Project 3

Syed Zaidi

March 2023

1 Multi-layer Perceptrons

1.1 Define the network

1. **MLP:** We have a Multi-Layer Perceptron (MLP) with an input layer, one hidden layer, and an output layer. The input layer has 784 input nodes (corresponding to the size of the flattened grayscale images), the hidden layer has 64 neurons, and the output layer has 10 neurons (corresponding to the 10 classes of MNIST).

2. **Activation Functions:** Use a sigmoid activation function for the hidden layer and a softmax activation function for the output layer. The sigmoid function is defined as:

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

   The softmax function is defined as:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \tag{1}$$

3. **Loss Function:** Use the cross-entropy loss function to measure the difference between the predicted probabilities and the true labels. The cross-entropy loss function is defined as:

$$CrossEntropy(\hat{y}, y) = -\sum_{i=1}^{n} y_i \log(\hat{y}_i)$$

4. **Forward Propagation:** For a given input image, calculate the weighted sum of inputs for the hidden layer, apply the sigmoid activation function, calculate the weighted sum of the hidden layer for the output layer, and finally apply the softmax activation function to obtain the predicted probabilities.

5. **Backpropagation:** Compute the gradients of the loss function with respect to the weights and biases of the output and hidden layers using the chain rule. Update the gradients for each weight and bias in the network.

6. **Parameter Update:** Update the weights and biases using the computed gradients and a learning rate.

1.2 Initialize the Network

To initialize the network weights and biases, we follow these steps:

1. Initialize the network weights with random Gaussian variables with a standard deviation of 0.1.
2. Initialize the biases with zeros.

The weight and bias initialization are implemented in the __init__ method of the class of the MLP.

After initializing the weights and biases, they can be used to build and train the neural network. During the training process, the weights and biases will be updated to minimize the loss function and improve the network's performance on the given task.

1.3 Back-propagation

Here we are asked to complete the update_grad method in the code. This method computes and backpropagates the loss through the network to update the gradients of the weights and biases. The back-propagated gradients are then added to the estimates of the gradients over the entire batch: self.W2_grad, self.b2_grad, self.W1_grad, and self.b1_grad.

The Jacobian of the loss function with respect to the pre-activation values of the output layer a_2 is calculated as follows:

$$\frac{\partial L}{\partial a_2} = [\hat{y}_1 - y_1, \hat{y}_2 - y_2, \ldots, \hat{y}_{10} - y10] \tag{2}$$

where $\hat{y}_i$ denotes the i-th element of the predicted output $\hat{y}$, and y_i denotes the i-th element of the true output y.

Using the Jacobian of the loss function, we can compute the gradients of the weights and biases by backpropagation. We first compute the gradients with respect to the weights and biases in the output layer, and then compute the gradients with respect to the weights and biases in the hidden layer.

With the gradients of the loss function with respect to the pre-activation values of the hidden layer a_1 computed, we can now compute the gradients for the hidden layer weights and biases.

These gradients are then added to the estimates of the gradients over the entire batch:

$$
\begin{aligned}
\text{self.W2\_grad} &\leftarrow \text{self.W2\_grad} + dLdW2 \\
\text{self.b2\_grad} &\leftarrow \text{self.b2\_grad} + dLdb2 \\
\text{self.W1\_grad} &\leftarrow \text{self.W1\_grad} + dLdW1 \\
\text{self.b1\_grad} &\leftarrow \text{self.b1\_grad} + dLdb1
\end{aligned}
$$

This completes the update_grad function, which calculates the gradients of the loss function with respect to the weights and biases in the MLP model. These gradients are then used in the update_params function to update the model parameters using gradient descent.

1.4 Training

1. **One-hot encoding:** The **one_hot_encode** function takes labels and the number of classes as input, and returns a one-hot encoded matrix of the labels. It initializes an empty matrix of shape (number of labels, number of classes) and sets the value to 1 at the index of the true label for each sample.

2. **Accuracy computation:** The **accuracy** function computes the accuracy of the predictions. It takes as input the predicted and true labels, and returns the mean of the comparison between the predicted labels and the true labels.

3. **Training the network:** The `train_network` function is the main function for training the neural network. It takes as input the network object, training and validation datasets, learning rate, number of epochs, and batch size. It performs the following steps:

   (a) One-hot encodes the training labels.

   (b) Iterates through the specified number of epochs.

   (c) Shuffles the training dataset.

   (d) Splits the training dataset into batches.

   (e) For each batch, computes the gradient of the network's weights with respect to the loss function and updates the network's parameters using gradient descent.

   (f) Computes the training and validation accuracies at the end of each epoch and prints them.

4. **myNet.reset_grad()**: This function is used to reset the gradients of the network parameters to zero. This is necessary at the beginning of each mini-batch iteration, as gradients accumulate over multiple data points during backpropagation.

5. **myNet.forward(x)**: This function is responsible for the forward pass of the neural network. It takes an input x (an image in this case) and computes the output of the network $\hat{y}$. The forward pass is done by applying

a series of transformations (e.g., linear layers, activation functions) to the input.

6. **myNet.update_grad(y)**: This function is used to update the gradients of the network parameters using the true label y and the predicted label $\hat{y}$ from the forward pass. It computes the gradient of the loss with respect to each parameter using the chain rule and adds it to the current gradient value.

7. **myNet.update_params(lr)**: This function updates the network parameters using the accumulated gradients and the given learning rate lr. The update is done using gradient descent

8. **Setting hyperparameters:** The learning rate, number of epochs, and batch size are set with the following lines of code:

$$\begin{aligned} \texttt{learning\_rate} &= 1e-3 \\ \texttt{n\_epochs} &= 100 \\ \texttt{batch\_size} &= 256 \end{aligned}$$

   These hyperparameters control the learning process and can be tuned to improve the performance of the neural network.

9. **Training the network:** Finally, the `train_network` function is called with the created instance of the neural network, the training and validation datasets, and the specified hyperparameters. The function trains the network and prints the training and validation accuracies at the end of each epoch.

1.5 Over-fitting?

The training and validation accuracy are computed and printed for each epoch inside the train_network function. Which allows us to monitor the training process and validate the performance of the network. (see figure1)

According to the figures, we can see that when the training was performed using only the first 2000 images, the training and validation accuracies were very close to each other, indicating that there was no overfitting to the training data. However, the accuracy was lower compared to training on the whole dataset. On the other hand, when training on the entire dataset, the training accuracy increased, which suggests some overfitting, as the accuracy on the validation set did not reach as high a percentage. Nevertheless, the difference is not drastic enough to indicate the need for intervention.

In cases where there is a more substantial difference between the training and validation accuracies, measures to reduce overfitting, such as regularization, can be employed.

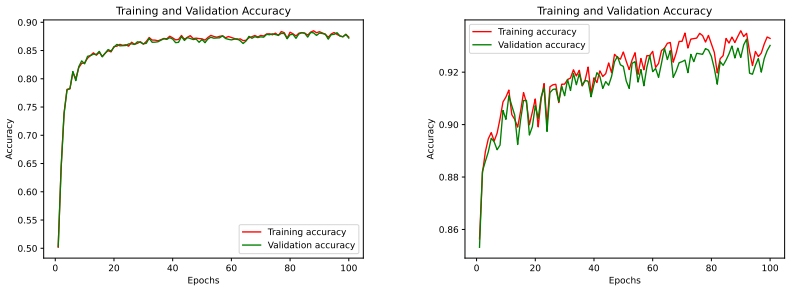

Figure 1: Figures show the training accuracy when training over 2000 images (Left) and 50,000 images (Right) and validated over the whole validation set.

1.6 How well does it work?

We use the function named test_network which is defined to evaluate the accuracy of a neural network on the test set of images and labels. The function takes three arguments: network, which represents the trained neural network, test_images, a set of images to test the network on, and test_labels, the corresponding labels for the test images.

Inside the function, a variable correct_predictions is initialized to keep track of the number of correctly classified images. The total number of images in the test set is calculated and stored in total_predictions. A for loop is then used to iterate over all the images in the test set. For each image, the input x and corresponding label y are retrieved from the test set.

Next, the neural network is applied to the input image x using the forward method of the network object. This produces a predicted output $\hat{y}$, which is a probability distribution over the possible classes. The predicted class is then computed as the index of the highest probability using np.argmax function. If the predicted class matches the true class y, then the correct_predictions count is incremented.

After iterating over all the test images, the accuracy of the neural network is calculated as the ratio of the correct_predictions to the total_predictions, multiplied by 100 to obtain a percentage. Finally, the function returns the accuracy value.

The accuracy achieved on the testing dataset is 90.66% for the saved weights.

1.7 Saving and loading weights

In the code, two functions save_weights and load_weights are defined to respectively save and load the weights of a neural network.

The save_weights function takes two arguments: network, which is the neural network object whose weights are to be saved, and filename_prefix, which is a string specifying the prefix to be used for the filenames of the saved weight files. The function uses the np.save function to save the weights of the neural network's layers to separate NumPy binary files with names constructed using the specified filename prefix and a layer-specific suffix. Specifically, the weights of the first and second layers of the network (W1, b1, W2, and b2) are saved.

The load_weights function takes two arguments: network, which is a new neural network object into which the saved weights are to be loaded, and filename_prefix, which is the same prefix used to save the weights in the save_weights function. The function uses the np.load function to load the saved weights from the NumPy binary files into the corresponding attributes of the network object.

1.8 Where does it make mistakes?

According to the the confusion matrix 2, the numbers that are more difficult to identify are (just as an example):

- Digit 5: has the most misclassification, mostly missclassified as digit 3
  Digit 8: has has also a high misclassification, mostly missclassified as digits 3 and 9.

These conclusions are based on the higher off-diagonal values in the confusion matrix, which indicate more misclassifications for those digits.

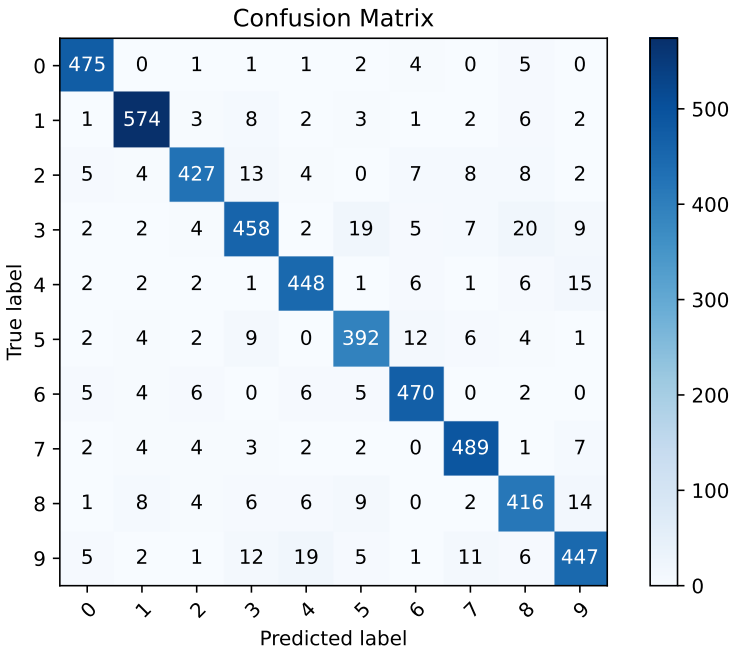

Figure 2: matrix depicting the classification performance of the neural network on the test dataset. The matrix highlights the number of correct predictions along the diagonal and the misclassifications for each digit as off-diagonal elements.

1.9 Visualize the weights

We make a which creates a grid of subplots, reshapes each row of W1 into a 28x28 image, and displays it using a grayscale colormap. These visualizations represent the templates that the network has learned to match in the input images.

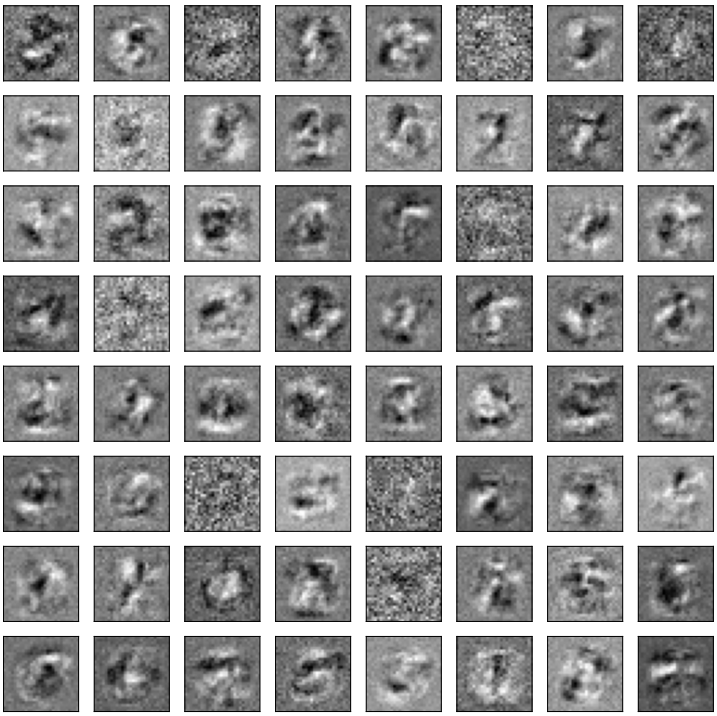

Figure 3: Visualization of the first layer's weight templates in the neural network, representing learned features and patterns in handwritten digits. Each template captures distinct aspects, such as edges and simple shapes, aiding in digit recognition.

When analyzing these templates, we notice some patterns that resemble parts of handwritten digits. These templates allow the network to identify features in the input images that help it distinguish between different classes (digits). However, the templates might not be perfectly interpretable, as the network learns complex and abstract representations that are not always visually meaningful to humans.

2 Convolutional Neural Networks

2.1 Describe the network architecture

The network consists of two convolutional layers and three fully connected layers. The first convolutional layer has 1 input channel and 6 output channels, with a kernel size of 5x5, stride of 1 and no padding. The second convolutional layer has 6 input channels and 16 output channels, with a kernel size of 5x5, stride of 1 and no padding. Both convolutional layers are followed by a max pooling layer with a kernel size of 2x2 and stride of 2.

After the convolutional layers, the output is flattened and fed into three fully connected layers with ReLU activations. The first fully connected layer has 16x4x4 (where 4 is the height and width of the output after max pooling from the second convolutional layer) input neurons and 120 output neurons, the second fully connected layer has 120 input neurons and 84 output neurons, and the third fully connected layer has 84 input neurons and 10 output neurons (corresponding to the 10 classes in the MNIST dataset).

2.2 Train CNN

The trainNN function takes five arguments: network, optimizer, epochs, criterion, and an optional full_data flag. The function trains the given neural network for a specified number of epochs using the provided optimizer and loss function.

The training process will use GPU acceleration, if available, and iterates through the dataset in mini-batches. The optimizer's gradients are zeroed before each forward and backward pass. The loss is calculated, and the optimizer updates the network's weights accordingly. The function prints the running loss for every epoch, allowing the user to monitor the training progress.

So we initialize two instances of the neural network, myNet_full and myNet_mini, and sets up the cross-entropy loss function and the stochastic gradient descent (SGD) optimizer for each network. The networks are then trained using the trainNN function, with myNet_mini training on the first 2000 images of the data (indicated by the full_data = False flag), while myNet_full trains on the entire dataset.

2.3 Over-fitting?

We can see a similar result in this section as the previous one, enforcing our claim that when training over the whole training set can over-fit the network and it reaches 100% accuracy on the training set in the first 50 epoch

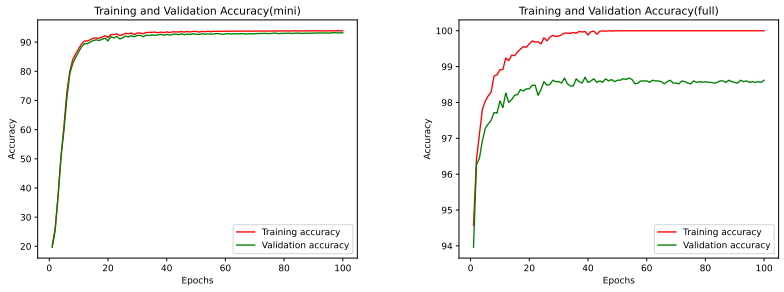

Figure 4: Figures show the training accuracy when training over 2000 images (Left) and 50,000 images (Right) and validated over the whole validation set.

2.4 How well does it work?

In the testing part of the code, the trained PyTorch model myNet_full is tested on a dataset of test images. First, the test images are converted into a tensor and reshaped to match the input shape expected by the model (batch size, channels, height, width). The tensor is then moved to the appropriate device (CPU or GPU) using the to() method. Next, the model is used to make predictions by passing the test inputs through a forward pass, generating test_outputs. These outputs are then processed using the argmax() function to extract the predicted class labels along the first dimension. The predictions are moved back to the CPU and converted to a NumPy array for further processing. The test accuracy is calculated by comparing the predicted class labels to the ground truth labels (test_labels_np) and computing the mean of the correct predictions. Finally, the test accuracy is printed as a percentage with a precision of two decimal places.

The accuracy achieved on the testing dataset is 98.72% for the saved weights.

2.5 Saving and loading weights

In this part of the code, the trained model myNet_full is saved and subsequently loaded for further use. First, the model's state dictionary, which contains its learnable parameters (weights and biases), is saved to a file named 'myNet_full.pth' using torch.save(). This approach is recommended as it is more flexible and less prone to errors. To load the model, a new instance of the Net class, named myNet_full_loaded, is created. The saved state dictionary is then loaded from the 'myNet_full.pth' file using torch.load() and applied to the new model instance with the load_state_dict() method. Finally, the loaded model is set to evaluation mode using the eval() method. This is an important step, as it affects the behavior of certain layers like BatchNorm and Dropout during inference, ensuring the model behaves correctly when making predictions.