Laboratory 5

Variant 5

Group 5

By Jan Szachno and Aleksandra Głogowska

---

1. Introduction

The task is to implement a multilayer perceptron for image classification. The neural network should be trained with the mini-batch gradient descent method, and the dataset should be split into training and validation sets. The main point of this task is to evaluate how various components and hyperparameters of a neural network and the training process affect the network's performance in terms of its ability to converge, the speed of convergence, and the final accuracy of the training and validation sets. Finally, there should be visualizations of the loss value for every learning step, as well as accuracy on the training and validation set after each epoch.

In our case, we are using the KMNIST dataset. We are evaluating at least 3 different numbers/values/types of learning rate, mini-batch size (including a batch containing only 1 example), number of hidden layers (including 0 hidden layers - a linear model), width (number of neurons in hidden layers), and optimizer type (e.g., SGD, SGD with momentum, Adam).

2. Algorithm Description

The algorithm is created to implement a multilayer perceptron (MLP) for image classification, which is a type of neural network architecture consisting of multiple layers of neurons. It is a feedforward neural network, meaning that information flows from the input layer through one or more hidden layers to the output layer without cycles or loops. Each layer comprises neurons (also called units or nodes) that perform computations on the input data. The neural network is trained using the mini-batch gradient descent method, which is an optimization algorithm used for training machine learning models, like neural networks. It updates model parameters by computing gradients on smaller subsets (mini-batches) of the training data, rather than the entire dataset.

In our implementation, we are starting by loading the KMNIST dataset, preprocessing it by converting images to tensors and normalizing them. Tensors are multi-dimensional arrays that store data. They are extensively used in PyTorch, a deep-learning framework, used in our code. Next, we are defining the neural network model architecture, the model consists of fully connected layers with ReLU activation functions. ReLU is a mathematical function that introduces non-linearity to the network, allowing it to learn complex patterns in the data. The KMNIST dataset is loaded and then split into training and validation sets.

We are setting the hyperparameters, which are configuration settings that govern the training process of machine learning models. They are distinct from the model's parameters, which are learned from the training data. Hyperparameters are set before the training begins and are not updated during training. In our case, these are learning rate, batch size, optimizer parameters, training ratio, loss function, and number of epochs. The learning rate determines the step size taken during gradient descent. The batch size determines the number of samples used in each training iteration. The optimizer determines the optimization algorithm used to update the model parameters during training. The training ratio determines the percentage of the dataset used for training. The rest will be used for the validation of the model's correctness. The loss function measures the difference between the model's predictions and the actual target values. Cross-Entropy Loss is used as the loss function, which is suitable for multi-class classification problems like the one in this code. The number

of epochs specifies how many times the entire dataset will be passed through the model during training.

The algorithm includes training and validating functions. The training function trains the neural network model for one epoch (one complete pass through the training dataset). It sets the model to training mode. The function iterates over the batches of training data. For each batch of input data, the function performs a forward pass through the model to obtain the predicted outputs. Generally, a forward pass refers to the process of propagating input data through the network to obtain the predicted output. During the forward pass, each network layer performs its computation and passes the result to the next layer until the final output is generated. Next, the loss between the predicted outputs and the actual targets is calculated using the specified loss function (criterion). Then, clearing the gradients of all optimized tensors before performing the backward pass. Gradients of the loss with respect to the model parameters are computed using backpropagation. Backward pass in neural networks is a process where after making predictions, we compare them to the correct answers. This tells us how wrong our predictions were. We then figure out how each parameter in the network contributed to those errors. We adjust each parameter to reduce the errors next time. This is done mathematically using the gradients calculated during the backward pass. After the backpropagation, in our code, the optimizer updates the model parameters based on the computed gradients. During the epoch, the total loss is accumulated. The total number of correct predictions is computed using the predicted values and compared with the actual targets. The function returns the average loss per batch and the accuracy over the entire training dataset. Accuracy is a metric used to evaluate the performance of a classification model. It measures the proportion of correctly classified samples out of the total number of samples.

The validation function starts with setting the model to the evaluation mode. The function iterates over the provided loader, which contains batches of validation data. For each batch of input data, the function performs a forward pass through the model using the model to obtain the predicted outputs. The specified loss function calculates the loss between the predicted outputs and the actual targets. During the validation epoch, the total loss is accumulated. Additionally, the total number of correct predictions is computed. This is done by comparing the predicted class indices with the actual target class indices. The number of correct predictions is then summed up across all batches. The function returns the average loss per batch and the accuracy over the entire validation dataset.

The training process iterates over a specified number of epochs, during which the model is trained on the training dataset using mini-batch gradient descent. At each epoch, the model's performance is evaluated on the validation dataset to assess its generalization capability. Key metrics such as loss and accuracy are monitored throughout the training process to track the model's progress and performance.

Finally, the training and validation metrics are visualized using plots, allowing for a comprehensive analysis of the model's learning dynamics and performance trends over epochs. This visualization aids in understanding how the model's loss decreases, and accuracy improves during training, as well as identify potential issues such as overfitting or underfitting.

3. Experiments

Control experiment

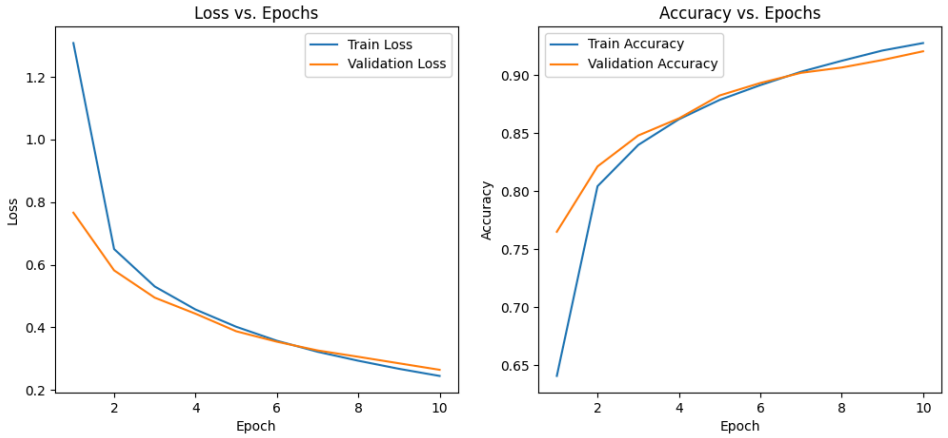
Use the KMNIST dataset.
Evaluate at least 3 different numbers/values/types of:
• Learning rate,
• Mini-batch size (including a batch containing only 1 example),
• Number of hidden layers (including 0 hidden layers - a linear model),
• Width (number of neurons in hidden layers),
• Optimizer type (e.g., SGD, SGD with momentum, Adam).

At first, we ran a “control group” run to make comparisons later on. We started it with the following config:

- learning rate = 0.01
- mini-batch size = 64
- num hidden layers = 1
- width of hidden layers = 512

```
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 10),
).to(device)
```

All further experiments will be run with the same configuration, with a single parameter changed so that we can isolate a single variable affecting the final result. For the “control group” we obtained the following results:



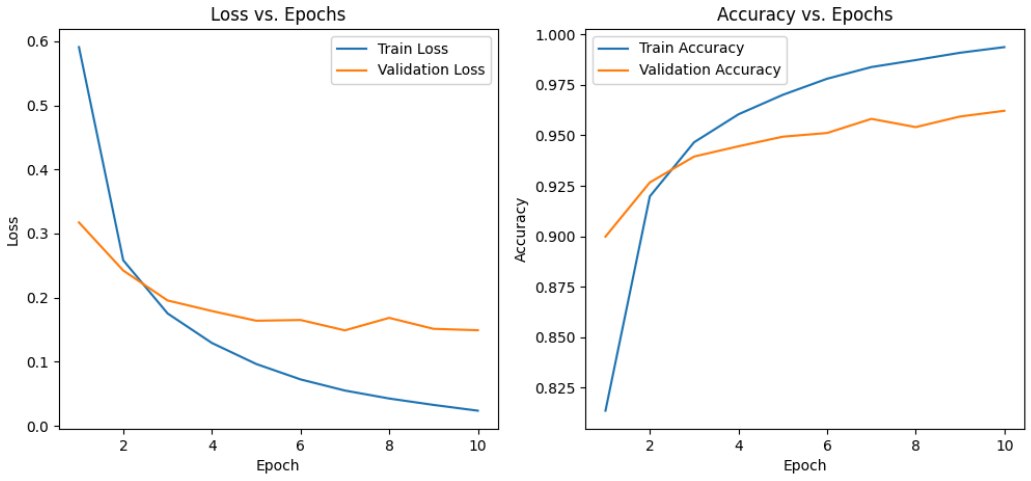
```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.3083, Train Acc: 0.6406
Val Loss: 0.7665, Val Acc: 0.7649
Epoch 2/10, Train Loss: 0.6504, Train Acc: 0.8042
Val Loss: 0.5823, Val Acc: 0.8213
Epoch 3/10, Train Loss: 0.5307, Train Acc: 0.8397
Val Loss: 0.4950, Val Acc: 0.8479
Epoch 4/10, Train Loss: 0.4573, Train Acc: 0.8619
Val Loss: 0.4439, Val Acc: 0.8627
Epoch 5/10, Train Loss: 0.4022, Train Acc: 0.8786
Val Loss: 0.3878, Val Acc: 0.8824
Epoch 6/10, Train Loss: 0.3574, Train Acc: 0.8913
Val Loss: 0.3543, Val Acc: 0.8932
Epoch 7/10, Train Loss: 0.3223, Train Acc: 0.9029
Val Loss: 0.3266, Val Acc: 0.9019
Epoch 8/10, Train Loss: 0.2936, Train Acc: 0.9123
Val Loss: 0.3062, Val Acc: 0.9065
Epoch 9/10, Train Loss: 0.2679, Train Acc: 0.9211
Val Loss: 0.2854, Val Acc: 0.9130
Epoch 10/10, Train Loss: 0.2450, Train Acc: 0.9276
Val Loss: 0.2647, Val Acc: 0.9205
```

Overall, the parameters in the control group were relatively well chosen, the model converged rather smoothly and quickly and was able to obtain good values of loss function and accuracy across both training and validation datasets.

Experimenting with learning rate

Learning rate = 0.1

In this experiment, we set the learning rate value to be equal to 0.1. These are the results we obtained:

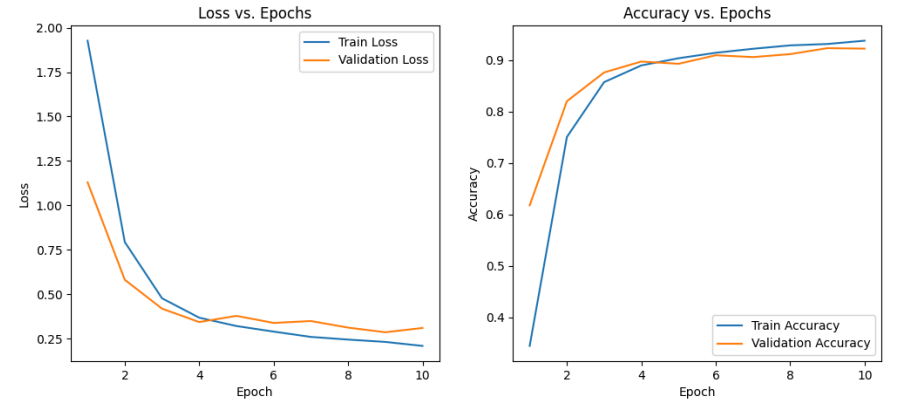


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.5909, Train Acc: 0.8136
Val Loss: 0.3173, Val Acc: 0.8998
Epoch 2/10, Train Loss: 0.2582, Train Acc: 0.9197
Val Loss: 0.2423, Val Acc: 0.9267
Epoch 3/10, Train Loss: 0.1753, Train Acc: 0.9466
Val Loss: 0.1956, Val Acc: 0.9395
Epoch 4/10, Train Loss: 0.1292, Train Acc: 0.9604
Val Loss: 0.1791, Val Acc: 0.9446
Epoch 5/10, Train Loss: 0.0964, Train Acc: 0.9701
Val Loss: 0.1640, Val Acc: 0.9493
Epoch 6/10, Train Loss: 0.0724, Train Acc: 0.9780
Val Loss: 0.1651, Val Acc: 0.9512
Epoch 7/10, Train Loss: 0.0550, Train Acc: 0.9838
Val Loss: 0.1489, Val Acc: 0.9582
Epoch 8/10, Train Loss: 0.0426, Train Acc: 0.9873
Val Loss: 0.1684, Val Acc: 0.9541
Epoch 9/10, Train Loss: 0.0326, Train Acc: 0.9909
Val Loss: 0.1513, Val Acc: 0.9593
Epoch 10/10, Train Loss: 0.0236, Train Acc: 0.9937
Val Loss: 0.1491, Val Acc: 0.9622
```

In this experiment, the learning rate was increased 10 times. This allowed the model to quickly decrease the value of loss and accuracy for the training dataset.The change of training loss and accuracy was not as rapid but we were still able to achieve good results, in fact better than in the control experiment.

Learning rate = 0.5

In this experiment, we set the learning rate value to be equal to 0.5. These are the results we obtained:

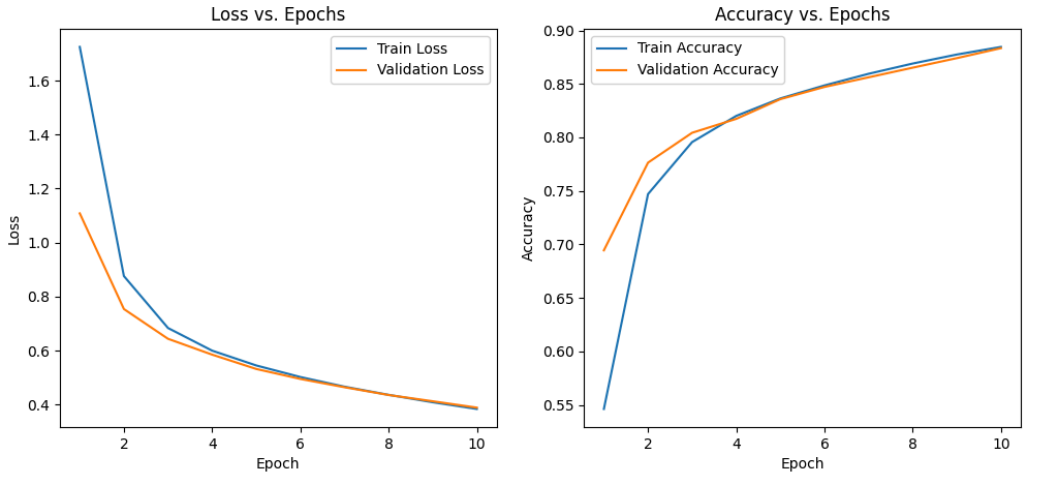


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.9267, Train Acc: 0.3449
Val Loss: 1.1297, Val Acc: 0.6180
Epoch 2/10, Train Loss: 0.7935, Train Acc: 0.7507
Val Loss: 0.5812, Val Acc: 0.8203
Epoch 3/10, Train Loss: 0.4770, Train Acc: 0.8573
Val Loss: 0.4188, Val Acc: 0.8761
Epoch 4/10, Train Loss: 0.3679, Train Acc: 0.8898
Val Loss: 0.3434, Val Acc: 0.8972
Epoch 5/10, Train Loss: 0.3210, Train Acc: 0.9037
Val Loss: 0.3782, Val Acc: 0.8929
Epoch 6/10, Train Loss: 0.2896, Train Acc: 0.9143
Val Loss: 0.3384, Val Acc: 0.9095
Epoch 7/10, Train Loss: 0.2598, Train Acc: 0.9221
Val Loss: 0.3491, Val Acc: 0.9059
Epoch 8/10, Train Loss: 0.2447, Train Acc: 0.9287
Val Loss: 0.3121, Val Acc: 0.9117
Epoch 9/10, Train Loss: 0.2316, Train Acc: 0.9314
Val Loss: 0.2860, Val Acc: 0.9233
Epoch 10/10, Train Loss: 0.2091, Train Acc: 0.9377
Val Loss: 0.3100, Val Acc: 0.9223
```

In this case, the chosen value of the learning rate proved to be too large. It allowed the model to move quickly in the beginning, however it made it difficult for the model to reach good values of loss and accuracy. This shows us that a larger learning rate allows for fast exploration but lacks ability for exploitation. This model performed worse than the control model.

Learning rate = 0.005

In contrast with the previous experiment, we tried a learning rate which is half of the control groups value. In this experiment, we set the learning rate value to be equal to 0.05. These are the results we obtained:



```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.7247, Train Acc: 0.5462
Val Loss: 1.1084, Val Acc: 0.6945
Epoch 2/10, Train Loss: 0.8762, Train Acc: 0.7470
Val Loss: 0.7541, Val Acc: 0.7764
Epoch 3/10, Train Loss: 0.6838, Train Acc: 0.7956
Val Loss: 0.6439, Val Acc: 0.8043
Epoch 4/10, Train Loss: 0.6000, Train Acc: 0.8201
Val Loss: 0.5853, Val Acc: 0.8173
Epoch 5/10, Train Loss: 0.5452, Train Acc: 0.8364
Val Loss: 0.5322, Val Acc: 0.8357
Epoch 6/10, Train Loss: 0.5024, Train Acc: 0.8486
Val Loss: 0.4952, Val Acc: 0.8472
Epoch 7/10, Train Loss: 0.4666, Train Acc: 0.8595
Val Loss: 0.4646, Val Acc: 0.8562
Epoch 8/10, Train Loss: 0.4364, Train Acc: 0.8690
Val Loss: 0.4358, Val Acc: 0.8652
Epoch 9/10, Train Loss: 0.4089, Train Acc: 0.8775
Val Loss: 0.4126, Val Acc: 0.8741
Epoch 10/10, Train Loss: 0.3840, Train Acc: 0.8846
Val Loss: 0.3890, Val Acc: 0.8834
```

In this case, we can see that the model was steadily approaching good results. However, the change in values of both loss and accuracy across epochs was smaller than in previous experiments. The model would probably approach similar values than previous experiments, however it would need more epochs to achieve such a result. From this we can conclude that too small learning rate makes the model needlessly time consuming without any added benefit.

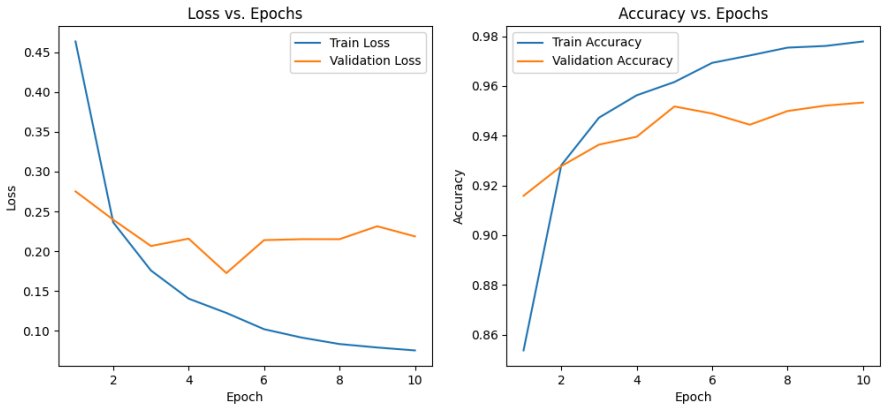
Conclusions for experimentation with the learning rate:

Choosing the learning rate allows us to choose the balance between exploration and exploitation. By tuning the value properly, we can create a model which will be able to obtain good results in a relatively short time. Too large a learning rate creates a model which learns fast however the results obtained from it are sub optimal. Too small learning rate makes the learning take significantly more time than it should.

Experimenting with batch size

batch size = 1

In this experiment, we set the batch size to be equal to 1. These are the results we obtained:

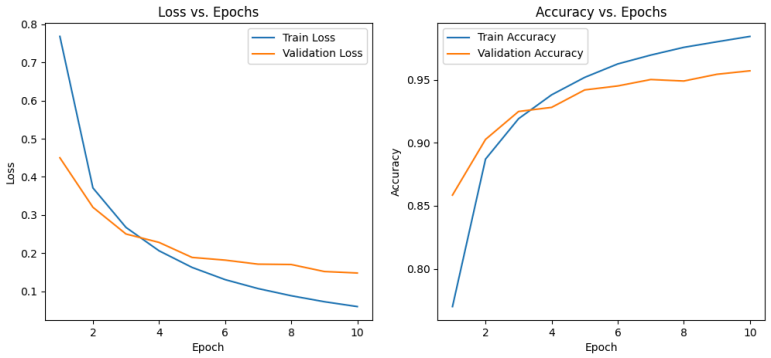


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.4634, Train Acc: 0.8536
Val Loss: 0.2751, Val Acc: 0.9158
Epoch 2/10, Train Loss: 0.2362, Train Acc: 0.9281
Val Loss: 0.2396, Val Acc: 0.9277
Epoch 3/10, Train Loss: 0.1758, Train Acc: 0.9473
Val Loss: 0.2065, Val Acc: 0.9364
Epoch 4/10, Train Loss: 0.1404, Train Acc: 0.9563
Val Loss: 0.2157, Val Acc: 0.9396
Epoch 5/10, Train Loss: 0.1225, Train Acc: 0.9616
Val Loss: 0.1725, Val Acc: 0.9517
Epoch 6/10, Train Loss: 0.1020, Train Acc: 0.9693
Val Loss: 0.2139, Val Acc: 0.9489
Epoch 7/10, Train Loss: 0.0915, Train Acc: 0.9723
Val Loss: 0.2151, Val Acc: 0.9444
Epoch 8/10, Train Loss: 0.0834, Train Acc: 0.9754
Val Loss: 0.2151, Val Acc: 0.9499
Epoch 9/10, Train Loss: 0.0790, Train Acc: 0.9761
Val Loss: 0.2313, Val Acc: 0.9521
Epoch 10/10, Train Loss: 0.0754, Train Acc: 0.9779
Val Loss: 0.2186, Val Acc: 0.9533
```

In such configuration, the model learns from the learning dataset sample by sample. This process took a lot of time. Even though the number of epochs was the same as in the control experiment, the training took significantly more time. Overall, we obtained better results than in the control case. However most of the gains are visible in the training loss and training accuracy. While the validation loss and accuracy is also better, the difference is relatively small and not worth the significant training time increase.

batch size = 16

In this experiment, we set the batch size to be equal to 16. These are the results we obtained:

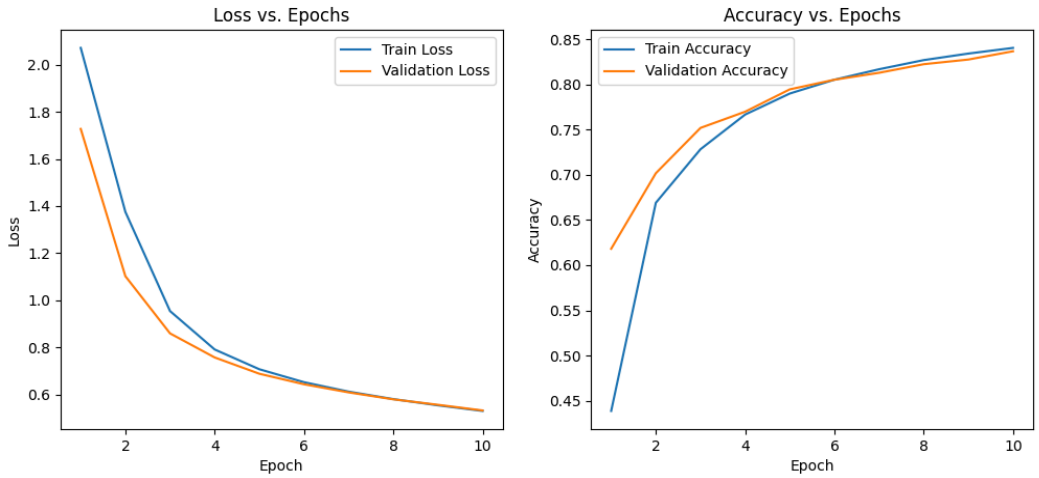


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.7681, Train Acc: 0.7700
Val Loss: 0.4501, Val Acc: 0.8585
Epoch 2/10, Train Loss: 0.3711, Train Acc: 0.8870
Val Loss: 0.3204, Val Acc: 0.9027
Epoch 3/10, Train Loss: 0.2677, Train Acc: 0.9191
Val Loss: 0.2504, Val Acc: 0.9248
Epoch 4/10, Train Loss: 0.2064, Train Acc: 0.9380
Val Loss: 0.2283, Val Acc: 0.9281
Epoch 5/10, Train Loss: 0.1628, Train Acc: 0.9519
Val Loss: 0.1890, Val Acc: 0.9419
Epoch 6/10, Train Loss: 0.1308, Train Acc: 0.9625
Val Loss: 0.1819, Val Acc: 0.9451
Epoch 7/10, Train Loss: 0.1073, Train Acc: 0.9696
Val Loss: 0.1713, Val Acc: 0.9502
Epoch 8/10, Train Loss: 0.0886, Train Acc: 0.9757
Val Loss: 0.1704, Val Acc: 0.9490
Epoch 9/10, Train Loss: 0.0729, Train Acc: 0.9801
Val Loss: 0.1521, Val Acc: 0.9543
Epoch 10/10, Train Loss: 0.0601, Train Acc: 0.9844
Val Loss: 0.1481, Val Acc: 0.9571
```

In this similarly to the previous one we obtained better results than in the control case. The training time was longer, but the difference was not as significant.

batch size = 256

In this experiment, we set the batch size to be equal to 256. These are the results we obtained:



This time the learning was significantly faster than in the previous cases. However we can instantly see that the model did not perform as well. Large batch size allowed the model to quickly go through the epochs, however processing too many samples at a time probably caused the model to miss important details hidden in the dataset and the adjustments to the model were too general.

```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 2.0719, Train Acc: 0.4387
Val Loss: 1.7276, Val Acc: 0.6182
Epoch 2/10, Train Loss: 1.3751, Train Acc: 0.6690
Val Loss: 1.1020, Val Acc: 0.7017
Epoch 3/10, Train Loss: 0.9539, Train Acc: 0.7283
Val Loss: 0.8591, Val Acc: 0.7519
Epoch 4/10, Train Loss: 0.7907, Train Acc: 0.7667
Val Loss: 0.7569, Val Acc: 0.7698
Epoch 5/10, Train Loss: 0.7068, Train Acc: 0.7900
Val Loss: 0.6877, Val Acc: 0.7946
Epoch 6/10, Train Loss: 0.6518, Train Acc: 0.8053
Val Loss: 0.6430, Val Acc: 0.8053
Epoch 7/10, Train Loss: 0.6118, Train Acc: 0.8169
Val Loss: 0.6084, Val Acc: 0.8129
Epoch 8/10, Train Loss: 0.5800, Train Acc: 0.8270
Val Loss: 0.5790, Val Acc: 0.8224
Epoch 9/10, Train Loss: 0.5534, Train Acc: 0.8342
Val Loss: 0.5558, Val Acc: 0.8276
Epoch 10/10, Train Loss: 0.5296, Train Acc: 0.8405
Val Loss: 0.5318, Val Acc: 0.8367
```

Conclusions for experimentation with the batch size:

Smaller values of batch size cause the model to train for a very long time. Also, noise from single samples can cause disruptions for the convergence. Processing samples in larger batches allows us to somehow ‘denoise’ the dataset. However when too large a value of the batch size is taken, the data fed to the model is too general and lacks important details. Selecting the appropriate batch size is crucial for balancing training efficiency, computational cost, and model generalization.

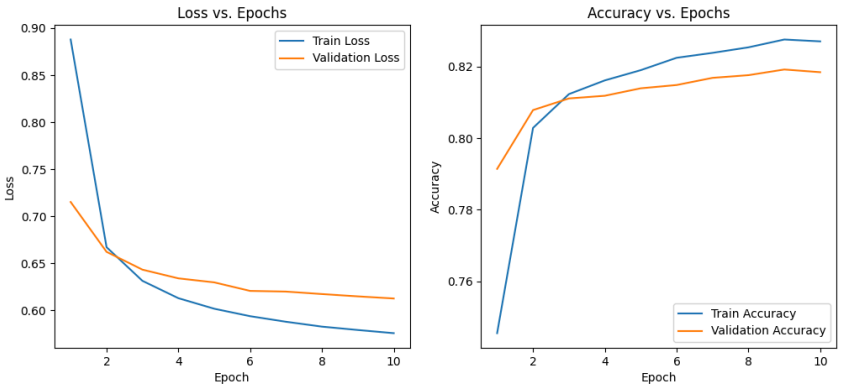
Experimenting with number of hidden layers

number of hidden layers = 0 (linear model)

In this experiment, we set the model to have no hidden layers (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 10),
).to(device)
```

These are the results we obtained:



```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.8879, Train Acc: 0.7455
Val Loss: 0.7151, Val Acc: 0.7914
Epoch 2/10, Train Loss: 0.6670, Train Acc: 0.8029
Val Loss: 0.6621, Val Acc: 0.8078
Epoch 3/10, Train Loss: 0.6312, Train Acc: 0.8123
Val Loss: 0.6431, Val Acc: 0.8111
Epoch 4/10, Train Loss: 0.6128, Train Acc: 0.8161
Val Loss: 0.6339, Val Acc: 0.8118
Epoch 5/10, Train Loss: 0.6015, Train Acc: 0.8190
Val Loss: 0.6296, Val Acc: 0.8139
Epoch 6/10, Train Loss: 0.5936, Train Acc: 0.8224
Val Loss: 0.6205, Val Acc: 0.8148
Epoch 7/10, Train Loss: 0.5877, Train Acc: 0.8238
Val Loss: 0.6198, Val Acc: 0.8168
Epoch 8/10, Train Loss: 0.5825, Train Acc: 0.8254
Val Loss: 0.6172, Val Acc: 0.8176
Epoch 9/10, Train Loss: 0.5789, Train Acc: 0.8275
Val Loss: 0.6147, Val Acc: 0.8192
Epoch 10/10, Train Loss: 0.5756, Train Acc: 0.8270
Val Loss: 0.6124, Val Acc: 0.8184
```

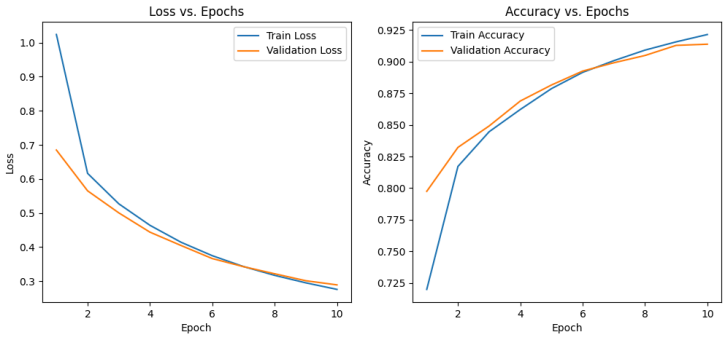
By looking at the results we can see that such a model does not perform well when compared to the control experiment. It learns rather slowly and is not able to provide satisfactory results. The amount of layers in this case is too small.

number of hidden layers = 1

In this experiment, we set the model to have 1 hidden layer (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 512),
    nn.ReLU(),
    nn.Linear(512, 10),
).to(device)
```

These are the results we obtained:



This model performed relatively well when compared to the control experiment. However this case has less layers. Thus, we can consider it better since the model is smaller while achieving similar results.

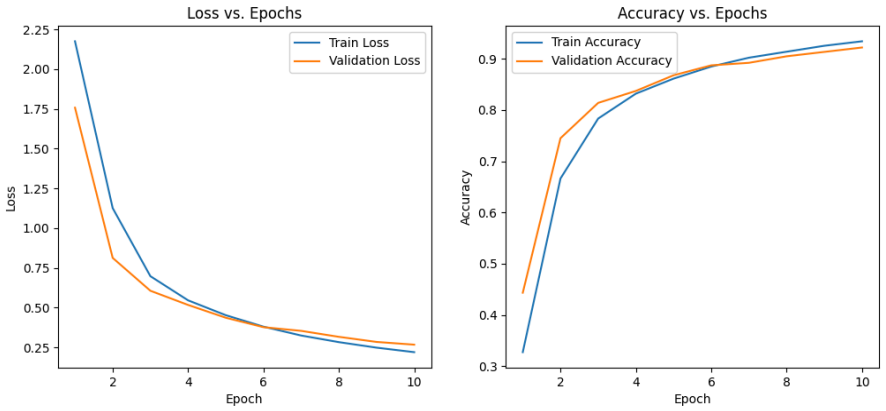
```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.0238, Train Acc: 0.7200
Val Loss: 0.6848, Val Acc: 0.7975
Epoch 2/10, Train Loss: 0.6164, Train Acc: 0.8172
Val Loss: 0.5652, Val Acc: 0.8323
Epoch 3/10, Train Loss: 0.5272, Train Acc: 0.8446
Val Loss: 0.5006, Val Acc: 0.8492
Epoch 4/10, Train Loss: 0.4640, Train Acc: 0.8622
Val Loss: 0.4439, Val Acc: 0.8689
Epoch 5/10, Train Loss: 0.4142, Train Acc: 0.8787
Val Loss: 0.4047, Val Acc: 0.8816
Epoch 6/10, Train Loss: 0.3748, Train Acc: 0.8916
Val Loss: 0.3664, Val Acc: 0.8926
Epoch 7/10, Train Loss: 0.3430, Train Acc: 0.9008
Val Loss: 0.3427, Val Acc: 0.8992
Epoch 8/10, Train Loss: 0.3171, Train Acc: 0.9092
Val Loss: 0.3218, Val Acc: 0.9049
Epoch 9/10, Train Loss: 0.2952, Train Acc: 0.9157
Val Loss: 0.3013, Val Acc: 0.9129
Epoch 10/10, Train Loss: 0.2759, Train Acc: 0.9215
Val Loss: 0.2892, Val Acc: 0.9138
```

number of hidden layers = 4

In this experiment, we set the model to have 4 hidden layers (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 512),
    nn.ReLU(),
    nn.Linear(512, 512),
    nn.ReLU(),
    nn.Linear(512, 512),
    nn.ReLU(),
    nn.Linear(512, 256),
    nn.ReLU(),
    nn.Linear(256, 10),
).to(device)
```

These are the results we obtained:



We can see that the model also performs similarly to the control experiment. So we can conclude that after reaching some critical point, adding more layers to the network does not result in better results. Larger model takes more time to train and requires more resources to

```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 2.1752, Train Acc: 0.3272
Val Loss: 1.7576, Val Acc: 0.4433
Epoch 2/10, Train Loss: 1.1258, Train Acc: 0.6664
Val Loss: 0.8128, Val Acc: 0.7451
Epoch 3/10, Train Loss: 0.6978, Train Acc: 0.7835
Val Loss: 0.6062, Val Acc: 0.8141
Epoch 4/10, Train Loss: 0.5460, Train Acc: 0.8321
Val Loss: 0.5170, Val Acc: 0.8374
Epoch 5/10, Train Loss: 0.4525, Train Acc: 0.8614
Val Loss: 0.4365, Val Acc: 0.8680
Epoch 6/10, Train Loss: 0.3804, Train Acc: 0.8850
Val Loss: 0.3772, Val Acc: 0.8874
Epoch 7/10, Train Loss: 0.3247, Train Acc: 0.9024
Val Loss: 0.3539, Val Acc: 0.8923
Epoch 8/10, Train Loss: 0.2831, Train Acc: 0.9140
Val Loss: 0.3164, Val Acc: 0.9051
Epoch 9/10, Train Loss: 0.2483, Train Acc: 0.9256
Val Loss: 0.2843, Val Acc: 0.9137
Epoch 10/10, Train Loss: 0.2200, Train Acc: 0.9344
Val Loss: 0.2672, Val Acc: 0.9223
```

Conclusions for experimentation with the number of hidden layers:

Larger amount of hidden layers in the model allows it to understand the data better and learn more of its intricacies. However, once the network is large enough to “fit” the given problem, increasing the number of layers does not result in better results. Training takes more time and running the model requires more computational resources without any added benefit.

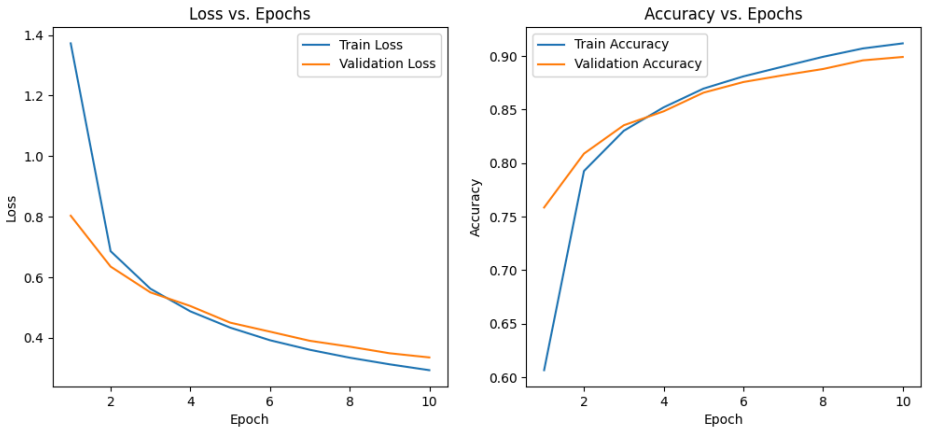
Experimenting with the width of the hidden layer

width of the hidden layer = 64

In this experiment, we set the hidden layers to have width equal to 64 (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 64),
    nn.ReLU(),
    nn.Linear(64, 64),
    nn.ReLU(),
    nn.Linear(64, 10),
).to(device)
```

These are the results we obtained:



```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.3726, Train Acc: 0.6067
Val Loss: 0.8035, Val Acc: 0.7586
Epoch 2/10, Train Loss: 0.6866, Train Acc: 0.7926
Val Loss: 0.6357, Val Acc: 0.8088
Epoch 3/10, Train Loss: 0.5624, Train Acc: 0.8302
Val Loss: 0.5504, Val Acc: 0.8353
Epoch 4/10, Train Loss: 0.4879, Train Acc: 0.8520
Val Loss: 0.5054, Val Acc: 0.8483
Epoch 5/10, Train Loss: 0.4341, Train Acc: 0.8695
Val Loss: 0.4505, Val Acc: 0.8658
Epoch 6/10, Train Loss: 0.3924, Train Acc: 0.8809
Val Loss: 0.4207, Val Acc: 0.8757
Epoch 7/10, Train Loss: 0.3609, Train Acc: 0.8901
Val Loss: 0.3906, Val Acc: 0.8820
Epoch 8/10, Train Loss: 0.3348, Train Acc: 0.8992
Val Loss: 0.3712, Val Acc: 0.8878
Epoch 9/10, Train Loss: 0.3129, Train Acc: 0.9070
Val Loss: 0.3495, Val Acc: 0.8959
Epoch 10/10, Train Loss: 0.2936, Train Acc: 0.9118
Val Loss: 0.3357, Val Acc: 0.8991
```

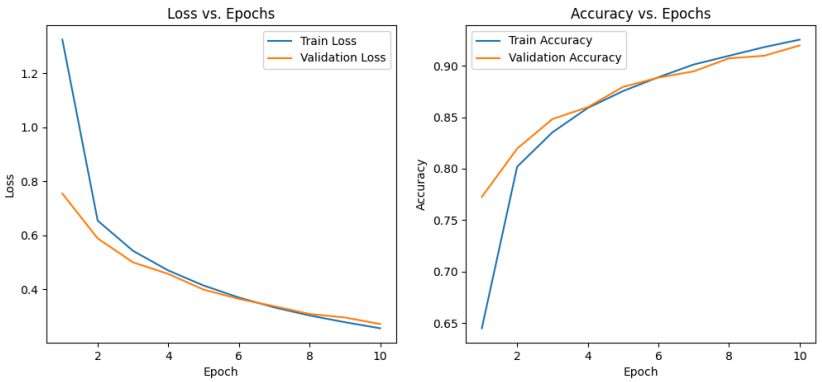
This model has a smaller width of the hidden layer than the control experiment. Although it performs worse than the control experiment, the difference is not that significant while the width is a lot smaller. From this we can conclude that optimal width of the layer lies somewhere between the control experiment and this experiment.

width of the hidden layer = 256

In this experiment, we set the hidden layers to have width equal to 256 (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 256),
    nn.ReLU(),
    nn.Linear(256, 256),
    nn.ReLU(),
    nn.Linear(256, 10),
).to(device)
```

These are the results we obtained:



```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.3250, Train Acc: 0.6449
Val Loss: 0.7544, Val Acc: 0.7726
Epoch 2/10, Train Loss: 0.6541, Train Acc: 0.8019
Val Loss: 0.5883, Val Acc: 0.8195
Epoch 3/10, Train Loss: 0.5423, Train Acc: 0.8353
Val Loss: 0.4996, Val Acc: 0.8482
Epoch 4/10, Train Loss: 0.4694, Train Acc: 0.8590
Val Loss: 0.4566, Val Acc: 0.8598
Epoch 5/10, Train Loss: 0.4136, Train Acc: 0.8754
Val Loss: 0.3985, Val Acc: 0.8795
Epoch 6/10, Train Loss: 0.3692, Train Acc: 0.8889
Val Loss: 0.3639, Val Acc: 0.8885
Epoch 7/10, Train Loss: 0.3326, Train Acc: 0.9012
Val Loss: 0.3368, Val Acc: 0.8946
Epoch 8/10, Train Loss: 0.3026, Train Acc: 0.9097
Val Loss: 0.3082, Val Acc: 0.9073
Epoch 9/10, Train Loss: 0.2777, Train Acc: 0.9181
Val Loss: 0.2951, Val Acc: 0.9097
Epoch 10/10, Train Loss: 0.2551, Train Acc: 0.9254
Val Loss: 0.2709, Val Acc: 0.9197
```

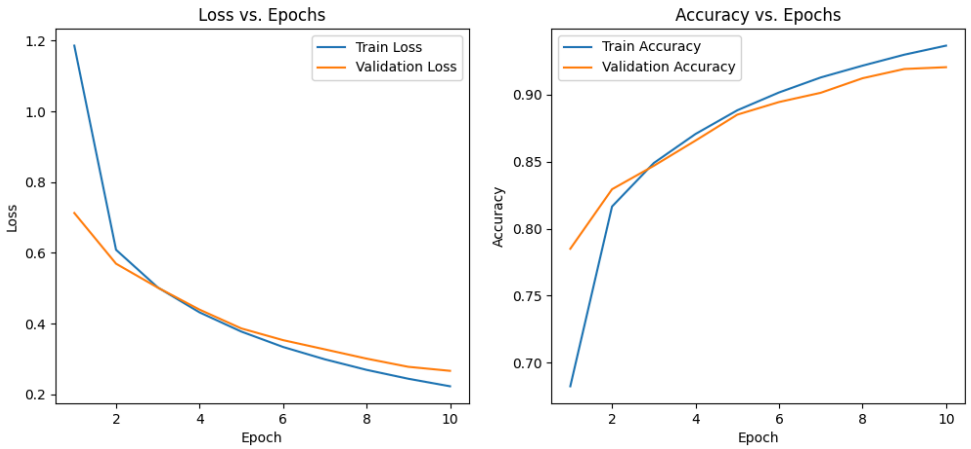
Even though the layer width is half of the value from the control experiment, the results are almost the same. We can consider this model better than the control group since it required less work to train with comparable results.

width of the hidden layer = 1024

In this experiment, we set the hidden layers to have width equal to 1024 (shown on photo).

```python
# Model
model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(28 * 28, 1024),
    nn.ReLU(),
    nn.Linear(1024, 1024),
    nn.ReLU(),
    nn.Linear(1024, 10),
).to(device)
```

These are the results we obtained:



```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 1.1858, Train Acc: 0.6824
Val Loss: 0.7126, Val Acc: 0.7850
Epoch 2/10, Train Loss: 0.6085, Train Acc: 0.8165
Val Loss: 0.5692, Val Acc: 0.8294
Epoch 3/10, Train Loss: 0.5021, Train Acc: 0.8490
Val Loss: 0.5016, Val Acc: 0.8469
Epoch 4/10, Train Loss: 0.4316, Train Acc: 0.8707
Val Loss: 0.4394, Val Acc: 0.8658
Epoch 5/10, Train Loss: 0.3773, Train Acc: 0.8883
Val Loss: 0.3863, Val Acc: 0.8851
Epoch 6/10, Train Loss: 0.3338, Train Acc: 0.9016
Val Loss: 0.3531, Val Acc: 0.8945
Epoch 7/10, Train Loss: 0.2989, Train Acc: 0.9129
Val Loss: 0.3271, Val Acc: 0.9014
Epoch 8/10, Train Loss: 0.2691, Train Acc: 0.9216
Val Loss: 0.3009, Val Acc: 0.9123
Epoch 9/10, Train Loss: 0.2440, Train Acc: 0.9298
Val Loss: 0.2777, Val Acc: 0.9192
Epoch 10/10, Train Loss: 0.2227, Train Acc: 0.9366
Val Loss: 0.2663, Val Acc: 0.9205
```

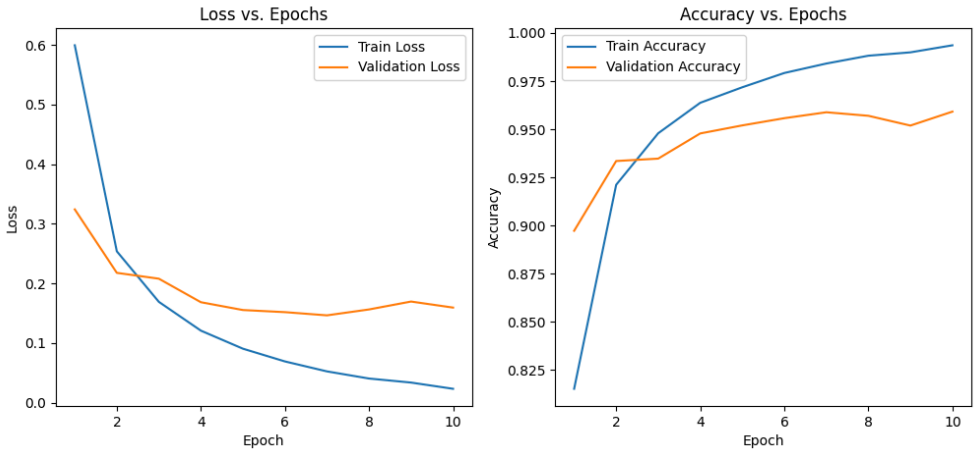
This model has basically the same results as the control model while the width is twice as big, so the added width did not result in any benefit for the learning.

Conclusions for experimentation with the width of hidden layers:
Similarly to the experiment with the number of hidden layers, the larger width of hidden layers in the model allows it to understand the data better and learn more of its intricacies. However, once making the layers wider does not result in better results. Training takes more time and running the model requires more computational resources without any added benefit.

Experimenting with the optimizer

Stochastic Gradient Descent (SGD) with momentum equal to 0.9

In this experiment, we change the optimizer to stochastic gradient descent with momentum equal to 0.9. In the control experiment we also used stochastic gradient descent but without momentum. These are the results that we obtained:

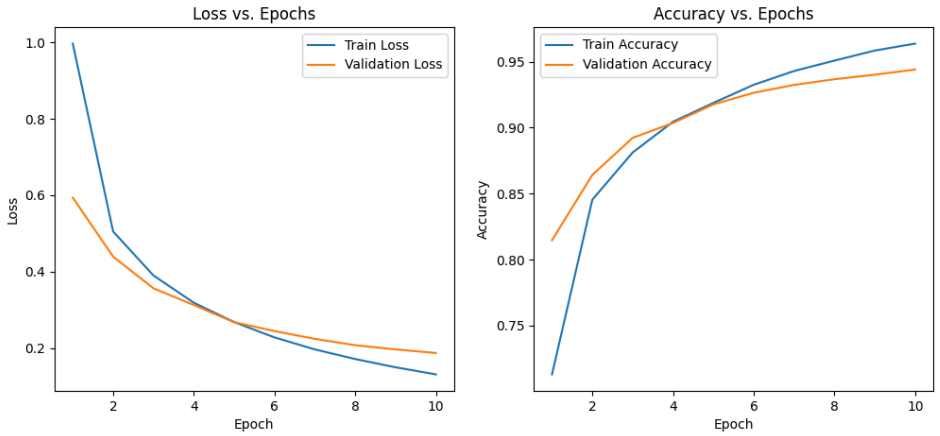


```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.5996, Train Acc: 0.8152
Val Loss: 0.3243, Val Acc: 0.8972
Epoch 2/10, Train Loss: 0.2540, Train Acc: 0.9211
Val Loss: 0.2178, Val Acc: 0.9335
Epoch 3/10, Train Loss: 0.1692, Train Acc: 0.9479
Val Loss: 0.2080, Val Acc: 0.9347
Epoch 4/10, Train Loss: 0.1209, Train Acc: 0.9637
Val Loss: 0.1686, Val Acc: 0.9478
Epoch 5/10, Train Loss: 0.0906, Train Acc: 0.9718
Val Loss: 0.1555, Val Acc: 0.9520
Epoch 6/10, Train Loss: 0.0692, Train Acc: 0.9792
Val Loss: 0.1519, Val Acc: 0.9557
Epoch 7/10, Train Loss: 0.0525, Train Acc: 0.9841
Val Loss: 0.1465, Val Acc: 0.9588
Epoch 8/10, Train Loss: 0.0406, Train Acc: 0.9882
Val Loss: 0.1564, Val Acc: 0.9570
Epoch 9/10, Train Loss: 0.0338, Train Acc: 0.9900
Val Loss: 0.1697, Val Acc: 0.9519
Epoch 10/10, Train Loss: 0.0233, Train Acc: 0.9936
Val Loss: 0.1596, Val Acc: 0.9592
```

While the rest of the network parameters remained the same, adding momentum to the optimizer allowed us to improve our results. It allowed the optimizer to escape local minima and reach better results without increasing the computational load.

Stochastic Gradient Descent (SGD) with momentum equal to 0.5

In this experiment, we change the optimizer to stochastic gradient descent with momentum equal to 0.5. In the control experiment we also used stochastic gradient descent but without momentum. These are the results that we obtained:

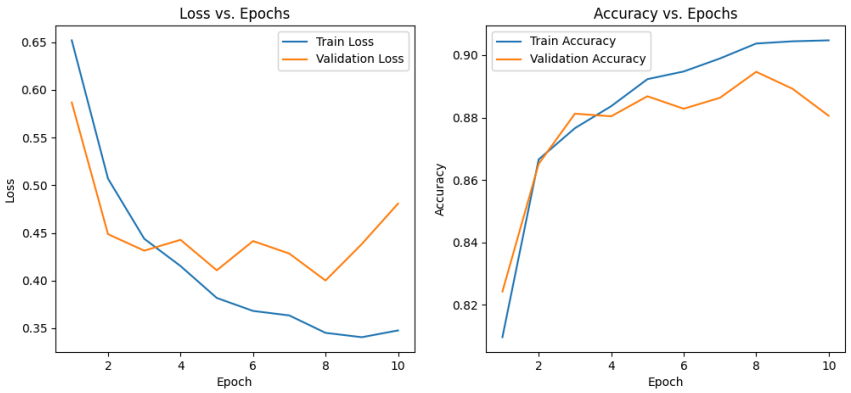


Obtained results are better than in the control experiment, however they were better for larger value of momentum. More trial and error would be required to find the best value of momentum to achieve the best results.

```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.9967, Train Acc: 0.7131
Val Loss: 0.5935, Val Acc: 0.8147
Epoch 2/10, Train Loss: 0.5048, Train Acc: 0.8455
Val Loss: 0.4391, Val Acc: 0.8642
Epoch 3/10, Train Loss: 0.3897, Train Acc: 0.8812
Val Loss: 0.3561, Val Acc: 0.8923
Epoch 4/10, Train Loss: 0.3184, Train Acc: 0.9045
Val Loss: 0.3123, Val Acc: 0.9036
Epoch 5/10, Train Loss: 0.2677, Train Acc: 0.9188
Val Loss: 0.2673, Val Acc: 0.9177
Epoch 6/10, Train Loss: 0.2276, Train Acc: 0.9325
Val Loss: 0.2442, Val Acc: 0.9265
Epoch 7/10, Train Loss: 0.1962, Train Acc: 0.9430
Val Loss: 0.2238, Val Acc: 0.9324
Epoch 8/10, Train Loss: 0.1709, Train Acc: 0.9509
Val Loss: 0.2071, Val Acc: 0.9367
Epoch 9/10, Train Loss: 0.1493, Train Acc: 0.9585
Val Loss: 0.1962, Val Acc: 0.9402
Epoch 10/10, Train Loss: 0.1308, Train Acc: 0.9637
Val Loss: 0.1868, Val Acc: 0.9441
```

Adam (Adaptive Moment Estimation)

In this experiment, we change the optimizer to Adam. These are the results that we obtained:



The obtained results are worse than stochastic gradient descent. In our case Adam optimizer proved to be less effective, but there might cases where it might produce good results.

```
(venv) jan@DESKTOP-SVCO8PE:~/code/earin/lab5$ python3 main.py
Epoch 1/10, Train Loss: 0.6518, Train Acc: 0.8096
Val Loss: 0.5867, Val Acc: 0.8243
Epoch 2/10, Train Loss: 0.5068, Train Acc: 0.8666
Val Loss: 0.4487, Val Acc: 0.8652
Epoch 3/10, Train Loss: 0.4438, Train Acc: 0.8766
Val Loss: 0.4314, Val Acc: 0.8812
Epoch 4/10, Train Loss: 0.4153, Train Acc: 0.8836
Val Loss: 0.4427, Val Acc: 0.8804
Epoch 5/10, Train Loss: 0.3818, Train Acc: 0.8923
Val Loss: 0.4108, Val Acc: 0.8868
Epoch 6/10, Train Loss: 0.3682, Train Acc: 0.8948
Val Loss: 0.4414, Val Acc: 0.8828
Epoch 7/10, Train Loss: 0.3635, Train Acc: 0.8989
Val Loss: 0.4283, Val Acc: 0.8863
Epoch 8/10, Train Loss: 0.3452, Train Acc: 0.9037
Val Loss: 0.4001, Val Acc: 0.8947
Epoch 9/10, Train Loss: 0.3406, Train Acc: 0.9044
Val Loss: 0.4383, Val Acc: 0.8892
Epoch 10/10, Train Loss: 0.3477, Train Acc: 0.9048
Val Loss: 0.4807, Val Acc: 0.8806
```

Conclusions for experimentation with the optimizer:
Changing the optimizer has a huge effect on the final results. It can greatly improve the trained model while keeping all of its parameters the same.

4. Conclusions

The primary objective was to assess how different components and hyperparameters affect the performance of a neural network. The MLP was implemented using PyTorch. The dataset was preprocessed into tensors, normalized, and divided into training and validation sets. The network architecture comprised fully connected layers with ReLU activation functions. The training employed the mini-batch gradient descent method, adjusting hyperparameters like learning rate, batch size, and optimizer parameters. Various configurations were tested to optimize performance metrics such as loss and accuracy.

- Control Experiment: A baseline configuration was established (learning rate = 0.01, mini-batch size = 64, one hidden layer of 512 neurons) to compare subsequent experiments.
- Learning Rate: Tested different rates (0.1, 0.5, 0.005). Found that too high a rate could lead to suboptimal results, while too low a rate slowed down training unnecessarily.
- Batch Size: Varied sizes (1, 16, 256) showed that smaller batches lead to longer training times but possibly finer model tuning, whereas larger batches sped up training but potentially missed finer details.
- Hidden Layers: Adjusted the number of layers (0, 1, 4). Found that adding more layers beyond a certain point did not improve performance but increased computational demands.
- Width of Hidden Layers: Tested various widths (64, 256, 1024). Concluded that there is an optimal width that balances performance and computational efficiency.
- Optimizer Type: Explored different optimizers (SGD with momentum, Adam). Demonstrated that optimizer choice significantly impacts performance, with momentum helping to escape local minima more effectively.

The experiments provided valuable insights into the effects of different hyperparameters and network configurations on the MLP's performance. Adjusting these parameters can significantly affect training efficiency and model accuracy. The optimal configuration depends on balancing computational resources with the desired accuracy and efficiency of the model. Visualizations of training and validation metrics were used throughout to monitor the effects of different configurations, helping to identify the most effective strategies for model training and performance improvement.