.random.randn(300, 10) * np.sqrt(1. / 300)
b2 = np.zeros((1, 10))

learning_rate = 0.1
epochs = 100

W1, b1, W2, b2, training_losses, training_accuracies, validation_accuracies =
↪   train_and_evaluate(
    X_train, y_train_one_hot, X_test, y_test_one_hot, W1, b1, W2, b2, epochs, learning_rate)
```

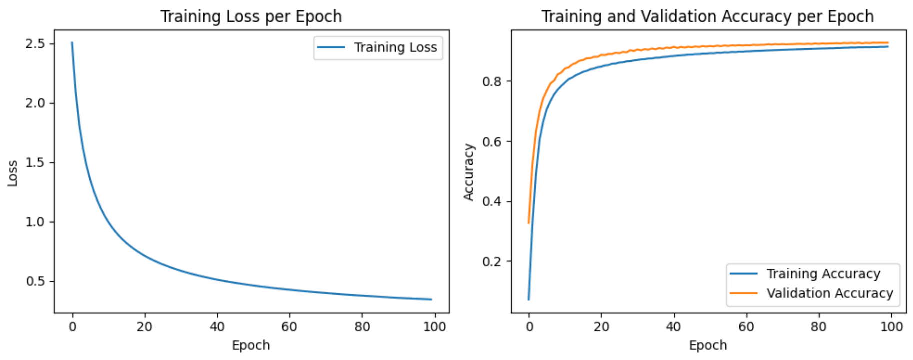


Figure 3: train and val losses and accuracies over 100 epochs

Question 6

```
batch_size = 16

for epoch in range(epochs):
    permutation = np.random.permutation(X_train.shape[0])
    X_train_shuffled = X_train[permutation]
    y_train_shuffled = y_train_one_hot[permutation]

    for i in range(0, X_train.shape[0], batch_size):
        X_batch = X_train_shuffled[i:i + batch_size]
        Y_batch = y_train_shuffled[i:i + batch_size]

        Z1, A1, Z2, A2 = forward_pass(X_batch, W1, b1, W2, b2)
        loss = compute_loss(A2, Y_batch)
        dW1, db1, dW2, db2 = backward_pass(X_batch, Y_batch, A1, A2, W2)
        W1, b1, W2, b2 = update_parameters(W1, b1, W2, b2, dW1, db1, dW2, db2,
        ↪  learning_rate)

    _, _, _, A2_train = forward_pass(X_train, W1, b1, W2, b2)
    train_loss = compute_loss(A2_train, y_train_one_hot)
    train_accuracy = np.mean(np.argmax(A2_train, axis=1) == np.argmax(y_train_one_hot,
    ↪  axis=1))

    _, _, _, A2_val = forward_pass(X_test, W1, b1, W2, b2)
    val_loss = compute_loss(A2_val, y_test_one_hot)
    val_accuracy = np.mean(np.argmax(A2_val, axis=1) == np.argmax(y_test_one_hot, axis=1))
```

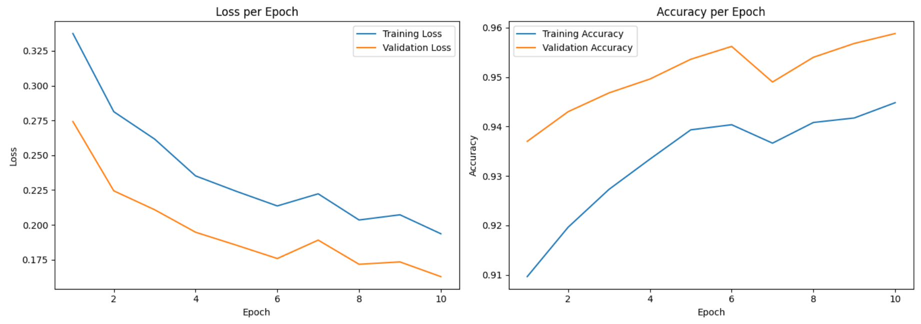


Figure 4: train and val losses and accuracies for batch-size 16

Question 7 The sharp initial decrease indicates rapid learning in the early stages, while the gradual plateauing suggests that the model is approaching its performance limits with the given architecture and hyperparameters. The relatively steady state towards the end of the curve, with minor fluctuations, indicates consistent learning with each batch, and that the model is nearing convergence.

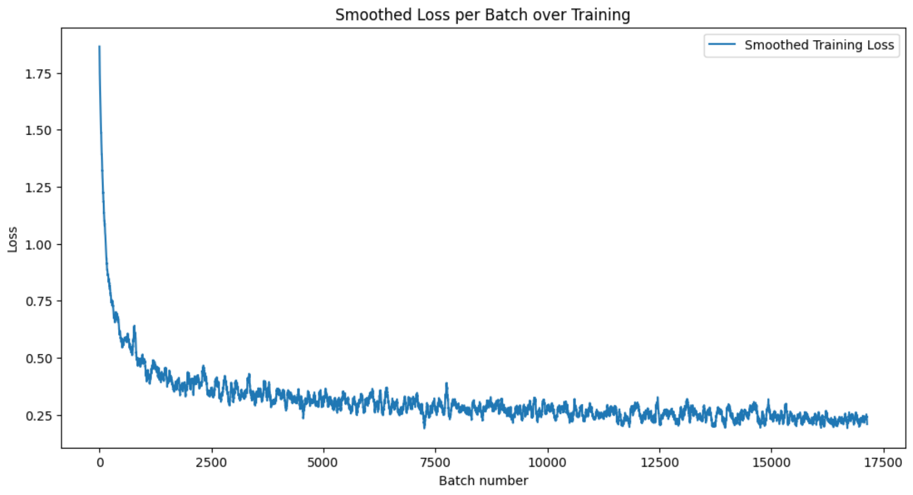


Figure 5: Smoothed Loss per Batch over Training

Point 1 The training and validation losses are very close to each other and both decrease as the number of epochs increases. This similarity suggests that the model is generalizing well and is not overfitting to the training data. Moreover, the fact that both the training and validation metrics improve in tandem is a positive indicator that the model is not memorizing the training data but learning general patterns that also apply to unseen data.

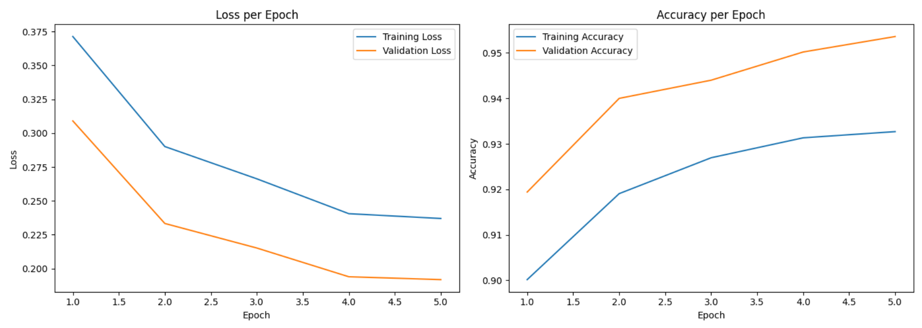


Figure 6: train and val losses and accuracies over 5 epochs

Point 2 We can observe a stable training progression over five epochs, with both training and validation losses diminishing, suggesting effective learning. Notably, the low standard deviation across epochs reflects consistent model performance on different data batches. Similarly, increasing accuracies for training and validation—with training slightly outperforming validation—point to proficient learning without significant overfitting. Early high accuracy suggests a beneficial initialization and possibly a well-processed dataset.

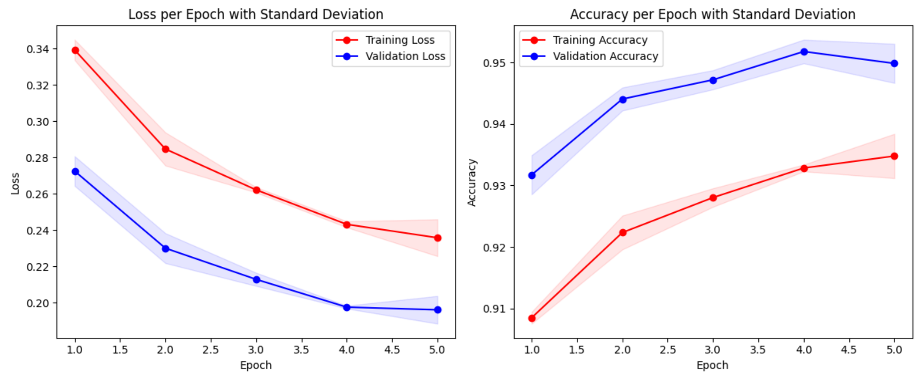


Figure 7: train and val losses and accuracies with Standard Deviation

Point 3 The accuracy trends upward for both training and validation, indicating better prediction capabilities over time. LR=0.003 and LR=0.01 achieve higher accuracy, with LR=0.003 seeming to provide a balance between convergence speed and stability and excelling in validation accuracy by the fifth epoch. Further training could clarify long-term trends and optimal LR selection.

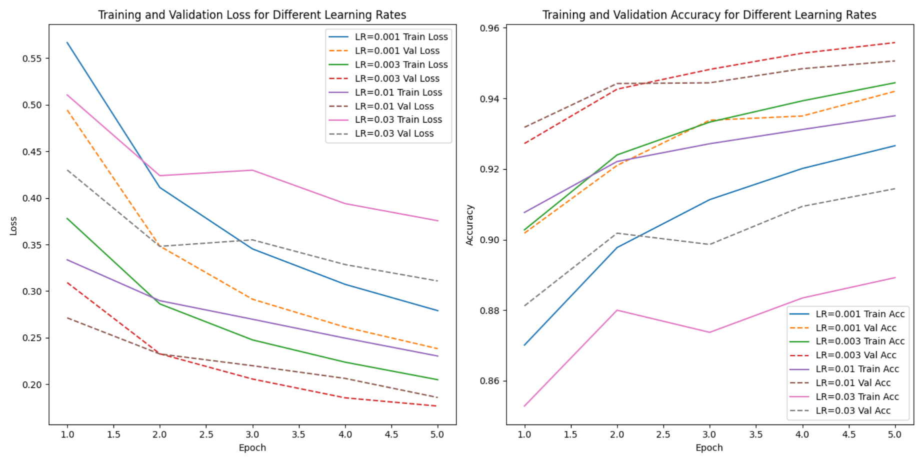


Figure 8: train and val losses for 4 different learning rates

Point 4 Selecting a learning rate of 0.003 and a batch size of 16 yielded an accuracy close to 0.95, being the best results. I also tested different batches for this evaluation, but 16 still turned out to be the best performing.

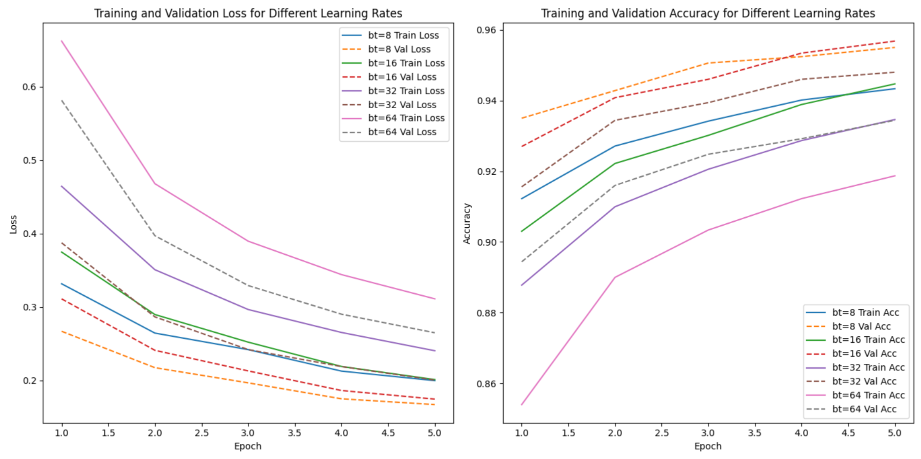


Figure 9: train and val losses for 4 different batch sizes

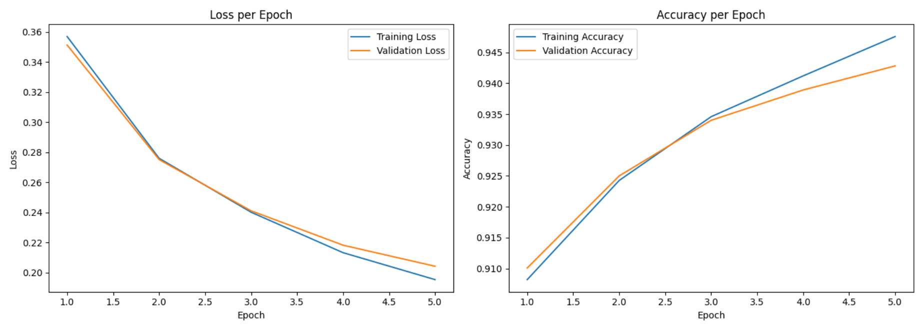


Figure 10: train and val losses for best-performing hyperparameters

A Appendix

For the complete code used in the experiments and analyses presented in this paper, please refer to the following GitHub repository:

`https://github.com/palulconst1/DL_assignment_1/blob/main/DL_assignment_1_final.ipynb`