EE5179:Deep Learning for Imaging Programming Assignment 1: MNIST Classification using MLP

Janani D, ns24z460

Introduction

This report details the implementation of a neural network from scratch using Python and NumPy, applied to the MNIST dataset. The key components include data loading, one-hot encoding, parameter initialization, forward propagation, backward propagation, and various activation functions. The major objective of this report is to understand the functions, parameters and their uses in the code written for the complete implementation of multi layered perceptron.

Importing Libraries

- **numpy (`np`)**: A powerful numerical computing library used for working with arrays and performing mathematical operations efficiently.
- **pandas (`pd`)**: A data manipulation and analysis library that provides data structures like DataFrames to handle structured data effectively.
- **matplotlib.pyplot (`plt`)**: A plotting library used for creating static, interactive, and animated visualizations in Python.
- **warnings**: A module to control the display of warnings during code execution. Here, warnings are suppressed using `warnings.filterwarnings('ignore')`.
- **torch**: A popular deep learning framework, PyTorch, used for building and training neural networks.
- **torchvision**: A PyTorch package containing popular datasets, model architectures, and image transformations for computer vision tasks.
- **torchvision.transforms (`transforms`)**: A module within `torchvision` for data augmentation and preprocessing of images.
- **datasets**: A submodule of `torchvision` that provides access to various common datasets used in machine learning.
- **torch.utils.data.DataLoader**: A PyTorch utility that provides an efficient way to load and iterate over datasets.
- **random_split**: A function from `torch.utils.data` to split datasets into random subsets, useful for creating train/validation/test splits.
- **TensorDataset**: A dataset wrapper in PyTorch that combines tensors into a dataset object.
- **torchvision.transforms.ToTensor (`ToTensor`)**: A specific transformation that converts images or arrays to PyTorch tensors.

Dataset Loading and Parameter Initializing

The MNIST dataset, consisting of handwritten digits from 0 to 9, is loaded using PyTorch's `datasets.MNIST` class. The dataset is split into training, validation, and test sets for training and testing respectively.

One Hot Encoding

This code defines a function `one_hot_encode` that converts a given 1D numpy array (`matrix`) into a one-hot encoded matrix. It first identifies the unique values in the array and assigns each a unique index. Then, it creates a zero matrix with rows equal to the array size and columns equal to the number of unique values and setting the appropriate position to 1 for each element based on the value's index.

Image Resizing

The function `resize_and_flatten` takes a collection of images (`data`) and resizes each image to a new size of 28x28 pixels and flattens each resized image into a 1D array. This is used to convert the 28x28 pixel images into a flat array of 784 elements (28x28=784).

Data Preprocessing

This code loads the MNIST dataset, splitting it into training, validation, and test sets. The `train_dataset` and `test_dataset` are downloaded and transformed into numpy arrays. The training set is split into a smaller training set and a validation set using `random_split`. Data loaders (`DataLoader`) are then created for batching and shuffling these datasets.

The code fetches batches from the loaders, converting images and labels into numpy arrays (`X_train`, `y_train`, etc.). The labels are one-hot encoded using the `one_hot_encode` function. Finally, the images (`X_train`, `X_val`, `X_test`) are resized to 28x28 and flattened to 1D arrays using the `resize_and_flatten` function.

Parameter Initialization

This function, `glorot_init`, initializes the weights and biases of a neural network using the Glorot (Xavier) initialization method, which helps to maintain the variance of activations and gradients across layers for better training. The Glorot initialization sets the weights with values drawn from a uniform distribution between -M and M, where $M = \sqrt{\frac{6}{N_i+N_o}}$. Here, N_i is the number of input units, and N_o is the number of output units of a layer.

Glorot Initialization Formula:

$$M = \sqrt{\frac{6}{N_i + N_o}} \tag{1}$$

The function creates a dictionary `parameters` that stores the weights (`W`) and biases (`b`) for each layer of the network. It initializes weights for the input layer, hidden layers, and the output layer using the Glorot distribution formula for each layer's size based on the number of neurons specified in the `neurons` list and the input/output dimensions.

Functions used in forward propogation and backpropogation

1. **Activation Functions**:
   - `sigmoid(x)`: Uses the logistic function from `scipy` (`expit`) to compute the sigmoid of `x`.
   - `relu(x)`: Applies the ReLU function, returning the maximum of 0 and `x`.
   - `tanh(x)`: Applies the hyperbolic tangent function.
   - `softmax(x)`: Computes the softmax of `x`, normalizing values to a probability distribution.
2. **Derivatives**:
   - `dsigmoid(a)`: Computes the derivative of the sigmoid function.
   - `drelu(x)`: Computes the derivative of the ReLU function, returning 1 for positive inputs and 0 otherwise.
   - `dtanh(x)`: Computes the derivative of the tanh function.
3. **Gradients**:
   - `gradient_w(x, w, b, y)`: Calculates the gradient of the loss with respect to weights using the sigmoid function.
   - `gradient_b(x, w, b, y)`: Computes the gradient of the loss with respect to biases.
4. **Loss Function**:
   - `cross_entropy(y_p, y_t)`: Computes the cross-entropy loss between predicted (`y_p`) and true labels (`y_t`).
5. **Utility Function**:
   - `average_every_200_elements(data)`: Calculates the average of every 200 elements in the given list `data`.

Forward Propogation

Input Parameters

- **`x`**: A numpy array representing the input data to the neural network.
- **`parameter`**: A dictionary containing the weights (`W`) and biases (`b`) for each layer of the network. These parameters are used to calculate the activations at each layer.
- **`h`**: An integer representing the total number of layers in the neural network (including the input and output layers).
- **`g_act`**: The activation function to be applied to the hidden layers of the network. It is typically a function like ReLU, sigmoid, or tanh.

Return type

- **`activation_output`**: A dictionary that contains the activations (`h`) and pre-activations (`a`) for each layer of the network. The keys are named based on the layer index (e.g., `'a1'`, `'h1'`, etc.), and the values are the corresponding numpy arrays of pre-activations and activations. This dictionary provides the results of the forward propagation step through the network.

Backpropogation

Input Parameters

- **`activation_output`**: A dictionary containing the activations (`h`) and pre-activations (`a`) from each layer of the network during forward propagation. These are used to calculate the gradients during backpropagation.
- **`parameters`**: A dictionary containing the weights (`W`) and biases (`b`) of each layer in the neural network.
- **`y`**: A numpy array representing the true labels or target values for the input data. It is used to compute the loss gradient with respect to the output of the network.
- **`k`**: An integer representing the total number of layers in the network (including the input layer but excluding the output layer).
- **`cl`**: Stands for the number of classes in the classification task.
- **`d_act`**: The derivative of the activation function (`g_act`) used in the hidden layers. This function is applied to compute the gradient of the loss with respect to the pre-activations in each hidden layer.

Return type

- **`gradients`**: A dictionary containing the gradients of the loss with respect to the weights (`dW`) and biases (`db`) for each layer. The keys are formatted as `dW1`, `dW2`, ..., `db1`, `db2`, ..., corresponding to each layer's weights and biases. These gradients are used to update the network parameters during training (e.g., via gradient descent).

Gradient Descent

Input Parameters:

- **`parameters`**: A dictionary containing the current weights (`W`) and biases (`b`) of the neural network. These are the parameters that will be updated using the gradients.
- **`losses`**: A dictionary containing the gradients of the loss with respect to the weights (`dW`) and biases (`db`) for each layer, as calculated by the backpropagation function. These gradients are used to update the parameters.
- **`eta`**: A float representing the learning rate. This is the step size used in the parameter update rule, controlling how much the parameters are adjusted in each iteration.

Return Type:

- **`new_parameters`**: A dictionary containing the updated weights (`W`) and biases (`b`) for each layer of the neural network. The updates are performed using gradient descent, where each parameter is adjusted by subtracting the product of the learning rate (`eta`) and the corresponding gradient from the `losses` dictionary. The `new_parameters` dictionary is the modified version of the input `parameters`, ready for the next iteration of training.

Batch Gradient Descent Training

This function, `train_batch2`, trains a neural network using batch gradient descent, iteratively updating the network parameters to minimize the loss over multiple epochs.

Input Parameters:

- **`X_train`**: A numpy array of the training input data.
- **`y_one_hot_train`**: A numpy array of the one-hot encoded training labels.
- **`parameters`**: A dictionary containing the initial weights (`W`) and biases (`b`) of the neural network.
- **`total_layers`**: An integer representing the total number of layers in the neural network.
- **`epochs`**: An integer specifying the number of training epochs.
- **`eta`**: A float representing the learning rate used in gradient descent.
- **`batch_size`**: An integer specifying the number of samples per batch for gradient descent updates.
- **`g_act`**: The activation function to be used in hidden layers.
- **`d_act`**: The derivative of the activation function for backpropagation.

Process:

1. **Shuffling Data**: At the start of each epoch, the training data (`X_train`) and labels (`y_one_hot_train`) are shuffled to ensure randomness in each training iteration.
2. **Batch Processing**: The data is processed in batches, where each batch is a subset of the total training data defined by `batch_size`. For each batch:
    - The function initializes a gradient accumulation dictionary to sum gradients across samples.
    - It computes activations and gradients for each input-output pair using `forward_prop` and `backprop`.
    - It accumulates the gradients over the batch and averages them.
3. **Parameter Updates**: Parameters (`W` and `b`) are updated using gradient descent with the averaged gradients and learning rate (`eta`).
4. **Loss Calculation**: The average loss is calculated for each batch and accumulated to compute the total loss for the epoch, which is printed and recorded.

5. **Returning Results**: The function returns the updated parameters, a list of training losses per epoch (`train_losses`), and batch losses (`batch_losses`) to monitor the training progress.

Return Type:

- **`parameters`**: A dictionary with updated weights and biases after training.
- **`train_losses`**: A list of average losses for each epoch, tracking how the loss evolves over the training process.
- **`batch_losses`**: A list of losses for each batch, providing more granular insight into the training dynamics within each epoch.

Parameter Initialization

- **`parameters`**: This variable holds the initialized weights and biases of the neural network, generated using the `glorot_init` function. The function is called with 784 input features (for 28x28 flattened images), three hidden layers with 500, 250, and 100 neurons respectively, and an output layer with 10 neurons (for 10 classes in the MNIST dataset).
- **`total_layers`**: This is set to 5, representing the total number of layers in the network, including the input layer, three hidden layers, and the output layer.
- **`epochs`**: This is set to 15, specifying the number of complete passes through the entire training dataset. Each epoch allows the model to learn and refine its parameters based on the entire dataset.
- **`eta` (Learning Rate)**: This is set to 0.01 and controls the step size in the parameter update during gradient descent.
- **`batch_size`**: Set to 64, this determines the number of samples processed together in one forward/backward pass during training.

Activation Functions

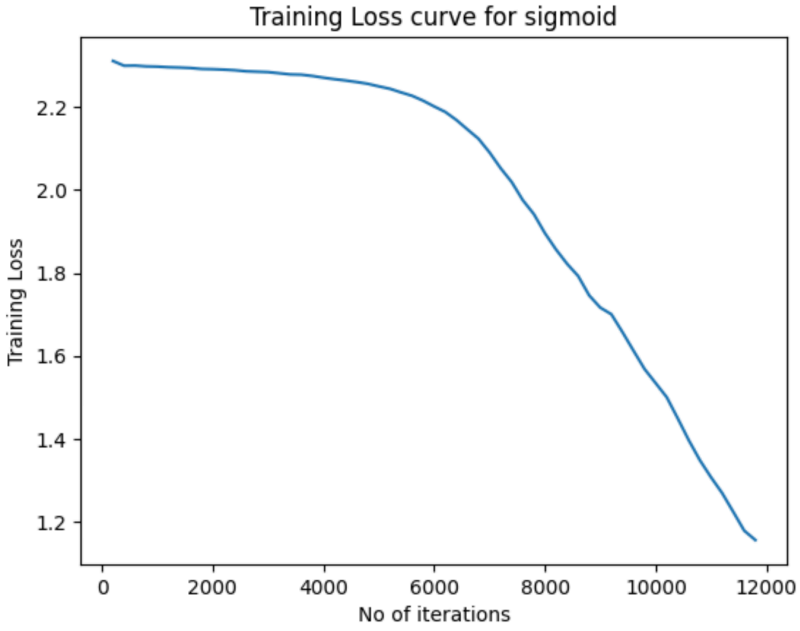
Sigmoid

Training with sigmoid Activation

```
g_act = sigmoid
d_act = dsigmoid
new_params_sig, train_losses_sig, iteration_losses_sig= train_batch2(X_train, y_train,
parameters, total_layers, epochs, eta, batch_size, g_act, d_act)
```

Plotting Training Loss over iterations

The function `average_every_200_elements` is used to compute the average training loss over every 200 iterations from the `iteration_losses_sig` list, resulting in the `sigmoid_200_iter` list. The `x` list is created to represent the iteration numbers corresponding to each averaged loss, where each point corresponds to 200 iterations. The `plt.plot` function is used to plot these averaged losses (`sigmoid_200_iter`) against the number of iterations (`x`).



Classification report of training data

Class	Precision	Recall	F1-Score	Support
0	0.77	0.91	0.84	4884
1	0.75	0.97	0.85	5672
2	0.63	0.54	0.58	4948
3	0.63	0.73	0.68	5106
4	0.59	0.63	0.61	4850
5	0.67	0.46	0.54	4513
6	0.67	0.75	0.71	4935
7	0.64	0.77	0.70	5202
8	0.64	0.41	0.50	4891
9	0.53	0.38	0.44	4999

Overall Metrics:

Metric	Value
Accuracy	0.66
Macro Avg	0.65 (Precision)
Weighted Avg	0.65 (Precision)

Classification report of test data

Class	Precision	Recall	F1-Score	Support
0	0.73	0.92	0.82	980
1	0.79	0.98	0.88	1135
2	0.67	0.54	0.60	1032
3	0.61	0.76	0.68	1010
4	0.58	0.60	0.59	982
5	0.67	0.45	0.54	892
6	0.69	0.75	0.72	958
7	0.62	0.77	0.69	1028
8	0.66	0.41	0.50	974
9	0.51	0.38	0.43	1009

Overall Metrics:

Metric	Value
Accuracy	0.66
Macro Avg	0.65 (Precision)
Weighted Avg	0.65 (Precision)

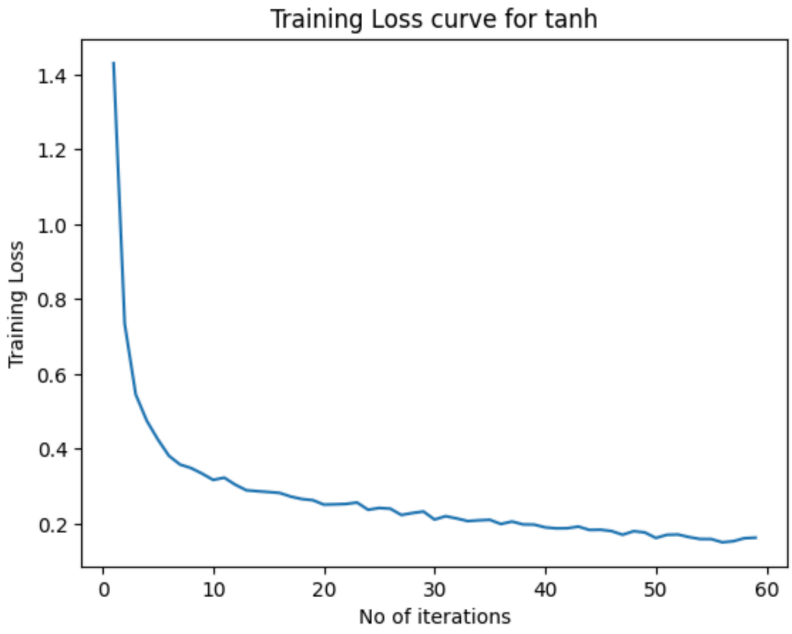
Tanh

Training with tanh Activation

```
g_act = tanh
d_act = dtanh
new_params_tanh, train_losses_tanh, iteration_losses_tanh  = train_batch2(X_train, y_train,
parameters, total_layers, epochs, eta, batch_size, g_act, d_act)
```

Plotting Training Loss over iterations

The function `average_every_200_elements` is used to compute the average training loss over every 200 iterations from the `iteration_losses_tanh` list, resulting in the `tanh_200_iter` list. The `x` list is created to represent the iteration numbers corresponding to each averaged loss, where each point corresponds to 200 iterations. The `plt.plot` function is used to plot these averaged losses (`tanh_200_iter`) against the number of iterations (`x`).



Classification report with train data

Class	Precision	Recall	F1-Score	Support
0	0.97	0.98	0.98	4884
1	0.97	0.98	0.98	5672
2	0.96	0.95	0.95	4948
3	0.94	0.96	0.95	5106
4	0.95	0.95	0.95	4850
5	0.95	0.94	0.95	4513
6	0.97	0.97	0.97	4935
7	0.97	0.96	0.96	5202
8	0.94	0.94	0.94	4891
9	0.95	0.94	0.94	4999

Overall Metrics:

Metric	Precision	Recall	F1-Score	Support
Accuracy			0.96	50000
Macro Avg	0.96	0.96	0.96	50000
Weighted Avg	0.96	0.96	0.96	50000

Classification report with test data

Class	Precision	Recall	F1-Score	Support
0	0.96	0.98	0.97	980
1	0.98	0.99	0.98	1135
2	0.96	0.94	0.95	1032
3	0.93	0.96	0.95	1010
4	0.96	0.95	0.95	982
5	0.95	0.93	0.94	892
6	0.96	0.96	0.96	958
7	0.96	0.94	0.95	1028
8	0.94	0.94	0.94	974
9	0.95	0.93	0.94	1009

Overall Metrics:

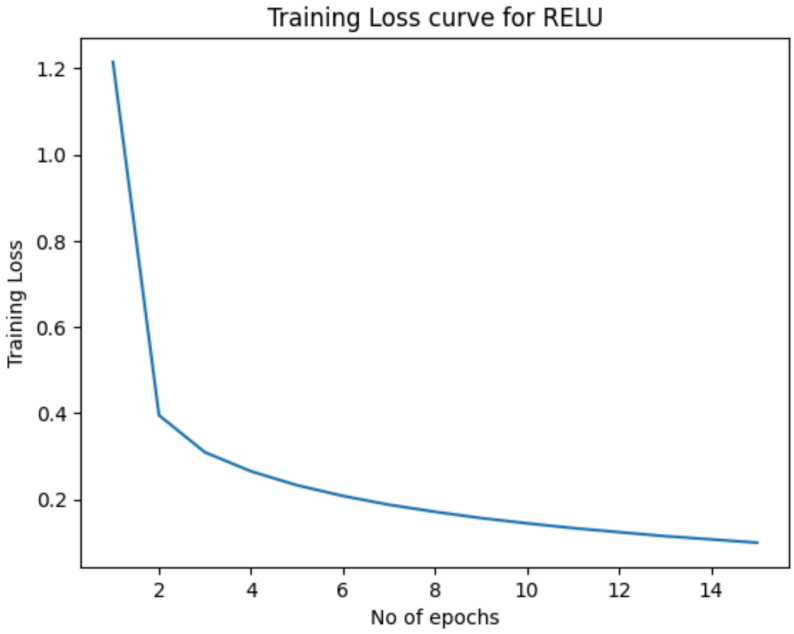
Metric	Precision	Recall	F1-Score	Support
Accuracy			0.95	10000
Macro Avg	0.95	0.95	0.95	10000
Weighted Avg	0.95	0.95	0.95	10000

RELU

```
g_act = relu
d_act = drelu
new_params_relu, train_losses_relu, iteration_losses_relu= train_batch2(X_train, y_train,
parameters, total_layers, epochs, eta, batch_size, g_act, d_act)
```

Plotting train losses for RELU

The function `average_every_200_elements` is used to compute the average training loss over every 200 iterations from the `iteration_losses_relu` list, resulting in the `relu_200_iter` list. The `x` list is created to represent the iteration numbers corresponding to each averaged loss, where each point corresponds to 200 iterations. The `plt.plot` function is used to plot these averaged losses (`relu_200_iter`) against the number of iterations (`x`).



Classification report with train data

Class	Precision	Recall	F1-Score	Support
0	0.99	0.98	0.98	4940
1	0.98	0.99	0.98	5622
2	0.99	0.95	0.97	4916
3	0.96	0.97	0.97	5102
4	0.97	0.96	0.97	4867
5	0.97	0.97	0.97	4521
6	0.98	0.98	0.98	4984
7	0.99	0.88	0.93	5216
8	0.96	0.95	0.96	4865
9	0.84	0.99	0.91	4967

Overall Metrics:

Metric	Precision	Recall	F1-Score	Support
Accuracy			0.96	50000
Macro Avg	0.96	0.96	0.96	50000
Weighted Avg	0.96	0.96	0.96	50000

Classification report with test data

Class	Precision	Recall	F1-Score	Support
0	0.98	0.98	0.98	980
1	0.98	0.99	0.98	1135
2	0.98	0.95	0.96	1032
3	0.94	0.97	0.95	1010
4	0.96	0.95	0.96	982
5	0.96	0.96	0.96	892
6	0.97	0.96	0.97	958
7	0.99	0.87	0.93	1028
8	0.96	0.95	0.96	974
9	0.85	0.97	0.91	1009

Overall Metrics:

Metric	Precision	Recall	F1-Score	Support
Accuracy			0.96	10000
Macro Avg	0.96	0.96	0.96	10000
Weighted Avg	0.96	0.96	0.96	10000

Conclusion

Based on the evaluation of different activation functions on the **MNIST dataset**, **ReLU (Rectified Linear Unit)** achieved the highest performance with an accuracy of **0.96** on the test data, surpassing both the **Tanh** and **Sigmoid** functions. This suggests that ReLU is more effective for this particular task, due to its ability to mitigate the vanishing gradient problem and provide faster convergence, making it the most suitable activation function among those tested for the MNIST data.

Pytorch Implementation

Feedforward

The class `FeedForwardNN` defines a feedforward neural network using PyTorch. It is initializesd with three hidden layers and an output layer, all connected linearly using `nn.Linear`. The network uses a specified activation function (default is `nn.Sigmoid`) for the hidden layers and a specified output function (default is `nn.Softmax`) for the output layer. The `forward` method processes input data through the layers sequentially, applying the activation function after each hidden layer and the output function after the final layer, returning the output.

Model Setup: FeedForwardNN, Loss Function, and Optimizer

1. **Model Initialization**:
   - `torch_model = FeedForwardNN(784, 500, 250, 100, 10, activation=torch.sigmoid).to(device)`:
     This line creates an instance of the `FeedForwardNN` class with an input size of 784 (for 28x28 pixel images), three hidden layers with 500, 250, and 100 neurons, respectively, and an output layer with 10 neurons (for classification into 10 classes). The hidden layers use the sigmoid activation function (`torch.sigmoid`). The model is moved to the specified device (`cuda` if available, otherwise `cpu`).
2. **Loss Function**:
   - `criterion = nn.CrossEntropyLoss()`:
     The loss function used is Cross-Entropy Loss, which is commonly used for classification tasks. It measures the difference between the predicted probability distribution and the true distribution (labels).
3. **Optimizer**:
   - `optimizer = optim.Adam(torch_model.parameters(), lr=0.01)`:
     The optimizer chosen is Adam, which is an adaptive learning rate optimization algorithm commonly used for training neural networks. The learning rate is set to 0.01.
4. **Model Display**:
   - `torch_model`:
     Simply printing the model will display its architecture, showing the structure and parameters of the defined neural network.

Train Function

Input Parameters:

- **model**: The neural network model to be trained.
- **train_loader**: DataLoader for the training dataset, providing batches of input data and corresponding labels.
- **val_loader**: DataLoader for the validation dataset, providing batches of input data and corresponding labels.
- **criterion**: The loss function used to evaluate the model's performance.
- **optimizer**: Optimization algorithm used to update the model's parameters.
- **num_epochs**: The number of epochs for which the model is trained.
- **device** (optional, default='cpu'): The device ('cpu' or 'cuda') on which the model and data will be processed.

Return Values:

- **train_losses**: A list containing the average training loss for each epoch.
- **val_losses**: A list containing the average validation loss for each epoch.

Data Preparation

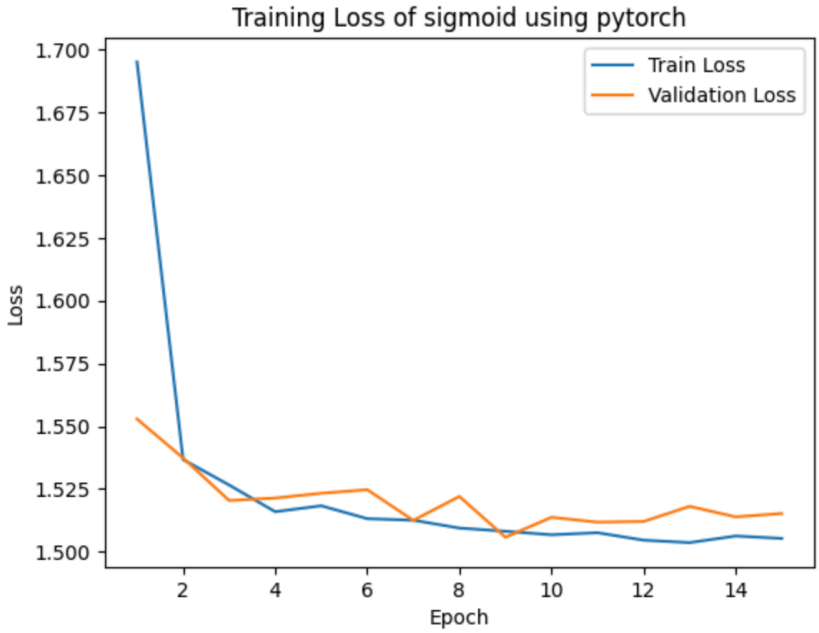
1. The code sets a batch size of 64 for loading the data.
2. It calculates the sizes for training (80%) and validation (20%) splits based on the length of `train_dataset`.
3. The training and validation datasets are created by splitting `train_dataset` into `train_data` and `val_data` using the calculated sizes.
4. A `DataLoader` is created for the training set (`train_loader`) with shuffling enabled to randomize the order of batches.
5. A `DataLoader` is created for the validation set (`val_loader`) without shuffling to maintain order.
6. A separate `DataLoader` is set up for the test dataset (`test_loader`), also without shuffling, for evaluation purposes.

Training

1. **`train` Function Call**: The function `train` is called with the neural network model (`torch_model`), training and validation data loaders (`train_loader`, `val_loader`), loss function (`criterion`), optimizer, number of epochs (`num_epochs=15`), and the device (`device`).
2. **Outputs**: It returns two lists:
    - `train_loss`: A list of average training losses for each epoch.
    - `val_loss`: A list of average validation losses for each epoch.

Plotting train and validation loss

- `x_t` and `x_v` are lists representing the epochs, created based on the length of `train_loss` and `val_loss`, respectively.
- It plots `train_loss` against `x_t` and `val_loss` against `x_v` using `matplotlib`.
- The x-axis is labeled "Epoch" and the y-axis is labeled "Loss".
- A legend distinguishes between the training loss and validation loss lines on the plot.



Evaluation

This function, `evaluate`, assesses a machine learning model's performance on a given dataset:

- It sets the model to evaluation mode with `model.eval()`, which disables certain layers like dropout for inference.
- The function initializes empty lists, `y_pred` and `true_y`, to store predictions and true labels.

- Using a context of `torch.no_grad()` (to save memory and computation by not tracking gradients), it iterates over the `data_loader`.
- For each batch, it moves the input (`x`) and labels (`y`) to the specified device (CPU by default).
- The input is reshaped and fed through the model, which produces outputs.
- It calculates the predicted class by taking the index of the maximum value in the output tensor.
- The predictions and true labels are converted to NumPy arrays and appended to their respective lists.

Train Data Classification report

Class	Precision	Recall	F1-Score	Support
0	0.97	0.98	0.98	4674
1	0.98	0.99	0.99	5391
2	0.94	0.97	0.96	4597
3	0.94	0.96	0.95	4791
4	0.97	0.93	0.95	4840
5	0.94	0.97	0.95	4243
6	0.98	0.98	0.98	4778
7	0.95	0.98	0.96	4884
8	0.90	0.95	0.92	4372
9	0.96	0.85	0.90	5430

Metric	Value
Accuracy	0.95
Macro Avg	0.95
Weighted Avg	0.95

Test Data Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.98	0.96	0.97	998
1	0.99	0.99	0.99	1130
2	0.93	0.97	0.95	989
3	0.95	0.95	0.95	1011
4	0.96	0.93	0.95	1016
5	0.93	0.96	0.95	863
6	0.97	0.96	0.96	968
7	0.94	0.97	0.95	991
8	0.89	0.94	0.91	916
9	0.96	0.86	0.91	1118

Metric	Value
Accuracy	0.95
Macro Avg	0.95
Weighted Avg	0.95

L_2 Regularization

1. **`alpha = 0.01` (Regularization Constant):**
   - This variable represents the strength of L2 regularization. It controls how much the weights are penalized in the loss function.
2. **`lr = 0.01` (Learning Rate):**
   - The learning rate is a critical hyperparameter that determines the step size at each iteration while moving towards a minimum of the loss function.
3. **`weight_decay = 1e-3` (L2 Regularization Parameter):**
   - This parameter directly applies L2 regularization by adding a penalty proportional to the sum of the squared weights to the loss function. The value `1e-3` is a specific setting that controls the extent of this penalty, influencing how strongly the optimizer discourages large weights.
4. **`torch_model.parameters()`:**
   - This function call passes the model's parameters (weights and biases) to the optimizer. It tells the optimizer which parameters to update during training.

Training Loss Plot using L2 Regularization

Classification Report for Train Data using L2 Regularization

Class	Precision	Recall	F1-Score	Support
0	1.00	0.10	0.18	48000
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	0.00	0.00	0.00	0
5	0.00	0.00	0.00	0
6	0.00	0.00	0.00	0
7	0.00	0.00	0.00	0
8	0.00	0.00	0.00	0
9	0.00	0.00	0.00	0

Overall Metrics:

Metric	Precision	Recall	F1-Score	Support
Accuracy			0.10	48000
Macro Avg	0.10	0.01	0.02	48000
Weighted Avg	1.00	0.10	0.18	48000

Classification Report for Test Data using L2 Regularization

Class	Precision	Recall	F1-Score	Support
0	1.00	0.10	0.18	10000
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	0.00	0.00	0.00	0
5	0.00	0.00	0.00	0
6	0.00	0.00	0.00	0
7	0.00	0.00	0.00	0
8	0.00	0.00	0.00	0
9	0.00	0.00	0.00	0

Overall Metrics:

Metric	Precision	Recall	F1-Score	Support
Accuracy			0.10	10000
Macro Avg	0.10	0.01	0.02	10000
Weighted Avg	1.00	0.10	0.18	10000