Project 3 Report

1 Multi-layer Perceptrons

Define the Network (Forward Pass)

The forward propagation function models the given network architecture. The input image x is first multiplied by the first layer of weights W_1 and added to the bias b_1 to obtain the first layer of activations a_1. Then, the sigmoid activation function is applied to a_1 to obtain f_1. Next, f_1 is multiplied by the second layer of weights W_2 and added to the corresponding bias b_2 to obtain the second set of activations a_2. Finally, a softmax activation function is applied to a_2, producing the predicted class probabilities y_{hat}.

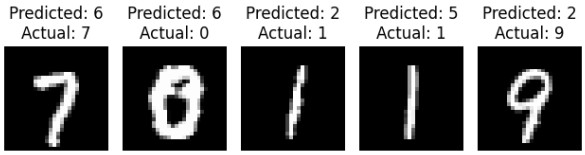

Figure 1: MLP's predictions of the first 5 training images after forward pass

In **Figure 1**, we see that the model's predictions were incorrect for all 5 test images. The predicted labels [6, 6, 2, 5, 2] did not match the actual labels [7, 0, 1, 1, 9]. This suggests that performing only the forward pass is not sufficient to adequately train the neural network. Additional steps, such as back propagation, are necessary to improve the model's performance.

Network Initialization

To initialize the MLP model, weights and biases for both layers are generated according to their respective dimensions. The first layer of weights, W_1, has dimensions $(64, 784)$ and is initialized using a normal distribution with a mean of 0 and a standard deviation of 0.1. The weights are then normalized by dividing each value by the square root of the input size (784) to ensure that the weights have an appropriate scale. The biases for the first layer, b_1, are initialized as an array of zeros with a length of 64.

Similarly, the second layer of weights, W_2, has dimensions $(10, 64)$ and is initialized using a normal distribution with a mean of 0 and a standard deviation of 0.1. These weights are also normalized by dividing each value by the square root of the input size (64) to maintain an appropriate scale. The biases for the second layer, b_2, are initialized as an array of zeros with a length of 10.

Back-propagation

Using the gradients we derived in HW 3, we can easily implement back propagation. However, instead of explicitly forming the sparse matrices we calculated before, I only stored and computed with the non-zero elements of the matrices. For example, I stored dAdW2 as f1 instead of the actual sparse Jacobian matrix. This serves to simplify calculations during backpropagation, as gradients are multiplied element-wise with neuron outputs before activation

functions. In this manner, we can treat the identity matrices as 1. After storing these partial derivatives, I then sets the batch size and L2 regularization lambda (0.0001) for subsequent calculations.
Using the chain rule, the gradients of the loss function with respect to the activation outputs (A_2, A_1), weights (W_2, W_1), and biases (b_2, b_1) are computed. To encourage smaller weight values and prevent overfitting, L2 regularization is applied to the weight gradients (dLdW2 and dLdW1) by adding a scaled version of the weights, scaled by lambda, to the computed gradients.
Finally, the accumulated gradients for weights (W_2, W_1) and biases (b_2, b_1) are updated by adding the computed gradients for y.

Training

The baseline model is trained using stochastic gradient descent with a batch size of 256, employing a cross-entropy loss function and an initial learning rate of 0.001. The network undergoes training for 100 epochs, during which the learning rate is adaptively updated at every epoch by multiplying it with a factor of 0.97. This adaptive learning rate enables the algorithm to converge more effectively, allowing for larger updates at the beginning of the training process when the model's parameters are far from their optimal values and gradually reducing the magnitude of updates as the model approaches the optimum. The training process entails iterating over the training data in batches of size 256. For each batch, the gradients of the loss function concerning the model's parameters are calculated and applied to update the parameters.

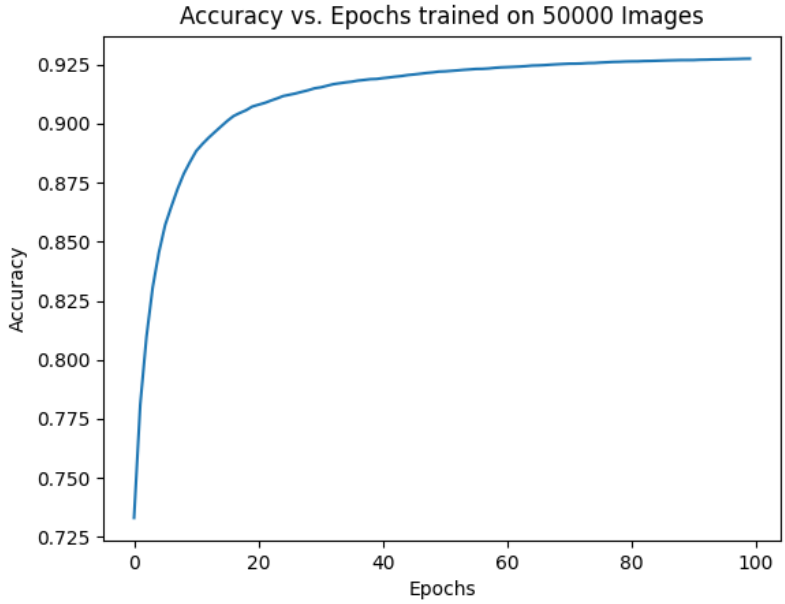

Figure 2: Accuracy of an MLP trained on 50000 images over 100 epochs.

As depicted in **Figure 2**, the model's accuracy increases rapidly as it is trained over more epochs and eventually plateaus at around 0.925. This indicates that the model is effectively learning to recognize patterns in the training data. The leveling off of the accuracy suggests that the model has reached a point where additional training does not significantly enhance its performance on the training data, potentially indicating a good balance between fitting the data and generalizing to new examples. However, there is still considerable room for improvement, such as by fine-tuning various hyperparameters.

Over-fitting?

The network is trained with two different training set sizes (2000 and 50000 images). For both cases, the network is trained using stochastic gradient descent with a batch size of 64, a cross-entropy loss function, and an initial learning rate of 0.01. The batch size and learning rate are tuned to optimize validation accuracy. The training process is executed for 70 epochs, and after each epoch, the accuracy is computed for both the training and validation sets.

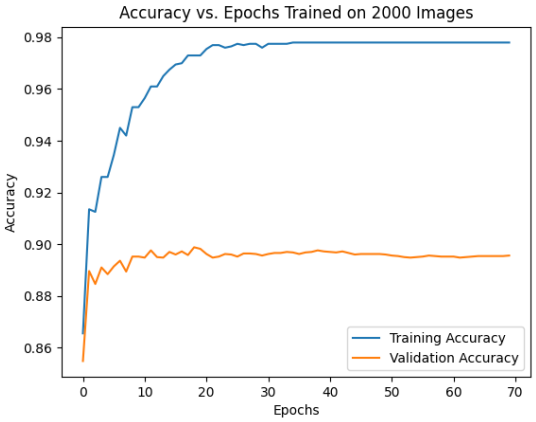

Figure 3: Training and validation accuracies as a function of epochs.

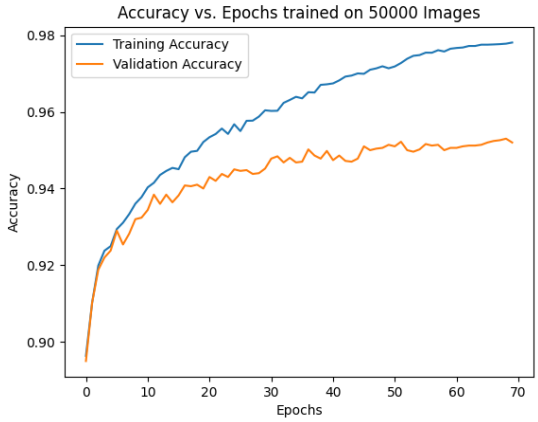

Figure 4: Training and validation accuracies as a function of epochs.

When training the model with the first 2000 images, the training accuracy increases quickly and levels out at approximately 0.98, while the validation accuracy levels out at around 0.88. This relatively large gap between training and validation accuracies suggests that the model overfits the smaller training set.
When training with the full dataset of 50000 images, the training accuracy increases more slowly and eventually levels out at approximately 0.98. However, the validation accuracy is higher, leveling out at around 0.95. The more gradual increase in training accuracy can be attributed to the larger dataset, as the model has more examples to learn from, and thus requires more time to reach a similar level of performance. The smaller gap between training and validation accuracies in this case indicates that the model generalizes better to new data when trained on a larger dataset.
The training times for the two cases are also significantly different. Training with 2000 images takes approximately 5.55 seconds, while training with 50000 images takes around 87.94 seconds. This is expected, as training with a larger dataset requires more computations, increasing the overall training time.

Testing

Using the network trained on all the training images from the previous section, we evaluate its performance on the testing dataset. The forward propagation function is applied to the test images, utilizing the trained weights and biases to generate predicted class probabilities. The class with the highest probability is chosen as the predicted class for each test image. The average accuracy is calculated by comparing the predicted class labels with the true test labels and computing the mean of the correctly classified instances.

To enhance the network's performance, techniques such as L2 regularization, adaptive learning rates, and hyperparameter tuning were employed. As previously mentioned, the learning rate is reduced by a factor of 0.97 at each epoch, facilitating faster convergence initially and more precise weight updates as training progresses. Additionally, to prevent overfitting, L2 regularization is incorporated into the weight updates during training. This introduces a penalty term proportional to the square of the weights, effectively pushing them towards zero. The model's performance can be further optimized by fine-tuning hyperparameters such as the number of epochs, learning rate, and batch size. For instance, the initial choice of 100 epochs was adjusted after observing that the validation accuracy plateaued around 70 epochs. I also experimented with various learning rates, regularization lambda, and batch sizes to improve the model's convergence speed and overall performance. As a result, the network was able to achieve a testing accuracy of 0.9528.

Saving and Loading Weights

The weights and biases of the trained MLP network are saved in "MLP_weights.npz" to avoid retraining the network. If the file doesn't exist, the network is trained from scratch and the weights and biases are saved into the file. If the file exists, it will load the weights (W_1, b_1, W_2, b_2) and assign them to the MLP network's corresponding variables (myNet.W1, myNet.b1, myNet.W2, myNet.b2). Then, it computes the test accuracy using the loaded weights and

prints the result.

Confusion Matrix

We can better understand the model through the confusion matrix. The confusion matrix below presents the true labels of the dataset (rows) against the predicted labels (columns). The diagonal elements represent the percentage of the correct predictions, while the off-diagonal elements show the misclassifications.

		Predicted									
	Classes	0	1	2	3	4	5	6	7	8	9
Actual	0	98.16%	0.00%	0.20%	0.00%	0.20%	0.41%	0.61%	0.00%	0.41%	0.00%
	1	0.17%	95.85%	0.33%	1.33%	0.33%	0.33%	0.33%	0.66%	0.50%	0.17%
	2	0.63%	0.00%	96.86%	0.21%	0.84%	0.21%	0.42%	0.21%	0.42%	0.21%
	3	0.19%	0.19%	0.95%	90.72%	0.00%	3.41%	0.38%	0.95%	2.08%	1.14%
	4	0.00%	0.41%	0.41%	0.00%	96.07%	0.00%	0.83%	0.00%	0.21%	2.07%
	5	0.46%	0.23%	1.39%	1.39%	0.46%	95.14%	0.00%	0.00%	0.69%	0.23%
	6	1.00%	0.60%	0.40%	0.00%	0.80%	0.60%	96.39%	0.00%	0.20%	0.00%
	7	0.58%	0.58%	0.78%	0.00%	0.39%	0.39%	0.00%	96.30%	0.00%	0.97%
	8	0.00%	1.07%	0.43%	1.07%	0.21%	0.86%	0.43%	0.43%	95.06%	0.43%
	9	0.59%	0.20%	0.39%	1.77%	1.77%	0.59%	0.20%	1.38%	0.59%	92.53%

We see that the model performs relatively poorly when identifying 3s (90.72% correct). Specifically, the highest off-diagonal percentage in its row is for column 5. In other words, 3.41% of 3s are being misclassified as 5s. Another high off-diagonal percentage is for column 8 (i.e. 2.08% of the actual 3s are misclassified as 8s). The model also seems to have trouble classifying 9s correctly (92.53% correct).

These misclassifications may be due to similarities in the shapes or features of these numbers. For instance, a poorly written 3 might resemble a 8, and a 9 might have similarities with numbers like 4 depending on the writing style. These similarities can make it difficult for the model to accurately classify these numbers.

Weight Visualization

Visualizing the first layer's weights as templates can provide insights into the patterns that the neural network has learned to detect. By examining these templates, we can better understand the internal workings of the network and the features it uses for classification. These templates represents patterns that the neural network has learned to recognize within input images, such as edges and corners.

The number of templates is determined by extracting the number of rows in the weight matrix W_1. This corresponds to the number of neurons in the first layer, each having learned a different pattern. For each of the 64 neurons, the corresponding template is extracted from W_1 and reshaped into a 28x28 matrix. Each template is then displayed as an image a 64×64 grid.

Figure 5: Visualization of W_1 templates.

This visualization helps the understanding of the features that the network utilizes for classification. For instance, in row 2, column 3, the template resembles the number 2. This suggests that the weights for this neuron are likely used to identify potential 2s by outputting a high value for 2s and a lower value for non-2s. By examining the templates, we can infer which input patterns activate specific neurons in the first layer, providing us with a deeper understanding of the network's decision-making process. Moreover, most templates appear clean and well-defined, indicating that training is proceeding effectively.

2 Convolution Neural Networks

Network Architecture

The Convolution Neural Network starts with a convolutional layer that has 1 input channel, 6 output channels, a 5x5 kernel, no padding, and a stride size of 1. Following this, a 2x2 max-pooling layer with a stride of 2 is applied. The next layer is another convolutional layer with 6 input channels, 16 output channels, a 5x5 kernel, no padding, and a stride size of 1. The same 2x2 max-pooling layer with a stride of 2 is applied again. The resulting feature maps are then flattened and passed through a fully connected layer that maps the 16x4x4 input to 120 output units. A dropout layer with a probability of 0.5 is applied, followed by another fully connected layer that maps the 120 input units to 84 output units. The dropout layer is applied again, and finally, a fully connected layer maps the 84 input units to 10 output units, representing the final classification. In the forward pass, the input tensor is reshaped and passed through each layer with ReLU activation functions applied after the convolutional and fully connected layers.

Training

The baseline model is trained using a batch size of 256, employing a cross-entropy loss function and an initial learning rate of 0.001. The network undergoes training for 100 epochs. Before training, I preprocessed the training, validation, and testing datasets. First, the images are normalized by dividing their pixel values by 255.0, scaling them from a

range of [0, 255] to [0, 1]. Next, the global mean is subtracted from the images, and the result is divided by the global standard deviation. This normalization process ensures that the input data is standardized, which helps the neural network learn more effectively and converge faster.

Once the images are normalized, the image and label datasets are converted from NumPy arrays to PyTorch tensors, with image tensors being of type torch.float32 and label tensors being of type torch.long. Lastly, a test DataLoader is created by combining the test dataset and the corresponding test labels using the TensorDataset function from torch.utils.data. The DataLoader loads the data in batches during the testing phase, with a specified batch size of 256.

To train, the TensorDataset and DataLoader objects are created for the training and validation data, with the specified batch size. The DataLoader feeds the model data in mini-batches. I used the SGD optimizer plus momentum, which helped the model converge faster by adding a fraction of the update vector from the previous time step to the current update vector. A weight decay of 0.0001 is applied, which is the L2 regularization term that helps prevent overfitting.

The training process involves looping over the specified number of epochs. For each epoch, the model processes mini-batches of data, computes the outputs, and calculates the loss. The gradients are computed using backpropagation, and the optimizer updates the model weights accordingly. After each epoch, the training accuracy is computed and appended to a list. If validation is enabled, the validation accuracy is also computed and appended to another list.

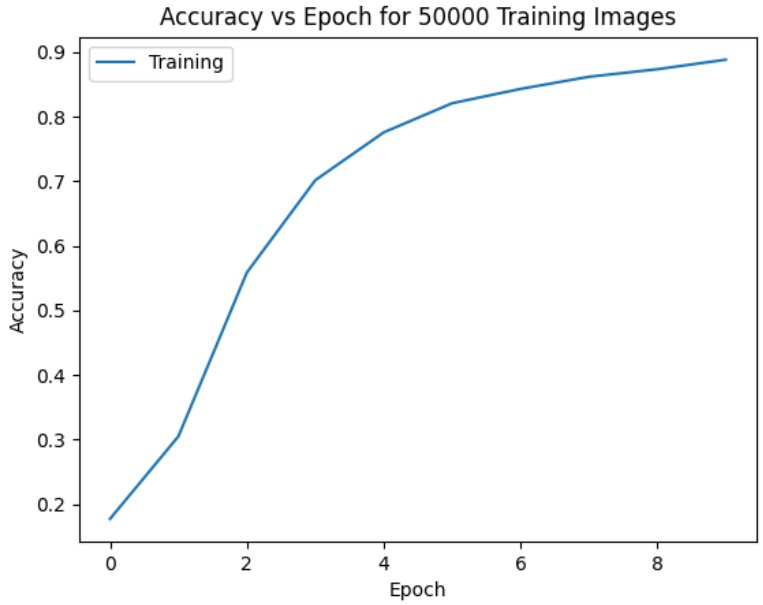

Figure 6: Accuarcy of a CNN trained on 50000 images over 10 epochs

The resulting plot shows a rapid increase in accuracy during the first three epochs, followed by a slower increase and leveling off at around 0.9 accuracy. This pattern suggests that the model is learning quickly in the initial epochs and then converging to a more stable state.

Over-fitting?

The network is trained with two different training set sizes (2000 and 50000 images). For both cases, the network is trained with a batch size of 64, a cross-entropy loss function, and an initial learning rate of 0.01. The training process is executed for 70 epochs, and the accuracy is computed for both the training and validation sets at the end of each epoch.
For the case with 2000 images, the training accuracy increases quickly and levels out at around 0.9, while the validation accuracy levels out at around 0.85. When training with 50,000 images, the training accuracy levels out at around 0.99, but the convergence is slower compared to the 2000-image case. The validation accuracy levels out at 0.98 for the 50,000-image case.

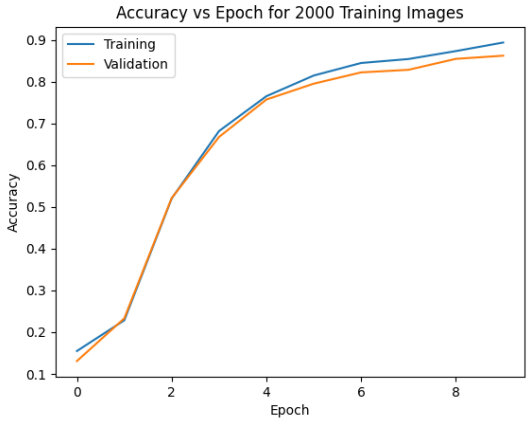

Figure 7: Training and validation accuracies as a function of epochs.

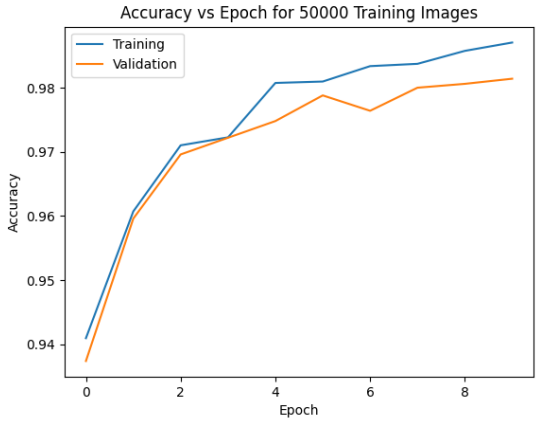

Figure 8: Training and validation accuracies as a function of epochs.

The gap between training and validation accuracy for the 2000-image case is larger than for the 50,000-image case. This is likely because the model trained on a smaller dataset (2000 images) is more prone to overfitting, as it has less diverse data to learn from. As a result, it performs better on the training data but struggles to generalize to the validation data. When trained on the complete trainig dataset, the model has more diverse data to learn from, which improves its generalization ability, leading to smaller gaps between training and validation accuracy. In both cases, the gaps between training and validation accuracies are smaller than for the MLP model. This is likely because CNN is inherently more suited for image processing tasks, as it can learn local spatial features from the images, making it more robust to variations in the data.

The training time for the 50,000-image case is around 87.66 seconds, while it is 6.16 seconds for the 2000-image case. These training times are similar to those observed for the MLP model. This is because the training times for both models are largely determined by the number of images, the number of epochs, and the optimization process. The architecture differences between the MLP and CNN models do not significantly affect the training times in this scenario, as both models use similar optimization techniques and are trained for the same number of epochs.

Testing

Using the CNN trained on all of the training images from the previous section, we evaluate its performance on the testing dataset. The network is first set to evaluation mode. I then iterate through the test data, feeding images to the network, and comparing the output predictions to the actual labels. The average accuracy is determined by computing the mean of the correctly classified instances.
To enhance the network's performance, techniques such as L2 regularization, dropout, and hyperparameter tuning were employed. By including a weight decay term in the optimizer, I was able to reduce overfitting by penalizing large weight values. Furthermore, I implemented dropout to force the network to learn more robust and redundant representations. Tuning various hyperparameters, such as the learning rate, batch size, dropout probability, and weight-decay also helped to improve the network's performance. As a result, the network was able to achieve a testing accuracy of 0.9868.

Saving and Loading Weights

The weights of the trained CNN are saved in "CNN_weights.npz" to avoid retraining the network. If the file doesn't exist, the network is trained from scratch and the weights are saved into the file using the "torch.save()" function. If the file exists, the weights are loaded into a new instance of the ConvNet model using the "load_state_dict()" method. The model is then set to evaluation mode using the "eval()" method to ensure that any regularization layers, such as dropout, are set to their test configurations. The test dataset is then evaluated, and the model's accuracy is calculated and printed.