Lab 1. EARIN

Variant 1

Oleg Kim

Seniv Volodymyr

26.03.2023

Task description:

Implement a multilayer perceptron for image classification. The neural network should be trained with the mini-batch gradient descent. Remember to split the dataset into train and validation sets. For evaluation, create plots with:

•loss value for every learning step

•accuracy on train set after each epoch

•accuracy on validation set after each epoch

Main point of this task is to evaluate how various components/hyperparameters of neural network and training process affect the performance of the network in terms of ability to converge and the speed of convergence, and final accuracy on train and validation sets.

Use MNIST dataset. Evaluate at least 3 different numbers/values/types of:

•learning rate

•mini-batch size (including batch containing only 1 example)

•number of hidden layers (including 0 hidden layers - linear model)•width (number of neurons in hidden layers)
•loss functions (e.g. Mean Squared Error, Mean Absolute Error, Cross En-tropy)

Launch instruction:

Before starting the program, you have to install appropriate packages:

```
numpy
matplotlib.pyplot
torch
keras.datasets
sklearn.model_selection
```

After you have installed all his packages just start the script and wait for a bit because this algorithm needs time to operate and give a result.

Theoretical background:

PyTorch: is a widely used open-source deep learning framework that provides a flexible and dynamic approach to building neural networks.

Computational Graph: PyTorch adopts a dynamic computational graph approach, known as a "define-by-run" framework. It allows users to construct and modify the computation graph on the fly, making it more flexible compared to static graph frameworks. The computation graph represents the flow of operations in a neural network and enables efficient computation and gradient propagation.

Automatic Differentiation: One of the key features of PyTorch is its automatic differentiation system. This system enables the computation of gradients, which are crucial for training deep neural networks through techniques like backpropagation. By tracking the operations performed on tensors, PyTorch automatically computes and propagates gradients, simplifying the implementation of complex neural network architectures.

Tensor Operations: PyTorch provides a powerful tensor computation library, which forms the fundamental data structure for numerical computations. Tensors are multidimensional arrays that can store and process numerical data efficiently. PyTorch supports various tensor operations, such as element-wise operations, matrix multiplications, reductions, and broadcasting, enabling users to perform complex computations efficiently.

GPU Acceleration: PyTorch leverages the power of Graphics Processing Units (GPUs) to accelerate deep learning computations. It provides seamless integration with CUDA, a parallel computing platform, enabling efficient execution of tensor operations on GPUs. This GPU acceleration significantly speeds up training and inference processes, making PyTorch suitable for large-scale deep learning tasks.

Neural Network Modules: PyTorch offers a rich set of pre-defined modules and utilities for building neural networks. These modules encapsulate common components like layers, activation functions, loss functions, and optimizers. By leveraging these modules, users can easily construct and customize neural network architectures, simplifying the implementation process.

Model Training and Deployment: PyTorch provides a high-level API for training deep learning models. Users can define custom training loops or utilize the built-in utilities for tasks such as data loading, model optimization, and evaluation. Additionally, PyTorch supports model deployment through integration with frameworks like TorchServe and ONNX, enabling seamless deployment of trained models in production environments.

MNIST:

The MNIST (Modified National Institute of Standards and Technology) dataset is a widely recognized benchmark in the field of machine learning and computer vision. This section provides a theoretical background on MNIST, including its origin, structure, challenges, and applications. MNIST has become a fundamental benchmark for evaluating the performance of machine learning algorithms, specifically in the domain of handwritten digit recognition. The goal is to develop models that can accurately classify the digits (0-9) based on the pixel intensities of the corresponding images. Researchers often use MNIST as a starting point to test and compare different algorithms, architectures, and techniques.

Learning rate:

The learning rate is typically a small positive value and is denoted by the symbol α (alpha). It controls the magnitude of parameter updates and affects the speed and stability of the learning process. Selecting an appropriate learning rate involves a trade-off between two key considerations:

1. Convergence Speed: A higher learning rate can accelerate convergence by taking larger steps towards the optimal solution. It allows the model to reach a reasonably good solution faster, especially in the early stages of training when the gradients may be steep. However, setting the learning rate too high can lead to overshooting the optimal solution or even divergence, causing the loss function to fluctuate or increase.
2. Stability and Accuracy: A lower learning rate ensures more stable and accurate parameter updates. It reduces the risk of overshooting the optimal solution and allows the model to fine-tune its parameters more precisely. However, setting the learning rate too low may result in slow convergence, requiring more iterations to achieve satisfactory performance or potentially getting stuck in suboptimal solutions.

Mini-batch size (including batch containing only 1 example)

The mini-batch size refers to the number of training examples that are processed together in parallel during each iteration of training in a machine learning algorithm. It is an important hyperparameter that affects the training dynamics and computational efficiency. The mini-batch size can vary, including the case of a batch containing only a single example, known as a batch size of 1 or online learning.

Here are some key points regarding different mini-batch sizes, including the batch size of 1:

1. Batch Size > 1 (Mini-Batch Training):
    - Common practice: Mini-batch training is widely used in deep learning. It involves dividing the training data into small batches, typically ranging from a few examples to a few hundred examples.

    - Advantages: Mini-batch training offers several advantages. It enables parallel processing of multiple examples, which can significantly speed up training on hardware with parallel computing capabilities (e.g., GPUs). It also allows for a more stable and representative estimate of the gradient, reducing the variance in parameter updates compared to using a single example (online learning).
    - Trade-offs: The choice of mini-batch size involves trade-offs. Larger mini-batches can provide a smoother estimate of the gradient, leading to faster convergence. However, they may require more memory and can make the learning process less noisy, potentially causing the model to converge to a flatter and less optimal region of the parameter space. Smaller mini-batches can introduce more noise but might offer better generalization and the ability to escape shallow local minima.
2. Batch Size = 1 (Online Learning):
    - Online learning: When the batch size is set to 1, it corresponds to online learning or stochastic gradient descent (SGD). In this case, the model updates its parameters after each individual training example.
    - Advantages: Online learning offers several advantages. It can adapt quickly to changing data patterns and is often more computationally efficient, as only a single example needs to be processed in each iteration. It also provides a more frequent and responsive update of the model's parameters, which can be beneficial when the data distribution is non-stationary or when dealing with online streaming data.
    - Trade-offs: Online learning can exhibit higher variance in parameter updates due to the use of a single example, potentially leading to noisy convergence. Additionally, the computational overhead of processing and updating the model after each example might be higher compared to mini-batch training.

Number of hidden layers (including 0 hidden layers - linear model)

The number of hidden layers in a neural network, including the case of zero hidden layers (a linear model), is an important architectural choice that affects the network's capacity, representational power, and ability to learn complex patterns. Here, we'll discuss the implications of different numbers of hidden layers, including the linear model scenario.

1. Zero Hidden Layers (Linear Model):
    - A neural network with zero hidden layers is essentially a linear model. It consists of only an input layer and an output layer.

    - Linear models can model linear relationships between the input features and the target variable. They have limited capacity and are suitable for tasks where the data can be effectively described by a linear relationship.
    - Linear models are interpretable and computationally efficient but may not capture more intricate nonlinear patterns in the data.
2. Single Hidden Layer:
    - A neural network with a single hidden layer is often referred to as a shallow neural network.
    - Adding a single hidden layer introduces the capacity to learn nonlinear transformations of the input features.
    - Shallow neural networks can capture more complex patterns compared to linear models. They have the ability to model a wider range of functions and are useful for tasks that require capturing nonlinear relationships.
3. Multiple Hidden Layers (Deep Neural Networks):
    - Deep neural networks (DNNs) consist of two or more hidden layers between the input and output layers.
    - Having multiple hidden layers allows the network to learn hierarchical representations of the input data.
    - Deep neural networks can learn complex and abstract representations, enabling them to capture intricate nonlinear relationships and perform well on tasks such as image recognition, natural language processing, and speech recognition.
    - However, training deep neural networks can be challenging due to issues like vanishing or exploding gradients and overfitting. Techniques like regularization, skip connections, and careful initialization can help address these challenges.

Width (number of neurons in hidden layers):

The width, also known as the number of neurons or units, in the hidden layers of a neural network refers to the number of nodes in each hidden layer. The width of the hidden layers plays a crucial role in determining the model's capacity, representation power, and ability to learn complex patterns. Here, we'll discuss the implications of different widths in the hidden layers.

1. Fewer Neurons (Narrow Hidden Layers):
    - Using a smaller number of neurons in the hidden layers reduces the model's capacity and restricts its ability to capture complex patterns.

    - Narrow hidden layers can be useful when dealing with limited training data or to prevent overfitting in situations where the model has a high risk of memorizing noise or outliers.
    - However, narrower layers may struggle to learn intricate or fine-grained patterns, particularly in complex tasks where the input-output mapping is highly nonlinear.
2. More Neurons (Wide Hidden Layers):
    - Increasing the number of neurons in the hidden layers expands the model's capacity and allows for capturing more complex patterns and relationships in the data.
    - Wide hidden layers enable the model to learn more diverse and abstract representations, which can be beneficial for tasks that involve high-dimensional data or complex decision boundaries.
    - However, wider layers may require a larger amount of training data to avoid overfitting, as they have more flexibility to fit noise or irrelevant patterns in the training set.

Finding the appropriate width for the hidden layers typically involves a trade-off between model complexity, generalization performance, and the available training data. Some considerations include:

- Overfitting: If the model tends to overfit the training data, reducing the width of the hidden layers can help mitigate overfitting and improve generalization performance.
- Computational Resources: Wider hidden layers increase the computational requirements during both training and inference, as they involve more calculations and memory usage. The available computational resources should be taken into account when determining the width.
- Empirical Evaluation: The choice of the width is often determined through empirical experimentation and validation. It is common to start with a moderate width and adjust it based on the observed performance on a validation set or through techniques like cross-validation.

Loss functions (e.g. Mean Squared Error, Mean Absolute Error, Cross En-tropy):

Loss functions are an essential component of machine learning algorithms that quantify the discrepancy between predicted and target values. The choice of a loss function depends on the specific problem and the nature of the data. Here are some commonly used loss functions:

1. Mean Squared Error (MSE):

- MSE is widely used for regression problems, where the goal is to predict continuous numeric values.
- It calculates the average squared difference between the predicted and target values, penalizing larger errors more heavily.
- MSE is differentiable, making it suitable for optimization using gradient-based methods.
- However, it can be sensitive to outliers and may prioritize minimizing large errors over smaller ones.

2. Mean Absolute Error (MAE):
    - MAE is another loss function commonly used for regression tasks.
    - It measures the average absolute difference between the predicted and target values, providing a more robust measure of error compared to MSE.
    - MAE is less sensitive to outliers, as it does not square the errors.
    - However, it is less sensitive to small errors, which may result in suboptimal performance in some cases.
3. Cross-Entropy Loss:
    - Cross-entropy loss is commonly used in classification tasks, particularly when the targets are represented as one-hot encoded vectors.
    - It measures the dissimilarity between the predicted class probabilities and the true class labels.
    - Cross-entropy loss is beneficial for training models that aim to estimate class probabilities, such as logistic regression and neural networks with softmax activation.
    - It encourages the model to assign high probabilities to the correct class and penalizes confident incorrect predictions.
4. Binary Cross-Entropy Loss:
    - Binary cross-entropy loss is a variant of cross-entropy loss specifically designed for binary classification tasks.
    - It quantifies the dissimilarity between the predicted probabilities of the positive class and the true binary labels.
    - Binary cross-entropy loss is commonly used in logistic regression and neural networks with sigmoid activation at the output layer.
5. Categorical Cross-Entropy Loss:
    - Categorical cross-entropy loss is used for multi-class classification problems, where the targets are represented as one-hot encoded vectors.
    - It measures the dissimilarity between the predicted class probabilities and the true class labels.

- Categorical cross-entropy loss is suitable for training models that aim to estimate class probabilities, such as multi-class logistic regression and neural networks with softmax activation.

Solution:

Preprocessing the dataset:

1. Load the MNIST dataset using the mnist.load_data() function. This function returns the training and testing images along with their corresponding labels.
2. Normalize the input images by dividing the pixel values by 255. This step scales the pixel values to the range [0, 1], making them suitable for training neural networks.
3. Flatten the images using np.reshape(). This converts the 2D images into 1D vectors, which is a common requirement for many machine learning algorithms.
4. Convert the labels to one-hot encoded vectors using nn.functional.one_hot(). This step converts the integer labels into a binary vector representation, where each vector has a length equal to the number of classes (10 in the case of MNIST). The element corresponding to the true class is set to 1, while the rest are set to 0.
5. Split the dataset into training and validation sets using train_test_split(). This function randomly divides the data into training and validation subsets, with 80% used for training and 20% for validation. This split helps evaluate the model's performance on unseen data and prevent overfitting.
6. Create PyTorch TensorDataset objects for the training and validation data. This class is used to wrap the input features and labels into a dataset object that can be easily accessed during training.
7. Create PyTorch DataLoader objects for the training and validation datasets. The DataLoader provides an iterable interface to efficiently load the data in mini-batches during training. It takes care of shuffling the data (in the case of the training set) and dividing it into batches.

Class Model(nn.Module):

1. The __init__ method:
   - The constructor of the Model class initializes the model's architecture and parameters.
   - It takes three arguments: num_hidden_layers, width, and activation.
   - num_hidden_layers specifies the number of hidden layers in the model.

    - width determines the number of neurons in each hidden layer.
    - activation is a string representing the activation function to be used in the model.
2. Architecture:
    - The model consists of an input layer, multiple hidden layers, and an output layer.
    - The input layer is defined using the nn.Linear module, which performs a linear transformation on the input data.
    - The hidden layers are defined using nn.Linear as well, with the specified width for each hidden layer.
    - The output layer is also defined using nn.Linear, with 10 output units corresponding to the 10 classes in the MNIST dataset.
3. Activation Function:
    - The chosen activation function is specified through the activation argument, which is dynamically retrieved from the nn module using getattr().
    - Common activation functions include ReLU, Sigmoid, Tanh, etc.
4. The forward method:
    - This method defines the forward pass of the model, specifying how the input flows through the layers.
    - It takes an input tensor x and applies the model's layers in sequence.
    - The input tensor is first passed through the input layer with the specified activation function applied.
    - Then, the tensor is passed through each hidden layer using the same activation function.
    - Finally, the tensor is passed through the output layer without applying any activation function.
    - The resulting tensor is returned as the output of the forward pass.

train_model(model, learning_rate, batch_size, loss_fn, num_epochs, width):

1. Setup:
    - The function takes several arguments, including the `model` to train, `learning_rate`, `batch_size`, `loss_fn` (as a string representing the loss function), `num_epochs`, and the `width` (number of neurons in the hidden layers).
    - An Adam optimizer is initialized with the model parameters and the specified learning rate.
    - The loss function is dynamically retrieved from the `nn` module using `getattr()` and instantiated.
    - Lists are initialized to store the training and validation losses, as well as the training and validation accuracies.

2. Training Loop:
    - The function performs training for the specified number of epochs.
    - Inside each epoch, the model is set to training mode using `model.train()`.
    - The training data is iterated through in batches using the `train_loader`.
    - For each batch, the optimizer gradients are zeroed using `optimizer.zero_grad()`.
    - The model predicts the output for the batch using `model(x)`.
    - The loss between the predictions and the true labels is calculated using the specified loss function.
    - The loss is backpropagated through the model using `loss.backward()`, and the optimizer updates the model's parameters using `optimizer.step()`.
    - Training loss, accuracy, and statistics are calculated and accumulated.
3. Validation Loop:
    - After each epoch of training, the model is set to evaluation mode using `model.eval()`.
    - The validation data is iterated through in batches using the `val_loader`.
    - The model predicts the output for the validation batch.
    - The loss between the predictions and the true labels is calculated, and validation loss and accuracy statistics are accumulated.
4. Printing and Plotting:
    - After each epoch, the training and validation losses, as well as the training and validation accuracies, are printed.
    - Training curves (loss and accuracy) are plotted using matplotlib.
5. Output:
    - The function returns the lists of training and validation losses, as well as the training and validation accuracies.

Experiment 1: Varying Learning Rate

- Three learning rates (0.001, 0.01, 0.1) are tested.
- For each learning rate, a model with one hidden layer (width=128, activation=ReLU) is trained.
- The train_model function is called with the specified learning rate and other parameters.

Experiment 2: Varying Mini-batch Size

- Three batch sizes (16, 64, 256) are tested.
- For each batch size, a model with one hidden layer (width=128, activation=ReLU) is trained.
- The train_model function is called with the specified batch size and other parameters.

Experiment 3: Varying Number of Hidden Layers

- Three configurations of the number of hidden layers (0, 1, 2) are tested.
- For each configuration, a model with the specified number of hidden layers (width=128, activation=ReLU) is trained.
- The train_model function is called with the specified number of hidden layers and other parameters.

Experiment 4: Change Width

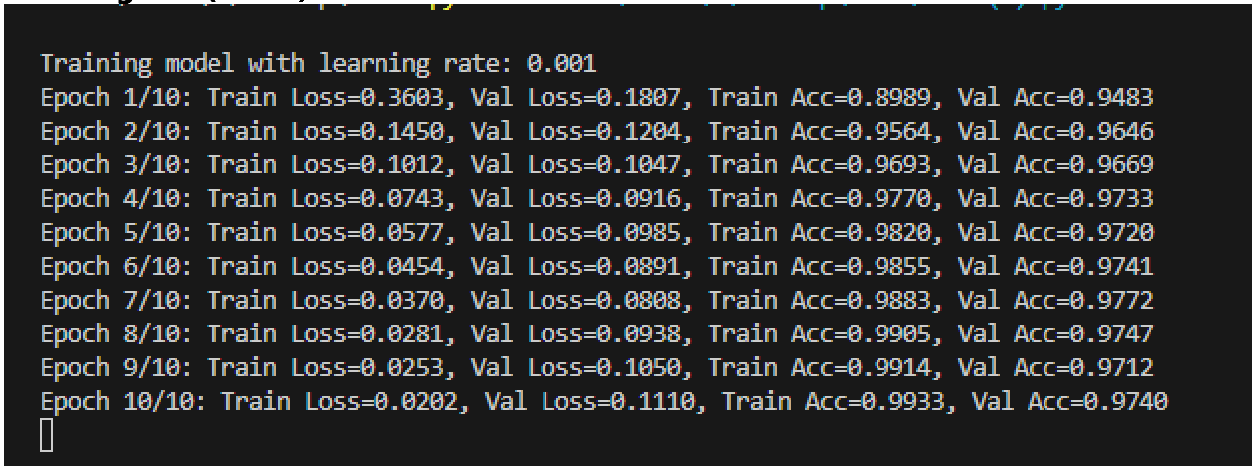
- Three widths (32, 64, 128) for the hidden layers are tested.
- For each width, a model with one hidden layer (width=width, activation=ReLU) is trained.
- The train_model function is called with the specified width and other parameters.

Experiment 5: Change Loss Function

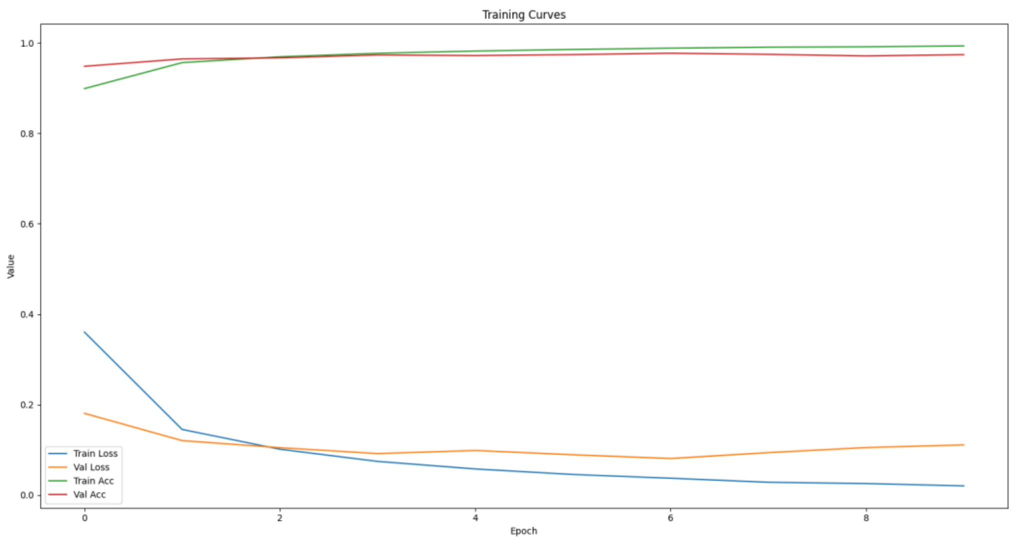
- Three loss functions (MSELoss, L1Loss, CrossEntropyLoss) are tested.
- For each loss function, a model with one hidden layer (width=128, activation=ReLU) is trained.
- The train_model function is called with the specified loss function and other parameters.

Result:

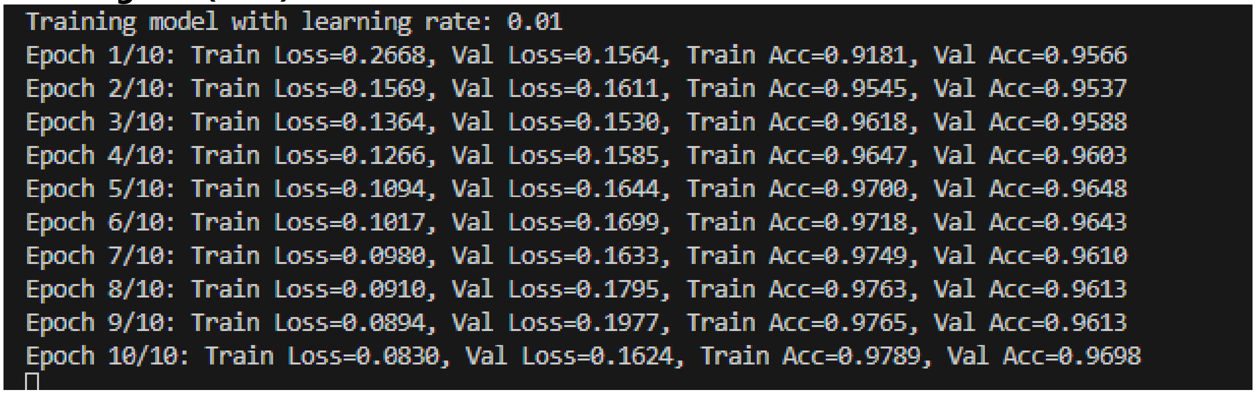
Learning rate(0.001):



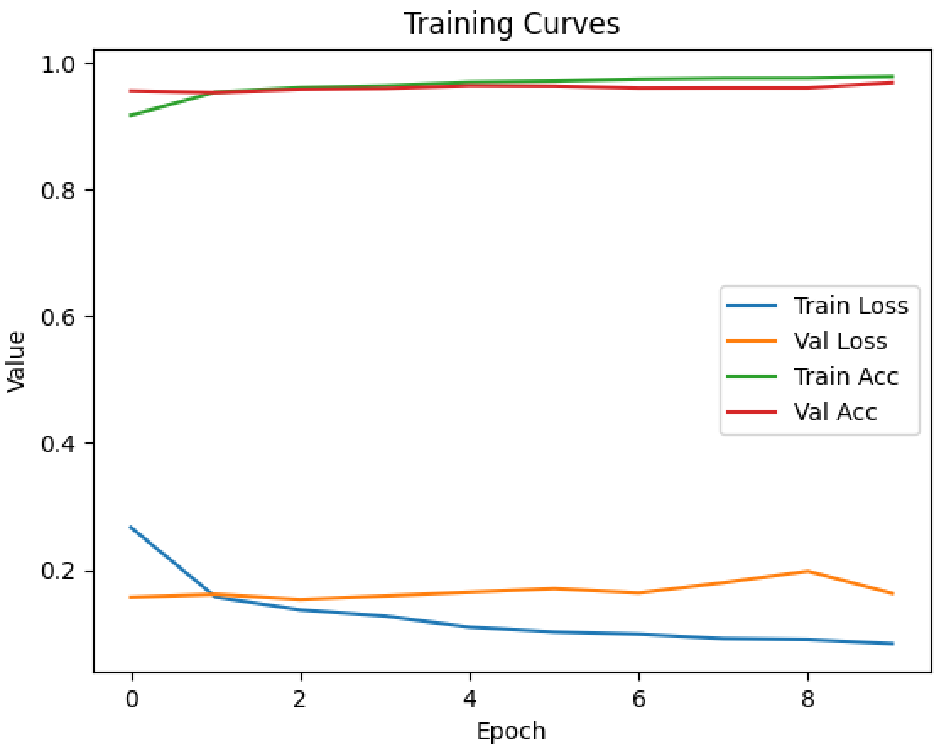
```
Training model with learning rate: 0.001
Epoch 1/10: Train Loss=0.3603, Val Loss=0.1807, Train Acc=0.8989, Val Acc=0.9483
Epoch 2/10: Train Loss=0.1450, Val Loss=0.1204, Train Acc=0.9564, Val Acc=0.9646
Epoch 3/10: Train Loss=0.1012, Val Loss=0.1047, Train Acc=0.9693, Val Acc=0.9669
Epoch 4/10: Train Loss=0.0743, Val Loss=0.0916, Train Acc=0.9770, Val Acc=0.9733
Epoch 5/10: Train Loss=0.0577, Val Loss=0.0985, Train Acc=0.9820, Val Acc=0.9720
Epoch 6/10: Train Loss=0.0454, Val Loss=0.0891, Train Acc=0.9855, Val Acc=0.9741
Epoch 7/10: Train Loss=0.0370, Val Loss=0.0808, Train Acc=0.9883, Val Acc=0.9772
Epoch 8/10: Train Loss=0.0281, Val Loss=0.0938, Train Acc=0.9905, Val Acc=0.9747
Epoch 9/10: Train Loss=0.0253, Val Loss=0.1050, Train Acc=0.9914, Val Acc=0.9712
Epoch 10/10: Train Loss=0.0202, Val Loss=0.1110, Train Acc=0.9933, Val Acc=0.9740
```



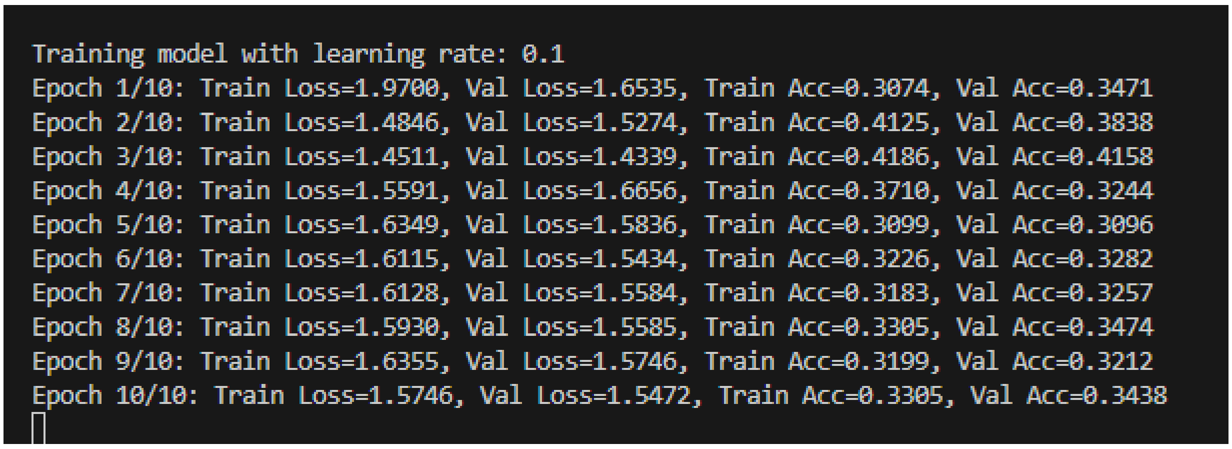
Learning rate(0.01):



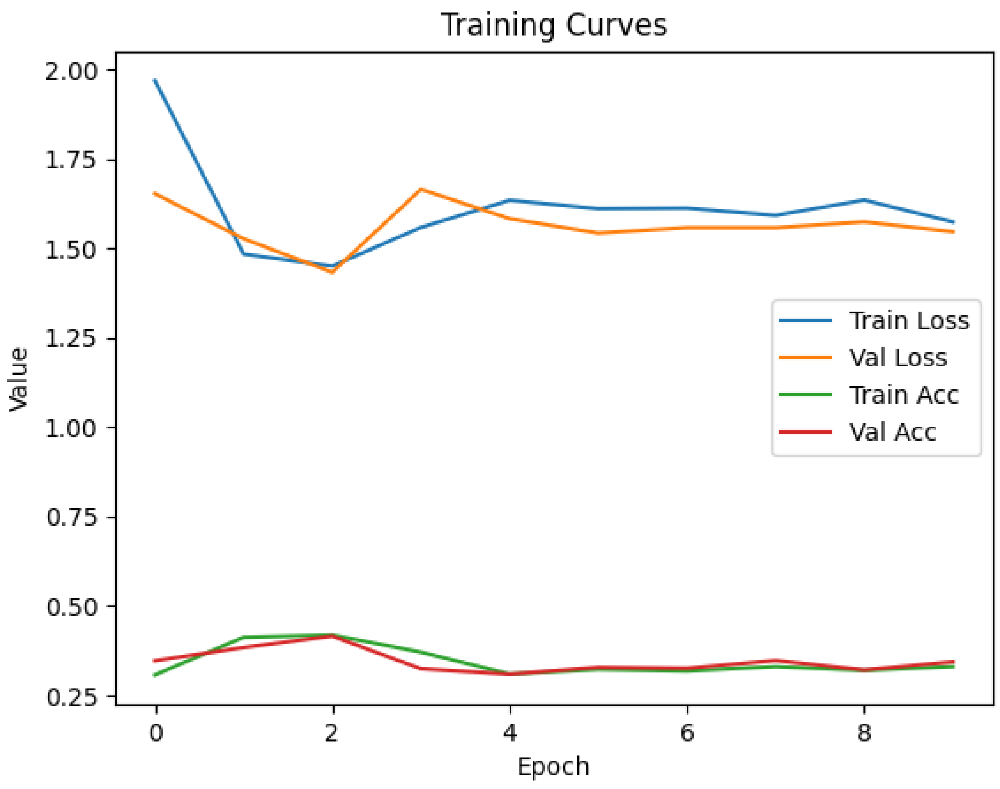
```
Training model with learning rate: 0.01
Epoch 1/10: Train Loss=0.2668, Val Loss=0.1564, Train Acc=0.9181, Val Acc=0.9566
Epoch 2/10: Train Loss=0.1569, Val Loss=0.1611, Train Acc=0.9545, Val Acc=0.9537
Epoch 3/10: Train Loss=0.1364, Val Loss=0.1530, Train Acc=0.9618, Val Acc=0.9588
Epoch 4/10: Train Loss=0.1266, Val Loss=0.1585, Train Acc=0.9647, Val Acc=0.9603
Epoch 5/10: Train Loss=0.1094, Val Loss=0.1644, Train Acc=0.9700, Val Acc=0.9648
Epoch 6/10: Train Loss=0.1017, Val Loss=0.1699, Train Acc=0.9718, Val Acc=0.9643
Epoch 7/10: Train Loss=0.0980, Val Loss=0.1633, Train Acc=0.9749, Val Acc=0.9610
Epoch 8/10: Train Loss=0.0910, Val Loss=0.1795, Train Acc=0.9763, Val Acc=0.9613
Epoch 9/10: Train Loss=0.0894, Val Loss=0.1977, Train Acc=0.9765, Val Acc=0.9613
Epoch 10/10: Train Loss=0.0830, Val Loss=0.1624, Train Acc=0.9789, Val Acc=0.9698
```



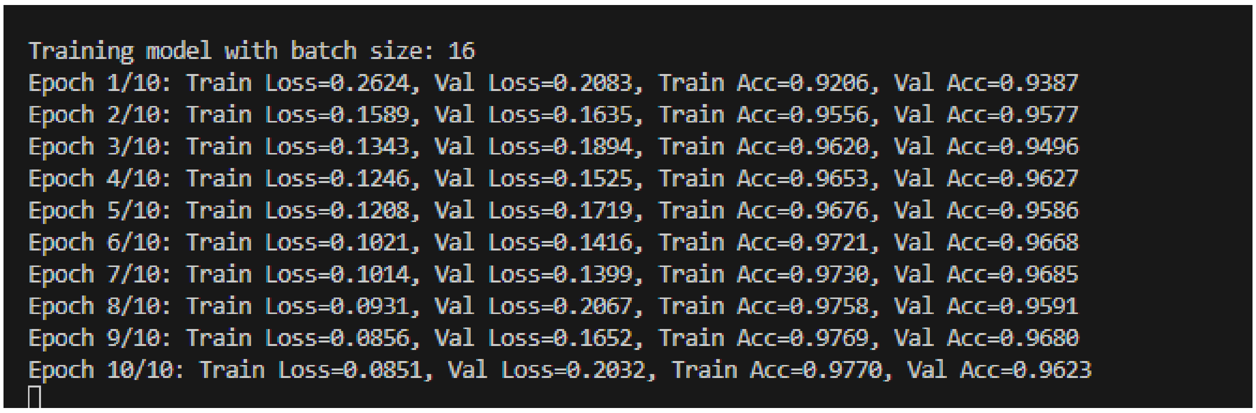
Learning rate(0.1):



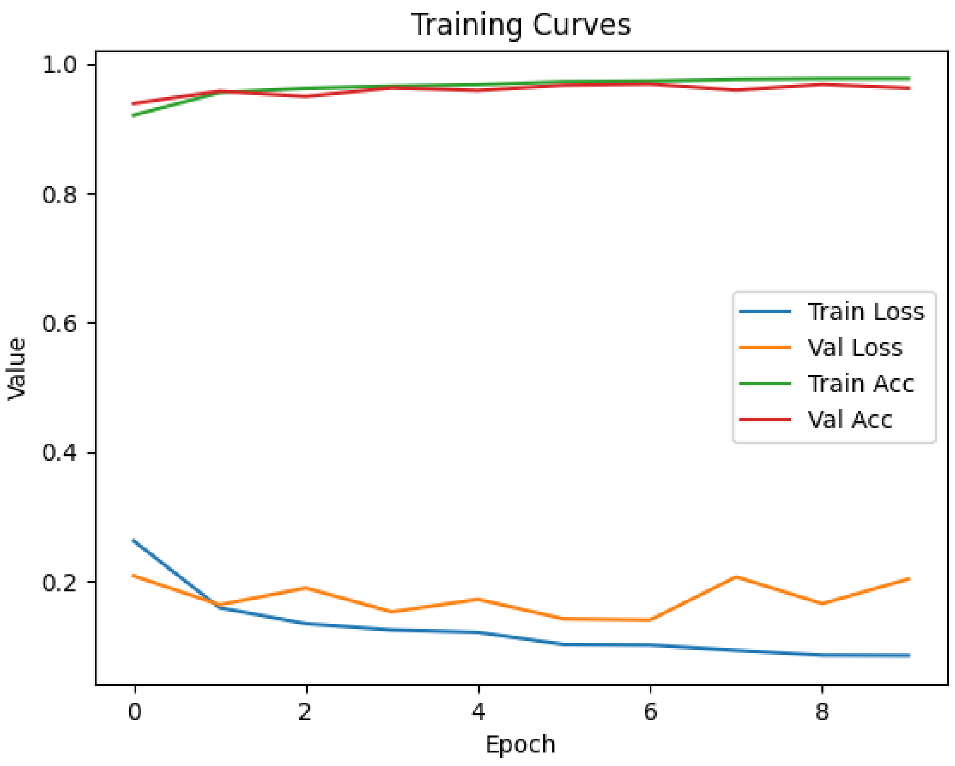
```
Training model with learning rate: 0.1
Epoch 1/10: Train Loss=1.9700, Val Loss=1.6535, Train Acc=0.3074, Val Acc=0.3471
Epoch 2/10: Train Loss=1.4846, Val Loss=1.5274, Train Acc=0.4125, Val Acc=0.3838
Epoch 3/10: Train Loss=1.4511, Val Loss=1.4339, Train Acc=0.4186, Val Acc=0.4158
Epoch 4/10: Train Loss=1.5591, Val Loss=1.6656, Train Acc=0.3710, Val Acc=0.3244
Epoch 5/10: Train Loss=1.6349, Val Loss=1.5836, Train Acc=0.3099, Val Acc=0.3096
Epoch 6/10: Train Loss=1.6115, Val Loss=1.5434, Train Acc=0.3226, Val Acc=0.3282
Epoch 7/10: Train Loss=1.6128, Val Loss=1.5584, Train Acc=0.3183, Val Acc=0.3257
Epoch 8/10: Train Loss=1.5930, Val Loss=1.5585, Train Acc=0.3305, Val Acc=0.3474
Epoch 9/10: Train Loss=1.6355, Val Loss=1.5746, Train Acc=0.3199, Val Acc=0.3212
Epoch 10/10: Train Loss=1.5746, Val Loss=1.5472, Train Acc=0.3305, Val Acc=0.3438
```



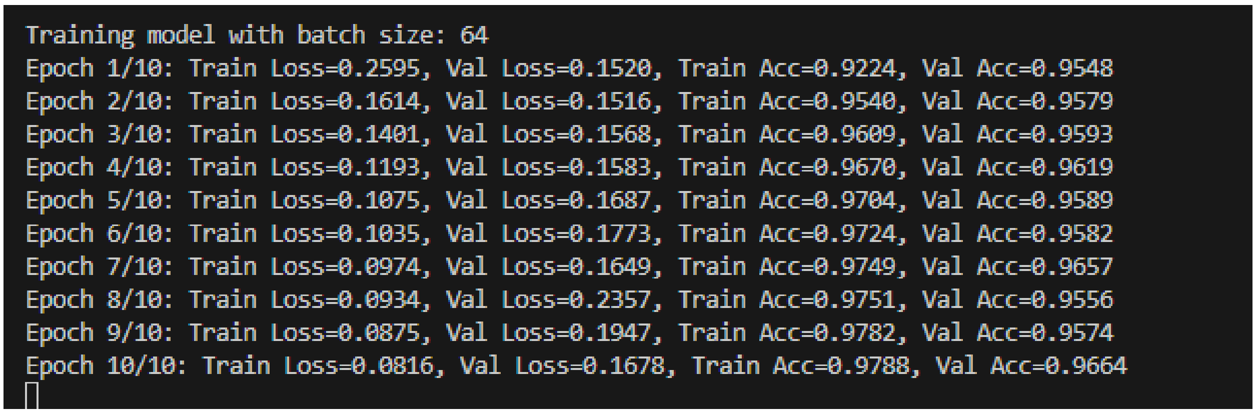
Batch size (16):



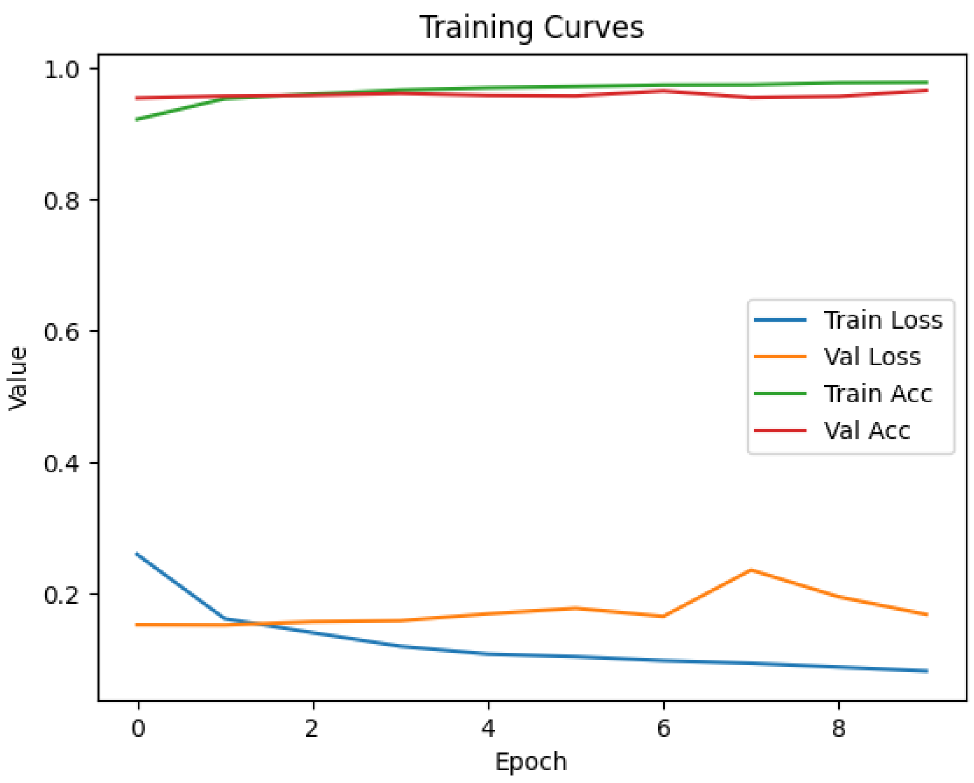
```
Training model with batch size: 16
Epoch 1/10: Train Loss=0.2624, Val Loss=0.2083, Train Acc=0.9206, Val Acc=0.9387
Epoch 2/10: Train Loss=0.1589, Val Loss=0.1635, Train Acc=0.9556, Val Acc=0.9577
Epoch 3/10: Train Loss=0.1343, Val Loss=0.1894, Train Acc=0.9620, Val Acc=0.9496
Epoch 4/10: Train Loss=0.1246, Val Loss=0.1525, Train Acc=0.9653, Val Acc=0.9627
Epoch 5/10: Train Loss=0.1208, Val Loss=0.1719, Train Acc=0.9676, Val Acc=0.9586
Epoch 6/10: Train Loss=0.1021, Val Loss=0.1416, Train Acc=0.9721, Val Acc=0.9668
Epoch 7/10: Train Loss=0.1014, Val Loss=0.1399, Train Acc=0.9730, Val Acc=0.9685
Epoch 8/10: Train Loss=0.0931, Val Loss=0.2067, Train Acc=0.9758, Val Acc=0.9591
Epoch 9/10: Train Loss=0.0856, Val Loss=0.1652, Train Acc=0.9769, Val Acc=0.9680
Epoch 10/10: Train Loss=0.0851, Val Loss=0.2032, Train Acc=0.9770, Val Acc=0.9623
```



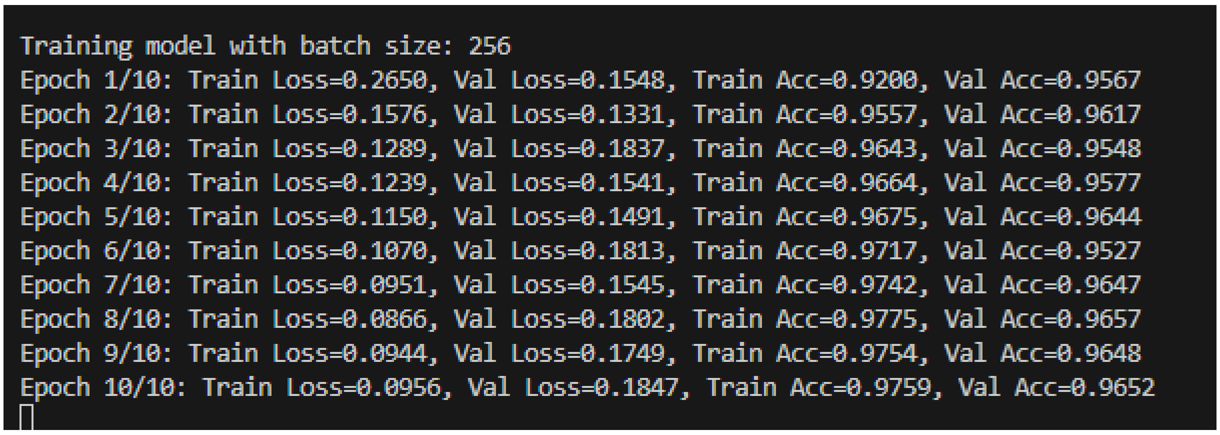
Batch size (64):



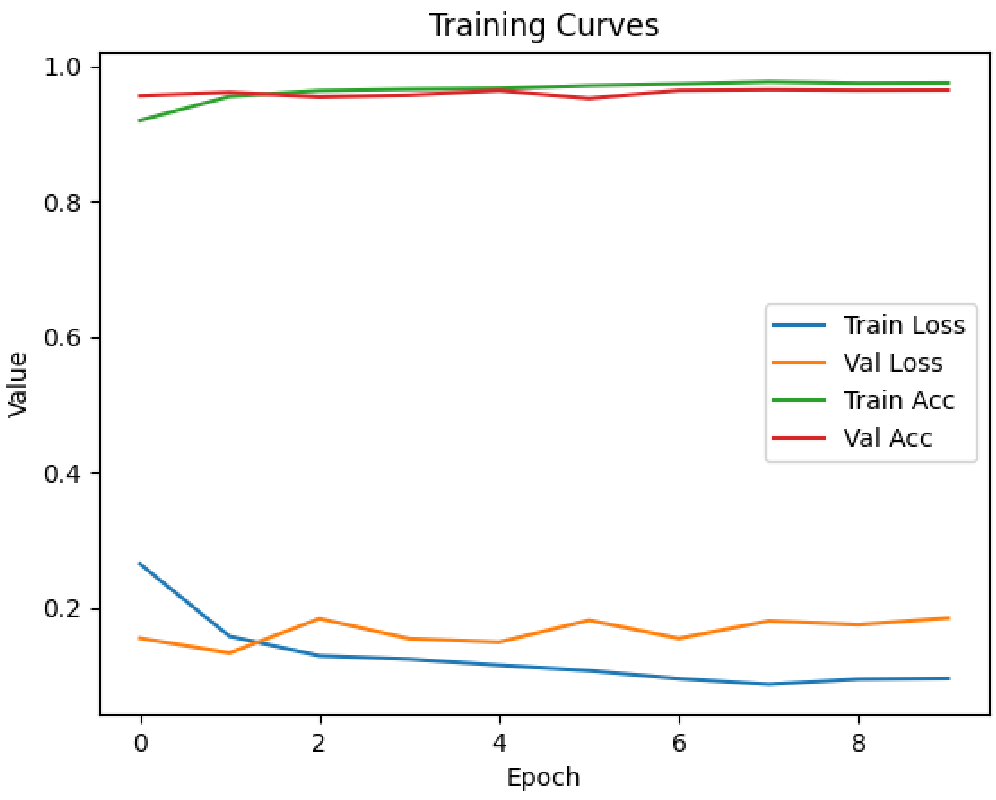
```
Training model with batch size: 64
Epoch 1/10: Train Loss=0.2595, Val Loss=0.1520, Train Acc=0.9224, Val Acc=0.9548
Epoch 2/10: Train Loss=0.1614, Val Loss=0.1516, Train Acc=0.9540, Val Acc=0.9579
Epoch 3/10: Train Loss=0.1401, Val Loss=0.1568, Train Acc=0.9609, Val Acc=0.9593
Epoch 4/10: Train Loss=0.1193, Val Loss=0.1583, Train Acc=0.9670, Val Acc=0.9619
Epoch 5/10: Train Loss=0.1075, Val Loss=0.1687, Train Acc=0.9704, Val Acc=0.9589
Epoch 6/10: Train Loss=0.1035, Val Loss=0.1773, Train Acc=0.9724, Val Acc=0.9582
Epoch 7/10: Train Loss=0.0974, Val Loss=0.1649, Train Acc=0.9749, Val Acc=0.9657
Epoch 8/10: Train Loss=0.0934, Val Loss=0.2357, Train Acc=0.9751, Val Acc=0.9556
Epoch 9/10: Train Loss=0.0875, Val Loss=0.1947, Train Acc=0.9782, Val Acc=0.9574
Epoch 10/10: Train Loss=0.0816, Val Loss=0.1678, Train Acc=0.9788, Val Acc=0.9664
```



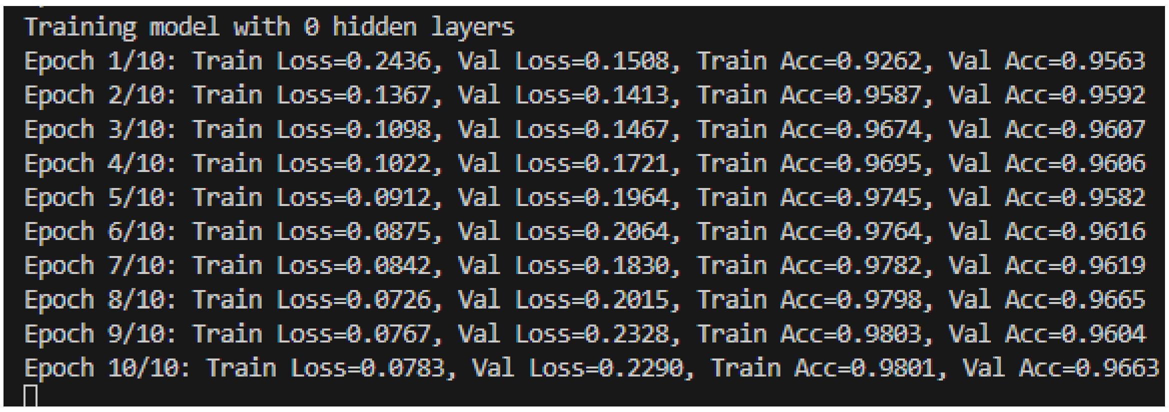
Batch size (256):



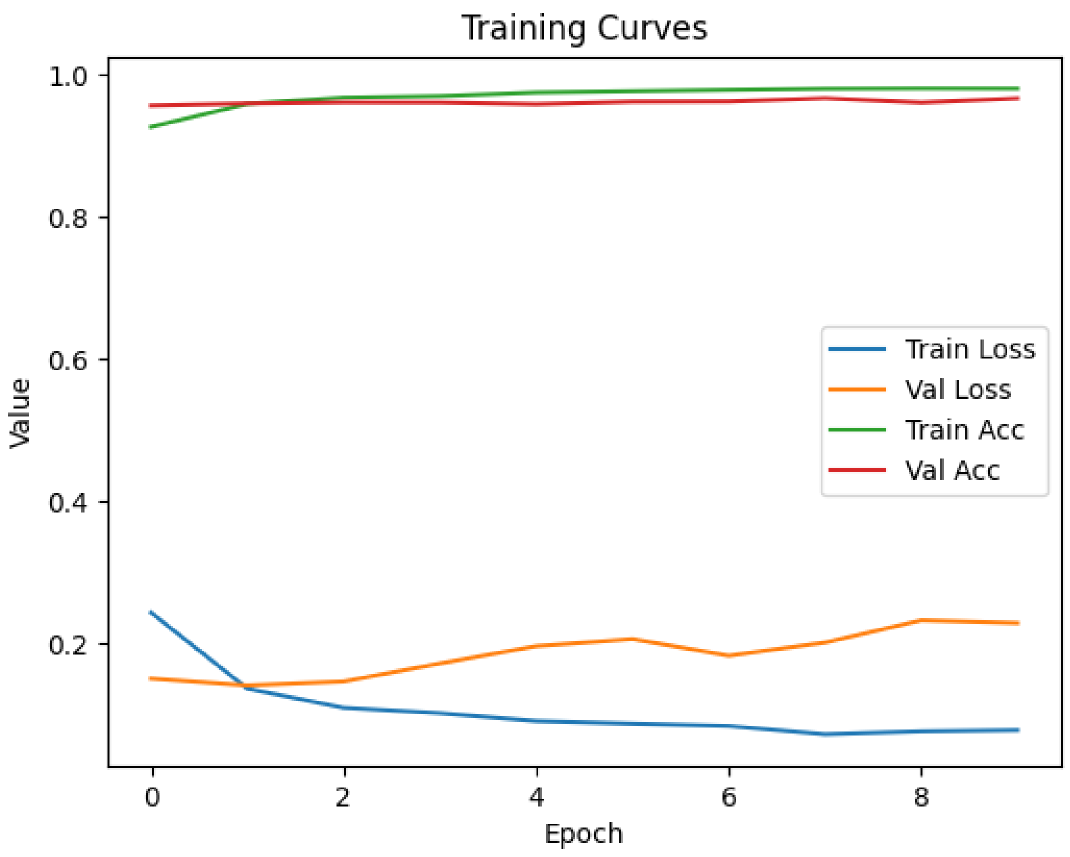
```
Training model with batch size: 256
Epoch 1/10: Train Loss=0.2650, Val Loss=0.1548, Train Acc=0.9200, Val Acc=0.9567
Epoch 2/10: Train Loss=0.1576, Val Loss=0.1331, Train Acc=0.9557, Val Acc=0.9617
Epoch 3/10: Train Loss=0.1289, Val Loss=0.1837, Train Acc=0.9643, Val Acc=0.9548
Epoch 4/10: Train Loss=0.1239, Val Loss=0.1541, Train Acc=0.9664, Val Acc=0.9577
Epoch 5/10: Train Loss=0.1150, Val Loss=0.1491, Train Acc=0.9675, Val Acc=0.9644
Epoch 6/10: Train Loss=0.1070, Val Loss=0.1813, Train Acc=0.9717, Val Acc=0.9527
Epoch 7/10: Train Loss=0.0951, Val Loss=0.1545, Train Acc=0.9742, Val Acc=0.9647
Epoch 8/10: Train Loss=0.0866, Val Loss=0.1802, Train Acc=0.9775, Val Acc=0.9657
Epoch 9/10: Train Loss=0.0944, Val Loss=0.1749, Train Acc=0.9754, Val Acc=0.9648
Epoch 10/10: Train Loss=0.0956, Val Loss=0.1847, Train Acc=0.9759, Val Acc=0.9652
```



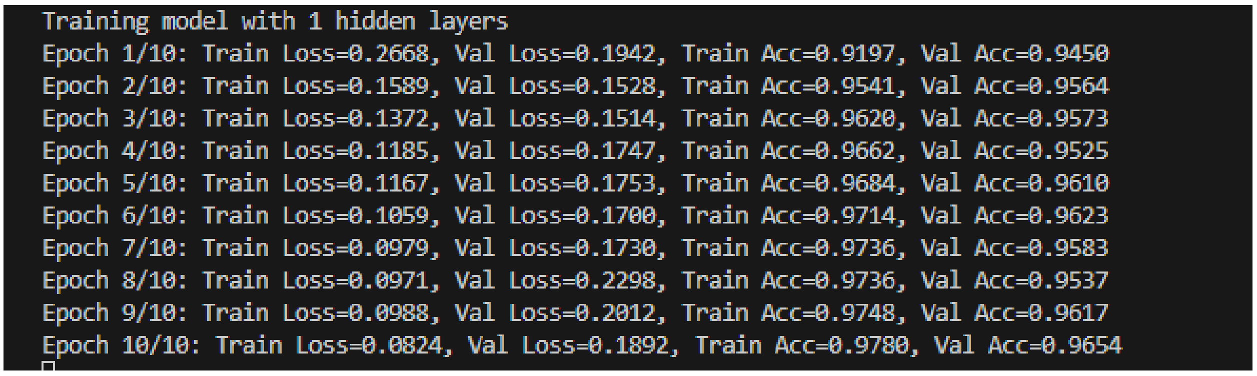
Hidden Layers (0):



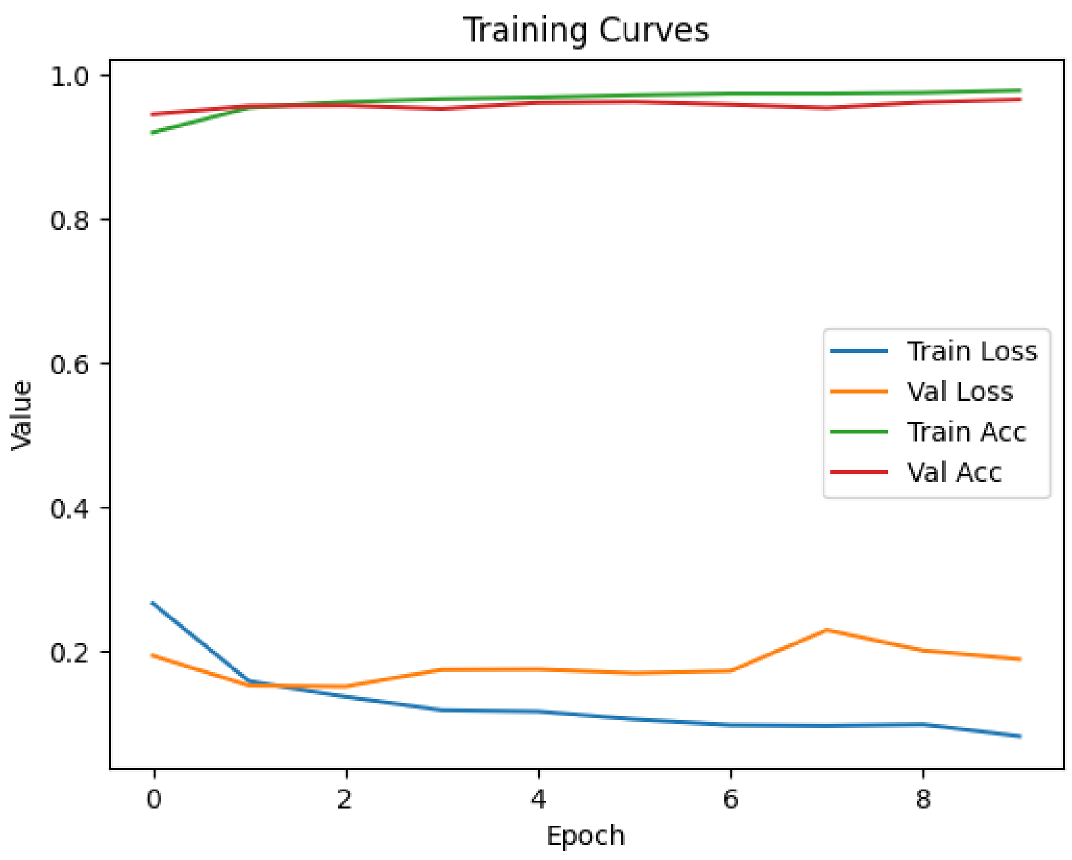
```
Training model with 0 hidden layers
Epoch 1/10: Train Loss=0.2436, Val Loss=0.1508, Train Acc=0.9262, Val Acc=0.9563
Epoch 2/10: Train Loss=0.1367, Val Loss=0.1413, Train Acc=0.9587, Val Acc=0.9592
Epoch 3/10: Train Loss=0.1098, Val Loss=0.1467, Train Acc=0.9674, Val Acc=0.9607
Epoch 4/10: Train Loss=0.1022, Val Loss=0.1721, Train Acc=0.9695, Val Acc=0.9606
Epoch 5/10: Train Loss=0.0912, Val Loss=0.1964, Train Acc=0.9745, Val Acc=0.9582
Epoch 6/10: Train Loss=0.0875, Val Loss=0.2064, Train Acc=0.9764, Val Acc=0.9616
Epoch 7/10: Train Loss=0.0842, Val Loss=0.1830, Train Acc=0.9782, Val Acc=0.9619
Epoch 8/10: Train Loss=0.0726, Val Loss=0.2015, Train Acc=0.9798, Val Acc=0.9665
Epoch 9/10: Train Loss=0.0767, Val Loss=0.2328, Train Acc=0.9803, Val Acc=0.9604
Epoch 10/10: Train Loss=0.0783, Val Loss=0.2290, Train Acc=0.9801, Val Acc=0.9663
```



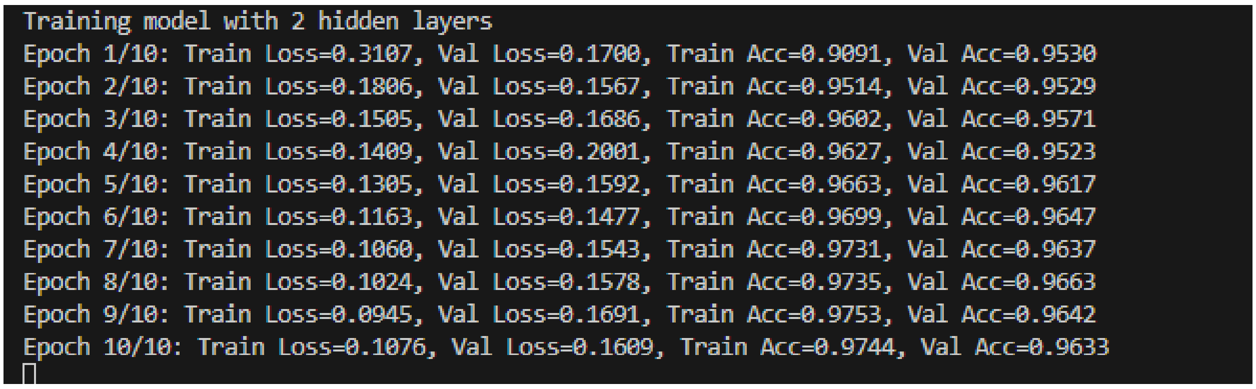
Hidden Layers (1):



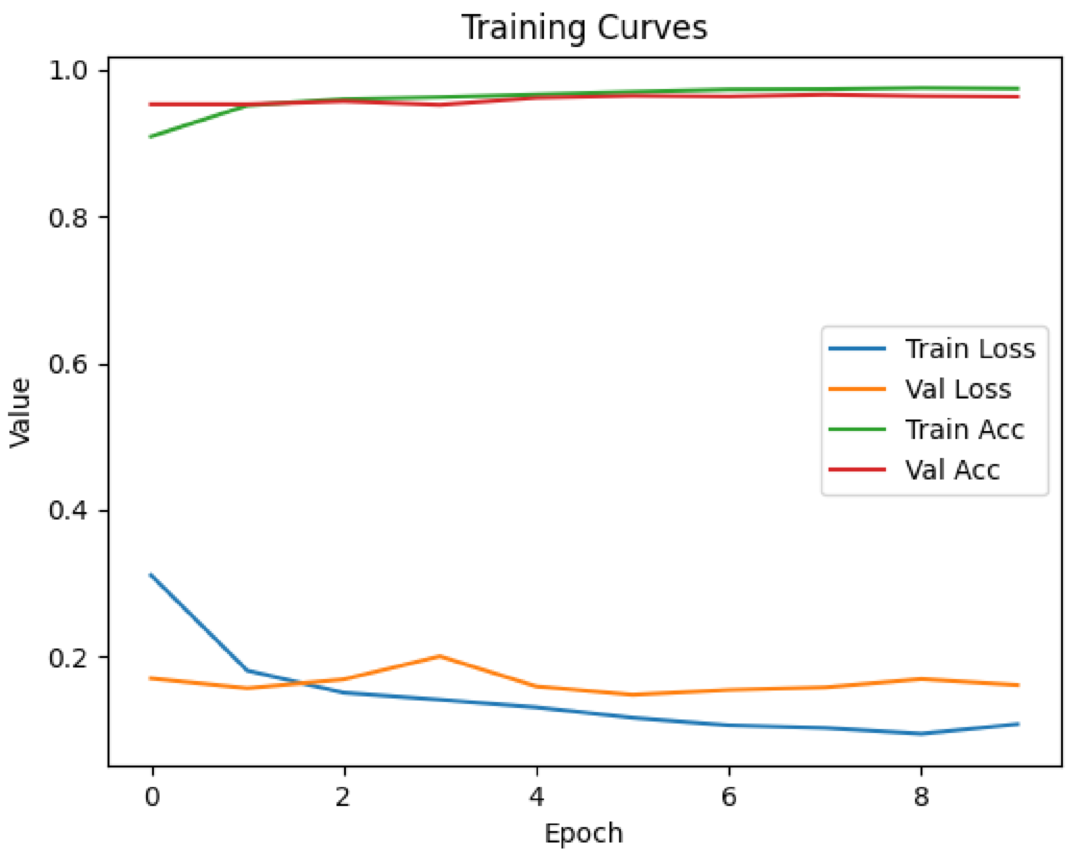
```
Training model with 1 hidden layers
Epoch 1/10: Train Loss=0.2668, Val Loss=0.1942, Train Acc=0.9197, Val Acc=0.9450
Epoch 2/10: Train Loss=0.1589, Val Loss=0.1528, Train Acc=0.9541, Val Acc=0.9564
Epoch 3/10: Train Loss=0.1372, Val Loss=0.1514, Train Acc=0.9620, Val Acc=0.9573
Epoch 4/10: Train Loss=0.1185, Val Loss=0.1747, Train Acc=0.9662, Val Acc=0.9525
Epoch 5/10: Train Loss=0.1167, Val Loss=0.1753, Train Acc=0.9684, Val Acc=0.9610
Epoch 6/10: Train Loss=0.1059, Val Loss=0.1700, Train Acc=0.9714, Val Acc=0.9623
Epoch 7/10: Train Loss=0.0979, Val Loss=0.1730, Train Acc=0.9736, Val Acc=0.9583
Epoch 8/10: Train Loss=0.0971, Val Loss=0.2298, Train Acc=0.9736, Val Acc=0.9537
Epoch 9/10: Train Loss=0.0988, Val Loss=0.2012, Train Acc=0.9748, Val Acc=0.9617
Epoch 10/10: Train Loss=0.0824, Val Loss=0.1892, Train Acc=0.9780, Val Acc=0.9654
```



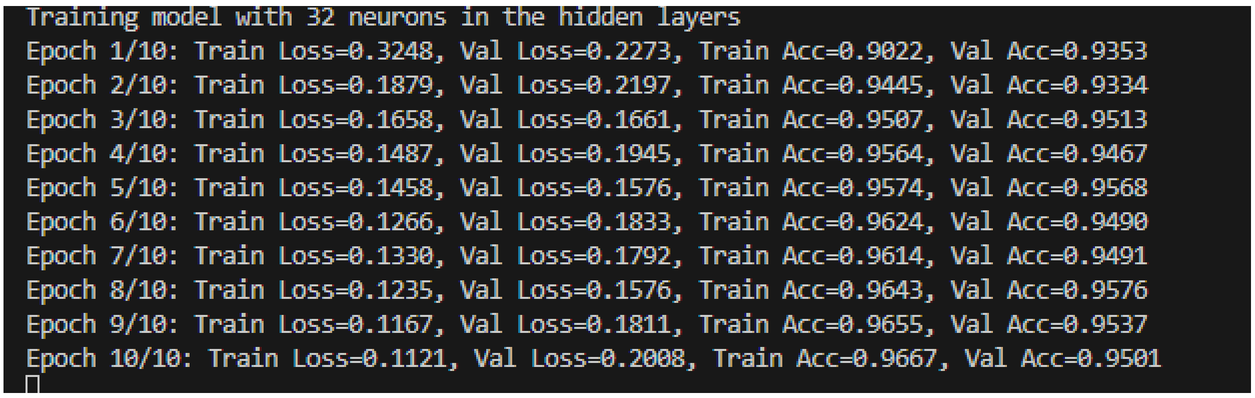
Hidden Layers (2):



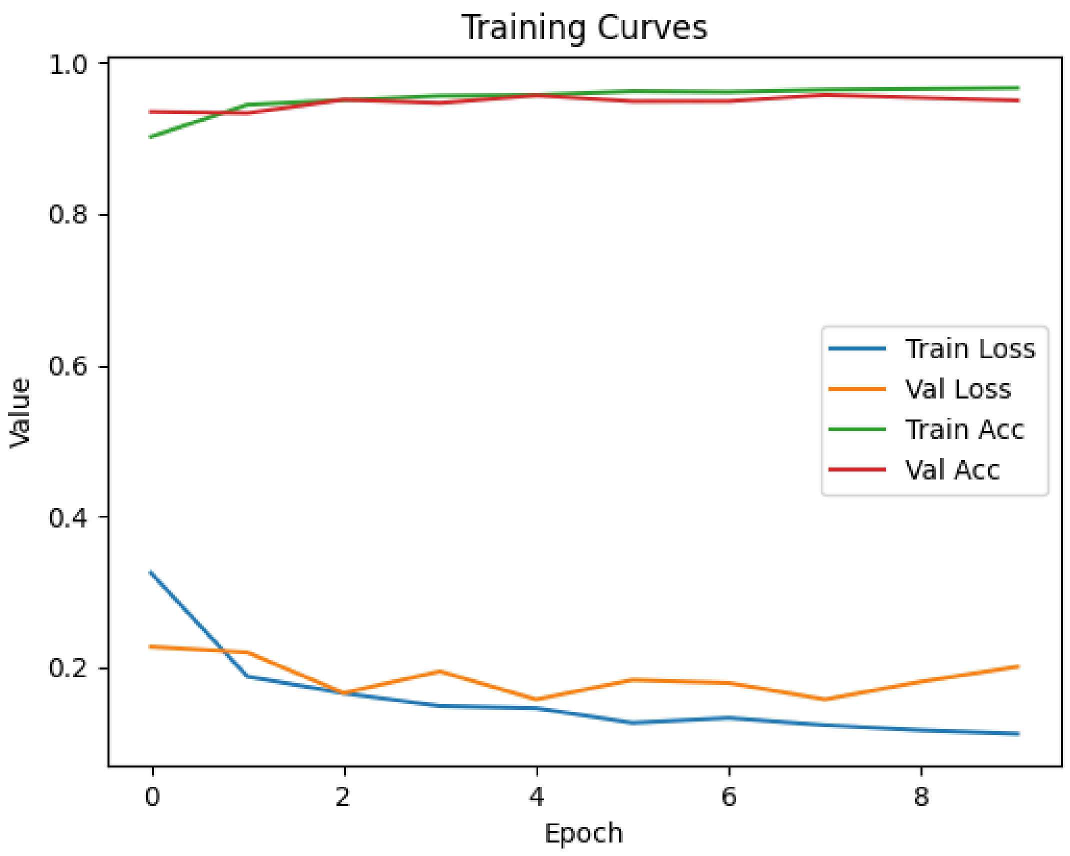
```
Training model with 2 hidden layers
Epoch 1/10: Train Loss=0.3107, Val Loss=0.1700, Train Acc=0.9091, Val Acc=0.9530
Epoch 2/10: Train Loss=0.1806, Val Loss=0.1567, Train Acc=0.9514, Val Acc=0.9529
Epoch 3/10: Train Loss=0.1505, Val Loss=0.1686, Train Acc=0.9602, Val Acc=0.9571
Epoch 4/10: Train Loss=0.1409, Val Loss=0.2001, Train Acc=0.9627, Val Acc=0.9523
Epoch 5/10: Train Loss=0.1305, Val Loss=0.1592, Train Acc=0.9663, Val Acc=0.9617
Epoch 6/10: Train Loss=0.1163, Val Loss=0.1477, Train Acc=0.9699, Val Acc=0.9647
Epoch 7/10: Train Loss=0.1060, Val Loss=0.1543, Train Acc=0.9731, Val Acc=0.9637
Epoch 8/10: Train Loss=0.1024, Val Loss=0.1578, Train Acc=0.9735, Val Acc=0.9663
Epoch 9/10: Train Loss=0.0945, Val Loss=0.1691, Train Acc=0.9753, Val Acc=0.9642
Epoch 10/10: Train Loss=0.1076, Val Loss=0.1609, Train Acc=0.9744, Val Acc=0.9633
```



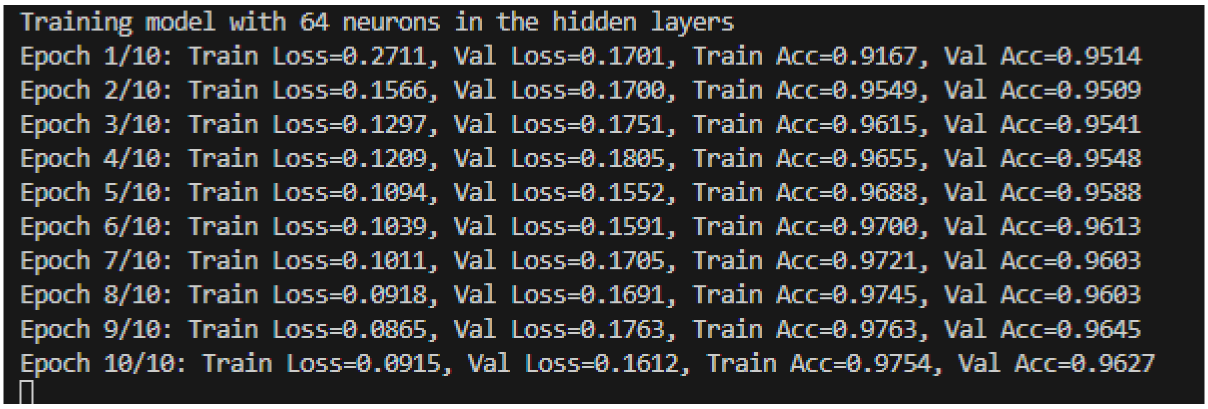
Neurons in hidden layers (32):



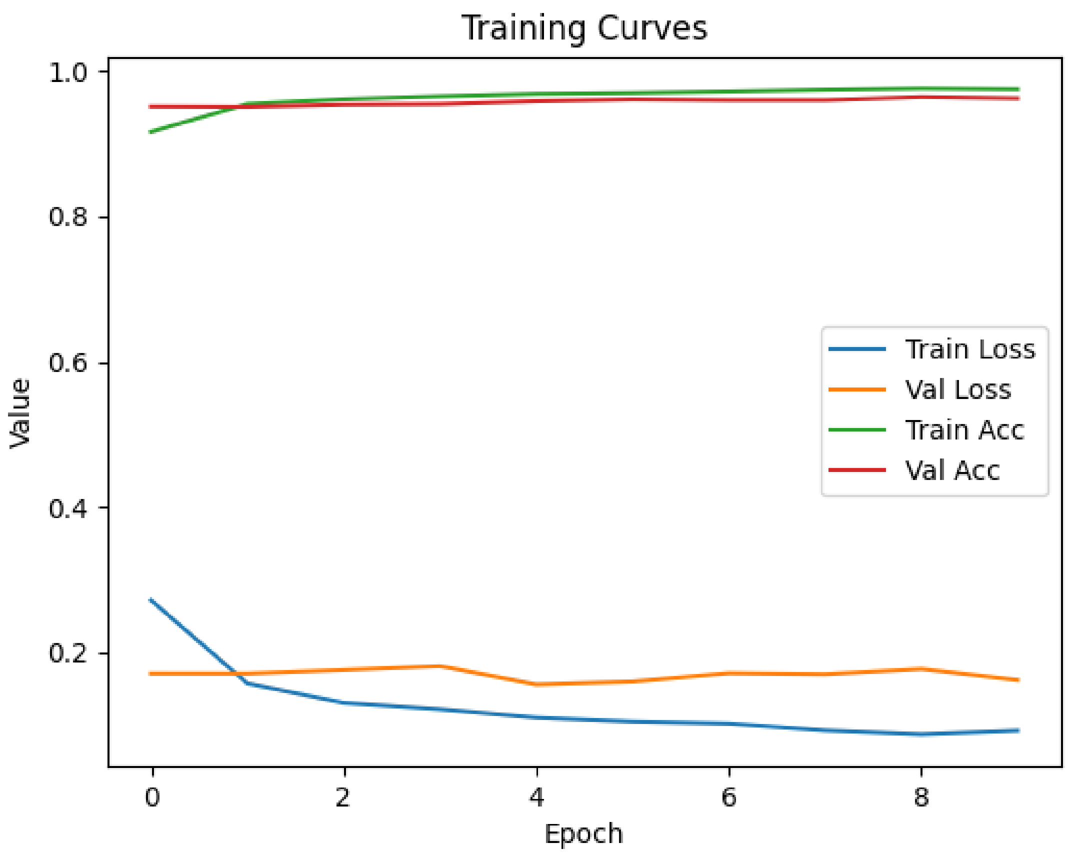
```
Training model with 32 neurons in the hidden layers
Epoch 1/10: Train Loss=0.3248, Val Loss=0.2273, Train Acc=0.9022, Val Acc=0.9353
Epoch 2/10: Train Loss=0.1879, Val Loss=0.2197, Train Acc=0.9445, Val Acc=0.9334
Epoch 3/10: Train Loss=0.1658, Val Loss=0.1661, Train Acc=0.9507, Val Acc=0.9513
Epoch 4/10: Train Loss=0.1487, Val Loss=0.1945, Train Acc=0.9564, Val Acc=0.9467
Epoch 5/10: Train Loss=0.1458, Val Loss=0.1576, Train Acc=0.9574, Val Acc=0.9568
Epoch 6/10: Train Loss=0.1266, Val Loss=0.1833, Train Acc=0.9624, Val Acc=0.9490
Epoch 7/10: Train Loss=0.1330, Val Loss=0.1792, Train Acc=0.9614, Val Acc=0.9491
Epoch 8/10: Train Loss=0.1235, Val Loss=0.1576, Train Acc=0.9643, Val Acc=0.9576
Epoch 9/10: Train Loss=0.1167, Val Loss=0.1811, Train Acc=0.9655, Val Acc=0.9537
Epoch 10/10: Train Loss=0.1121, Val Loss=0.2008, Train Acc=0.9667, Val Acc=0.9501
```



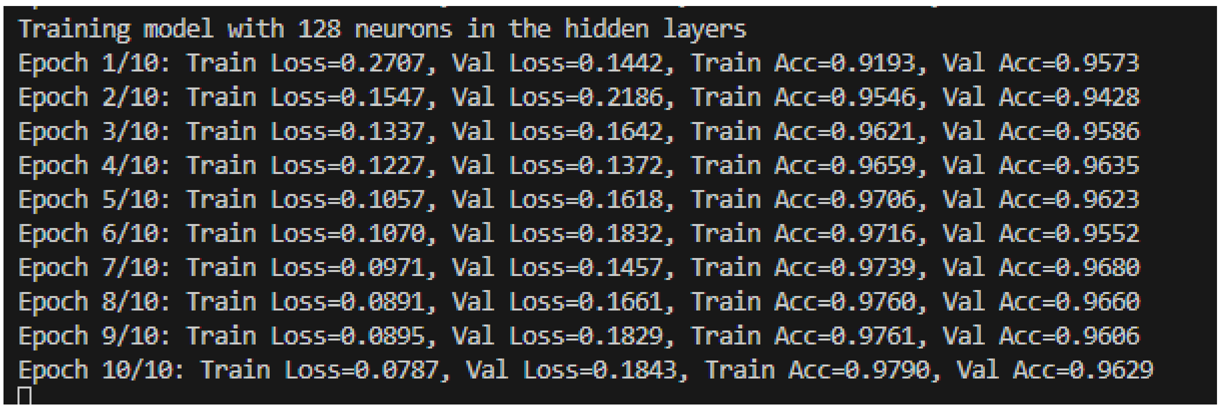
Neurons in hidden layers (64):



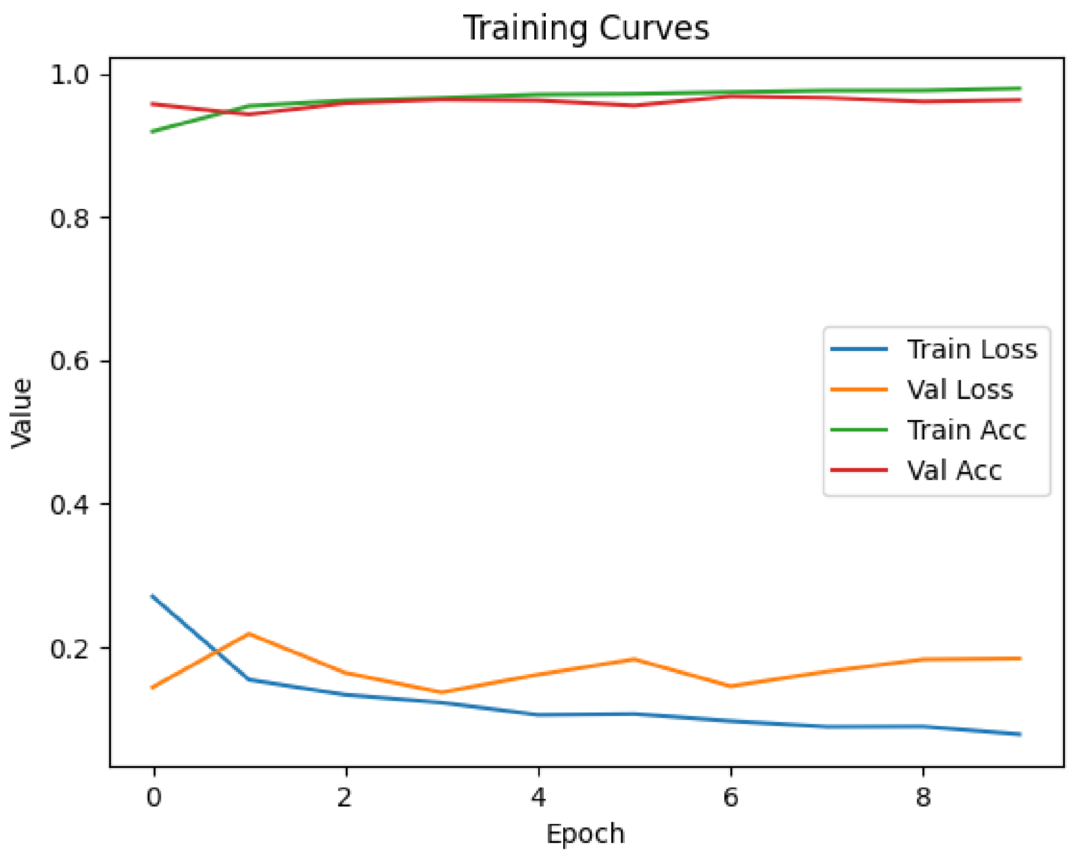
```
Training model with 64 neurons in the hidden layers
Epoch 1/10: Train Loss=0.2711, Val Loss=0.1701, Train Acc=0.9167, Val Acc=0.9514
Epoch 2/10: Train Loss=0.1566, Val Loss=0.1700, Train Acc=0.9549, Val Acc=0.9509
Epoch 3/10: Train Loss=0.1297, Val Loss=0.1751, Train Acc=0.9615, Val Acc=0.9541
Epoch 4/10: Train Loss=0.1209, Val Loss=0.1805, Train Acc=0.9655, Val Acc=0.9548
Epoch 5/10: Train Loss=0.1094, Val Loss=0.1552, Train Acc=0.9688, Val Acc=0.9588
Epoch 6/10: Train Loss=0.1039, Val Loss=0.1591, Train Acc=0.9700, Val Acc=0.9613
Epoch 7/10: Train Loss=0.1011, Val Loss=0.1705, Train Acc=0.9721, Val Acc=0.9603
Epoch 8/10: Train Loss=0.0918, Val Loss=0.1691, Train Acc=0.9745, Val Acc=0.9603
Epoch 9/10: Train Loss=0.0865, Val Loss=0.1763, Train Acc=0.9763, Val Acc=0.9645
Epoch 10/10: Train Loss=0.0915, Val Loss=0.1612, Train Acc=0.9754, Val Acc=0.9627
```



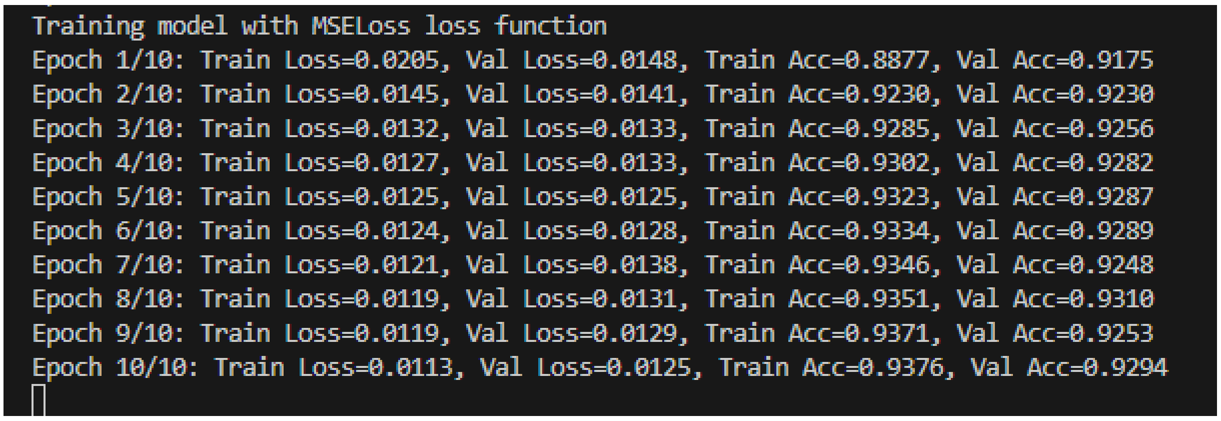
Neurons in hidden layers (128):



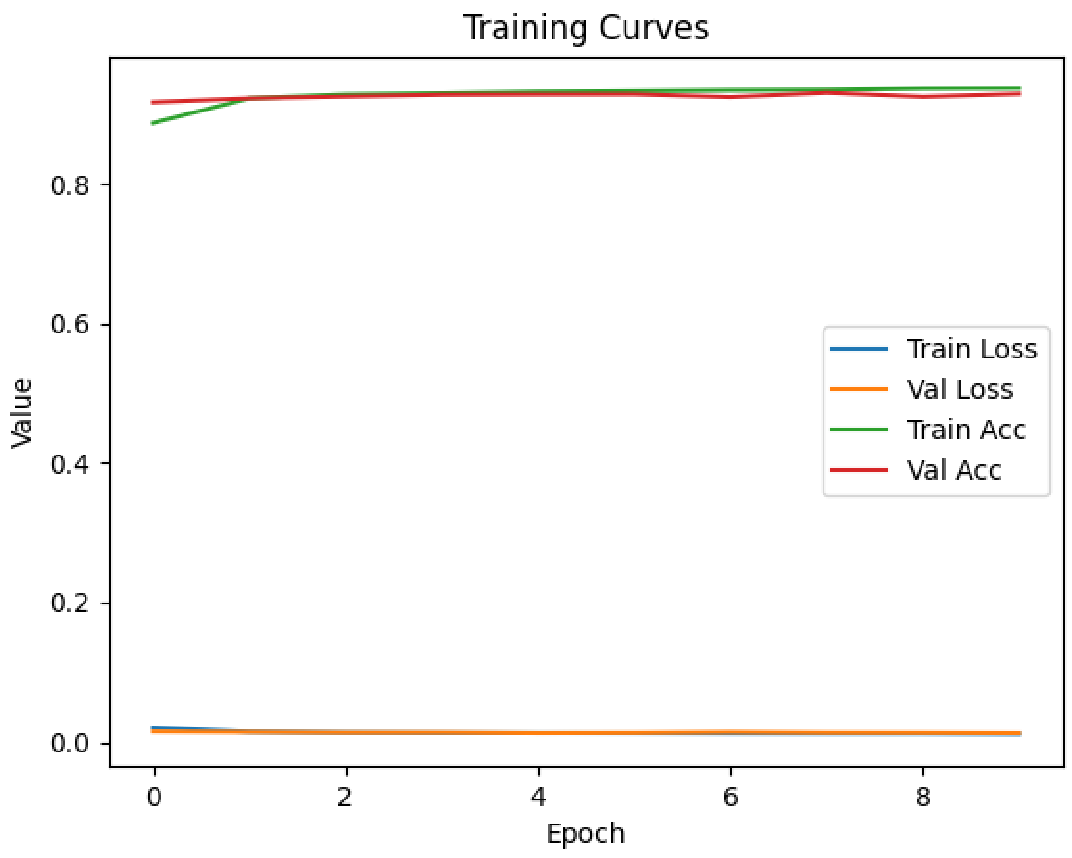
```
Training model with 128 neurons in the hidden layers
Epoch 1/10: Train Loss=0.2707, Val Loss=0.1442, Train Acc=0.9193, Val Acc=0.9573
Epoch 2/10: Train Loss=0.1547, Val Loss=0.2186, Train Acc=0.9546, Val Acc=0.9428
Epoch 3/10: Train Loss=0.1337, Val Loss=0.1642, Train Acc=0.9621, Val Acc=0.9586
Epoch 4/10: Train Loss=0.1227, Val Loss=0.1372, Train Acc=0.9659, Val Acc=0.9635
Epoch 5/10: Train Loss=0.1057, Val Loss=0.1618, Train Acc=0.9706, Val Acc=0.9623
Epoch 6/10: Train Loss=0.1070, Val Loss=0.1832, Train Acc=0.9716, Val Acc=0.9552
Epoch 7/10: Train Loss=0.0971, Val Loss=0.1457, Train Acc=0.9739, Val Acc=0.9680
Epoch 8/10: Train Loss=0.0891, Val Loss=0.1661, Train Acc=0.9760, Val Acc=0.9660
Epoch 9/10: Train Loss=0.0895, Val Loss=0.1829, Train Acc=0.9761, Val Acc=0.9606
Epoch 10/10: Train Loss=0.0787, Val Loss=0.1843, Train Acc=0.9790, Val Acc=0.9629
```



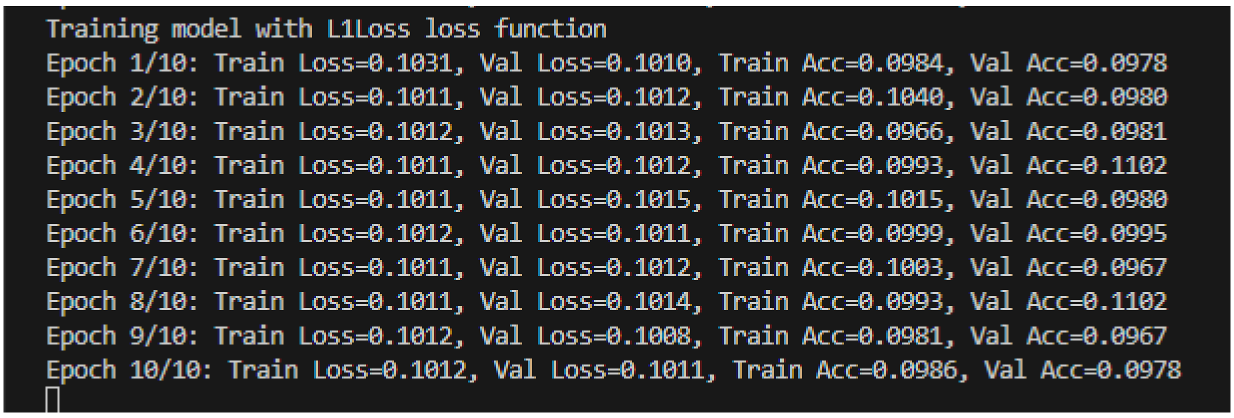
MSELoss loss function:



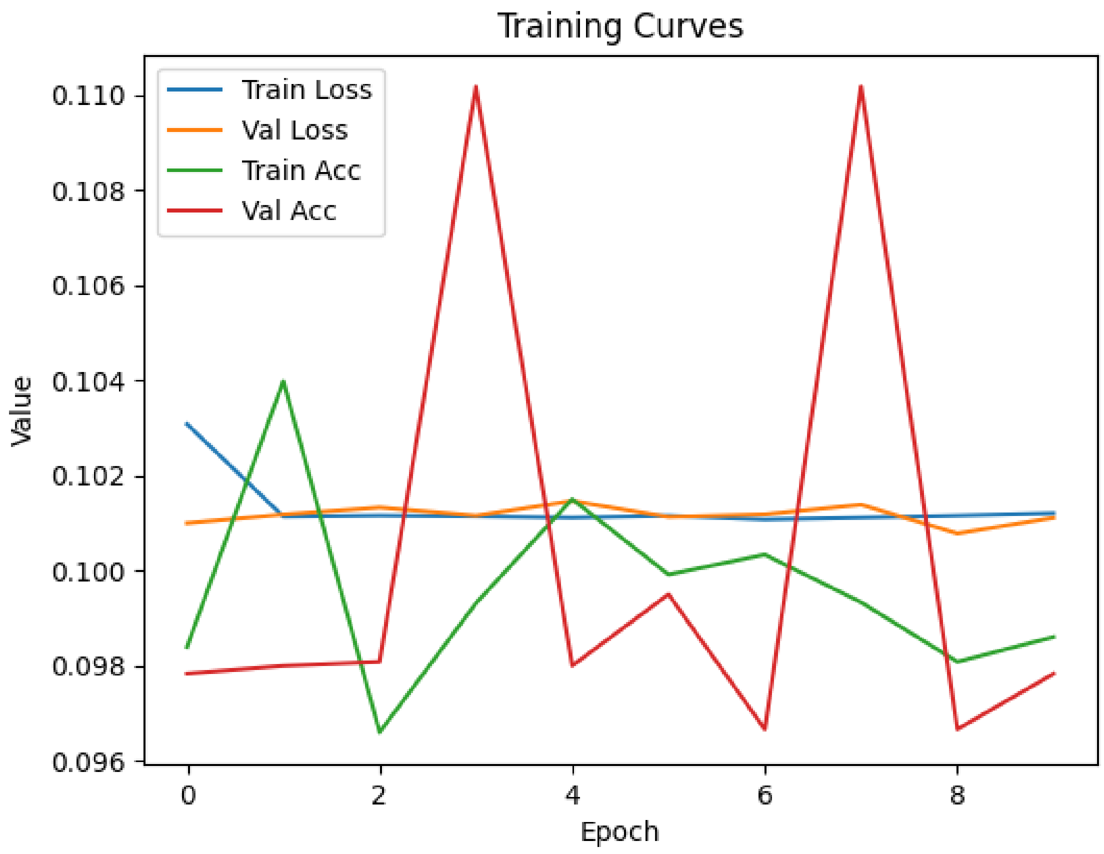
```
Training model with MSELoss loss function
Epoch 1/10: Train Loss=0.0205, Val Loss=0.0148, Train Acc=0.8877, Val Acc=0.9175
Epoch 2/10: Train Loss=0.0145, Val Loss=0.0141, Train Acc=0.9230, Val Acc=0.9230
Epoch 3/10: Train Loss=0.0132, Val Loss=0.0133, Train Acc=0.9285, Val Acc=0.9256
Epoch 4/10: Train Loss=0.0127, Val Loss=0.0133, Train Acc=0.9302, Val Acc=0.9282
Epoch 5/10: Train Loss=0.0125, Val Loss=0.0125, Train Acc=0.9323, Val Acc=0.9287
Epoch 6/10: Train Loss=0.0124, Val Loss=0.0128, Train Acc=0.9334, Val Acc=0.9289
Epoch 7/10: Train Loss=0.0121, Val Loss=0.0138, Train Acc=0.9346, Val Acc=0.9248
Epoch 8/10: Train Loss=0.0119, Val Loss=0.0131, Train Acc=0.9351, Val Acc=0.9310
Epoch 9/10: Train Loss=0.0119, Val Loss=0.0129, Train Acc=0.9371, Val Acc=0.9253
Epoch 10/10: Train Loss=0.0113, Val Loss=0.0125, Train Acc=0.9376, Val Acc=0.9294
```



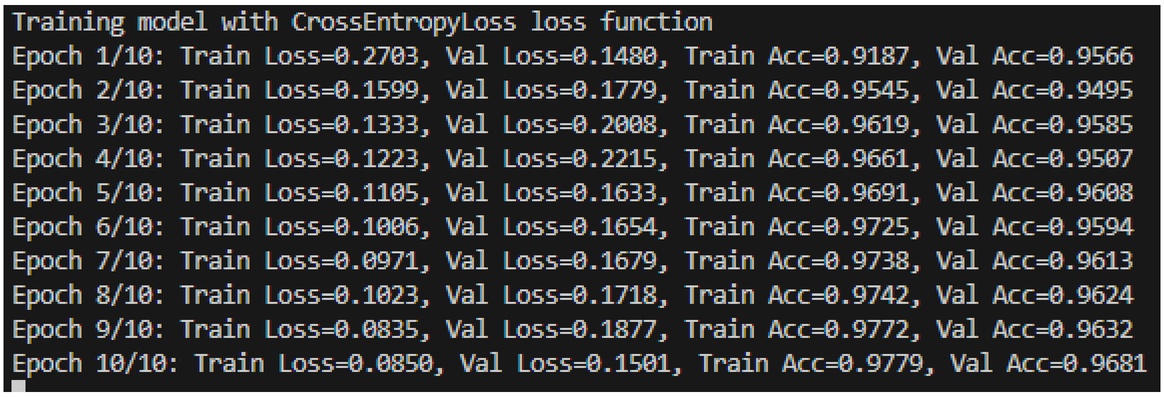
L1Loss loss function:



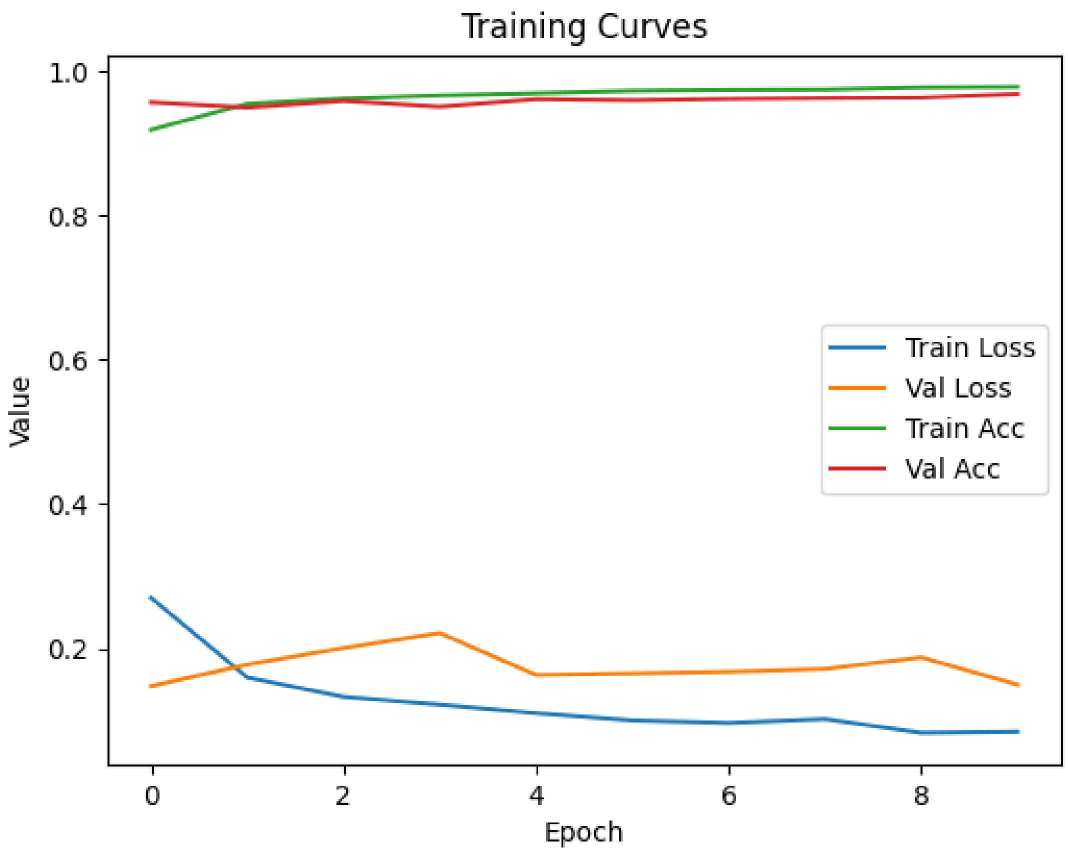
```
Training model with L1Loss loss function
Epoch 1/10: Train Loss=0.1031, Val Loss=0.1010, Train Acc=0.0984, Val Acc=0.0978
Epoch 2/10: Train Loss=0.1011, Val Loss=0.1012, Train Acc=0.1040, Val Acc=0.0980
Epoch 3/10: Train Loss=0.1012, Val Loss=0.1013, Train Acc=0.0966, Val Acc=0.0981
Epoch 4/10: Train Loss=0.1011, Val Loss=0.1012, Train Acc=0.0993, Val Acc=0.1102
Epoch 5/10: Train Loss=0.1011, Val Loss=0.1015, Train Acc=0.1015, Val Acc=0.0980
Epoch 6/10: Train Loss=0.1012, Val Loss=0.1011, Train Acc=0.0999, Val Acc=0.0995
Epoch 7/10: Train Loss=0.1011, Val Loss=0.1012, Train Acc=0.1003, Val Acc=0.0967
Epoch 8/10: Train Loss=0.1011, Val Loss=0.1014, Train Acc=0.0993, Val Acc=0.1102
Epoch 9/10: Train Loss=0.1012, Val Loss=0.1008, Train Acc=0.0981, Val Acc=0.0967
Epoch 10/10: Train Loss=0.1012, Val Loss=0.1011, Train Acc=0.0986, Val Acc=0.0978
```



CrossEntropyLoss loss function:



```
Training model with CrossEntropyLoss loss function
Epoch 1/10: Train Loss=0.2703, Val Loss=0.1480, Train Acc=0.9187, Val Acc=0.9566
Epoch 2/10: Train Loss=0.1599, Val Loss=0.1779, Train Acc=0.9545, Val Acc=0.9495
Epoch 3/10: Train Loss=0.1333, Val Loss=0.2008, Train Acc=0.9619, Val Acc=0.9585
Epoch 4/10: Train Loss=0.1223, Val Loss=0.2215, Train Acc=0.9661, Val Acc=0.9507
Epoch 5/10: Train Loss=0.1105, Val Loss=0.1633, Train Acc=0.9691, Val Acc=0.9608
Epoch 6/10: Train Loss=0.1006, Val Loss=0.1654, Train Acc=0.9725, Val Acc=0.9594
Epoch 7/10: Train Loss=0.0971, Val Loss=0.1679, Train Acc=0.9738, Val Acc=0.9613
Epoch 8/10: Train Loss=0.1023, Val Loss=0.1718, Train Acc=0.9742, Val Acc=0.9624
Epoch 9/10: Train Loss=0.0835, Val Loss=0.1877, Train Acc=0.9772, Val Acc=0.9632
Epoch 10/10: Train Loss=0.0850, Val Loss=0.1501, Train Acc=0.9779, Val Acc=0.9681
```



What could be improved:

We can add a time count to see how much time is needed to complete the training, also we could implement the script using “tikerflow” because the visualization will be better, less code and performance is faster.

Conclusion:

Experiment 1: Varying Learning Rate

- The learning rate has a significant impact on model training.
- A learning rate of 0.001 results in slow convergence and possibly suboptimal performance.
- A learning rate of 0.1 may lead to unstable training or overshooting the optimal solution.
- A learning rate of 0.01 tends to provide a good balance between convergence speed and stability.

Experiment 2: Varying Mini-batch Size

- The mini-batch size affects the speed and stability of training.
- A smaller batch size, such as 16, may result in faster convergence but with higher variability in the training process.
- A larger batch size, such as 256, can provide more stable updates but at the cost of slower convergence.
- A batch size of 64 often strikes a reasonable balance between convergence speed and stability.

Experiment 3: Varying Number of Hidden Layers

- Adding more hidden layers generally increases the model's capacity to learn complex patterns.
- A model without hidden layers (linear model) performs poorly on complex tasks like MNIST.
- Adding one or two hidden layers improves the model's performance, but there may be diminishing returns beyond a certain number of layers.

Experiment 4: Change Width

- Increasing the width (number of neurons) in the hidden layers can increase the model's capacity.
- Models with wider hidden layers generally have better performance compared to narrower layers.
- However, excessively wide layers can lead to overfitting or increased computational cost.

Experiment 5: Change Loss Function

- The choice of loss function depends on the task and desired model behavior.
- CrossEntropyLoss is commonly used for multi-class classification tasks like MNIST and provides good performance.

- MSELoss and L1Loss are more suitable for regression tasks or scenarios where the predicted values are continuous.