CS6140 Project 4

Yihan Xu
NUID: 001566238
Jiahui Zou
NUID: 001535167

Overview:

In this project, we have went through the basic process of running **CNN** and other **ANN**s in python with **pytorch** by following a tutorial for the MNIST digit recognition task; and undertook some experimentations with **6 different dimensions** on the MNIST fashion dataset, evaluated each dimension, and developed the best model; we also performed **transfer learning** on Greek letter using the model from task 1, and achieved 97% accuracy with certain number of epochs; In addition, we tried **4 different ANN** architectures on heart disease prediction. Finally, we followed the **image transferring tutorial on ants and bees** and achieved best accuracy 0.97.

Extensions covered:

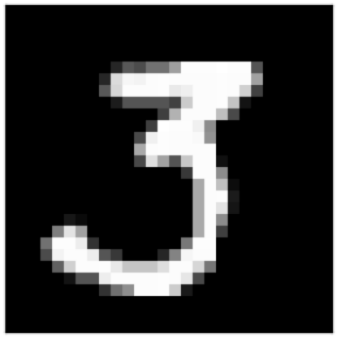
- Experimented 6 dimensions instead of 3 dimensions in task 2.
- Created some additional test data for the greek letter task.
- Explored 4 different architectures for task 4.
- Followed the ImageNet transfer learning tutorial and achieved accuracy 0.97.
- Explored creating networks for an alphabet dataset.

Task 1: MNIST Tutorial

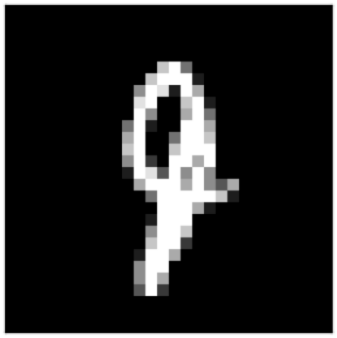
In this task, we went through a tutorial on building a convolutional neural network for the MNIST digit recognition task following this blog.

Dataset Examples:

Ground Truth: 3



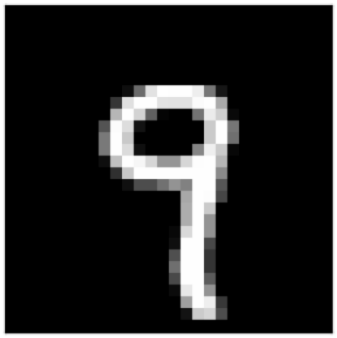
Ground Truth: 9



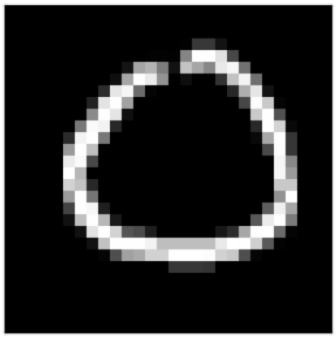
Ground Truth: 4

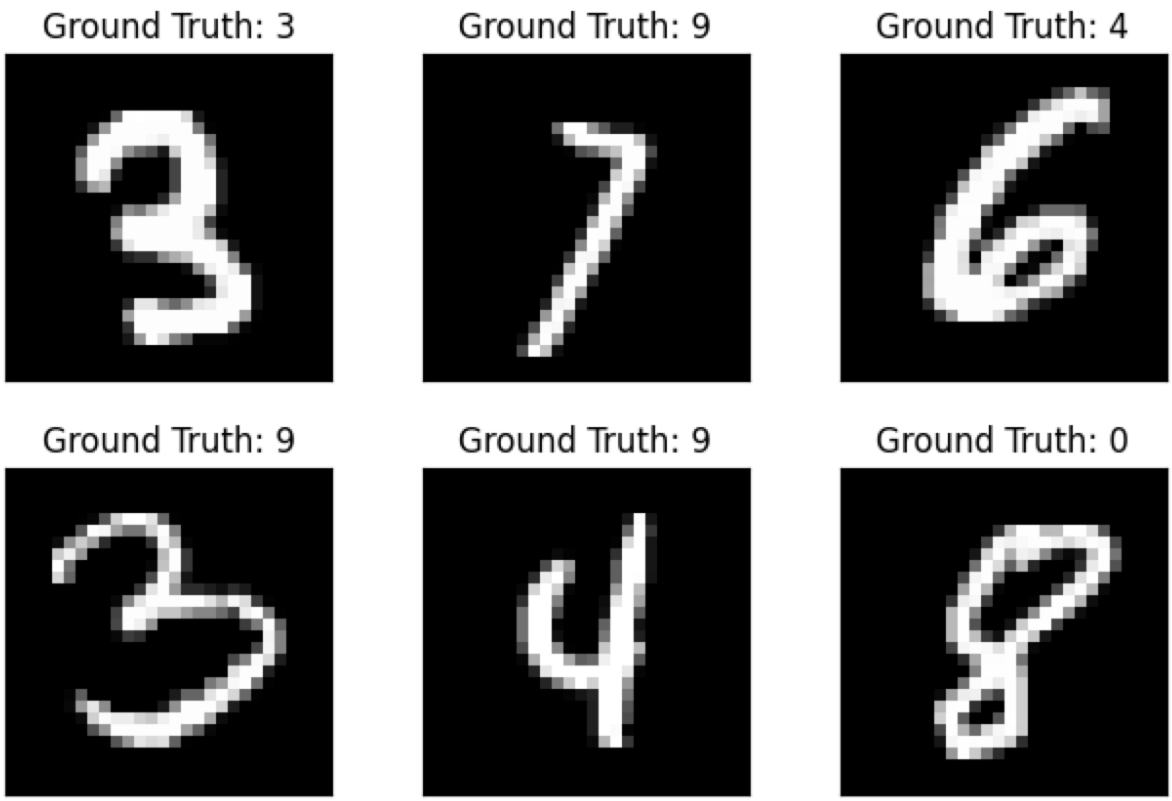
Ground Truth: 9

Ground Truth: 9



Ground Truth: 0

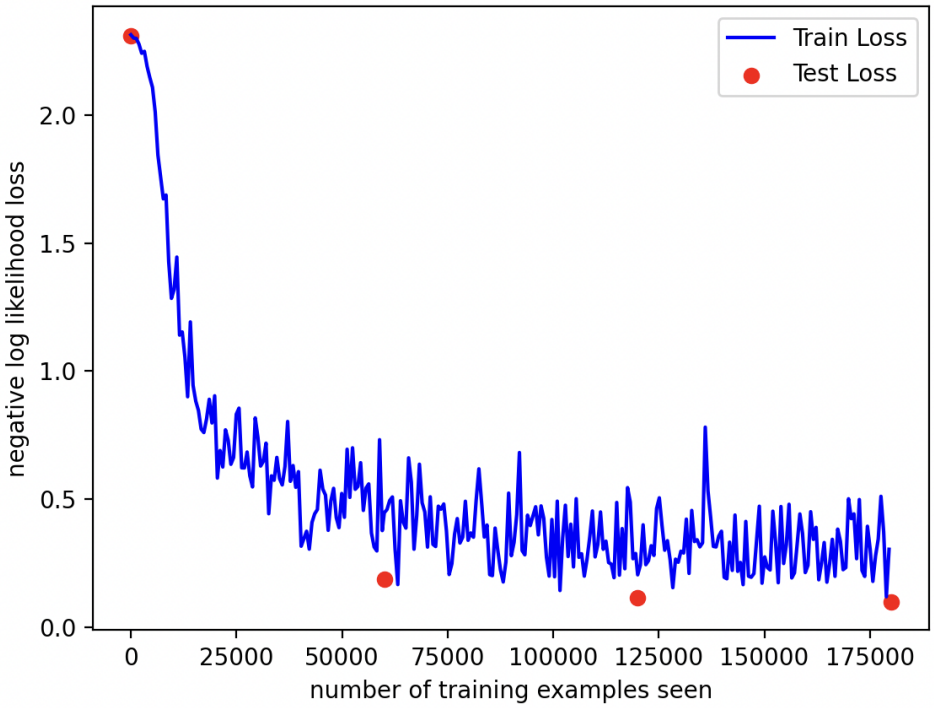




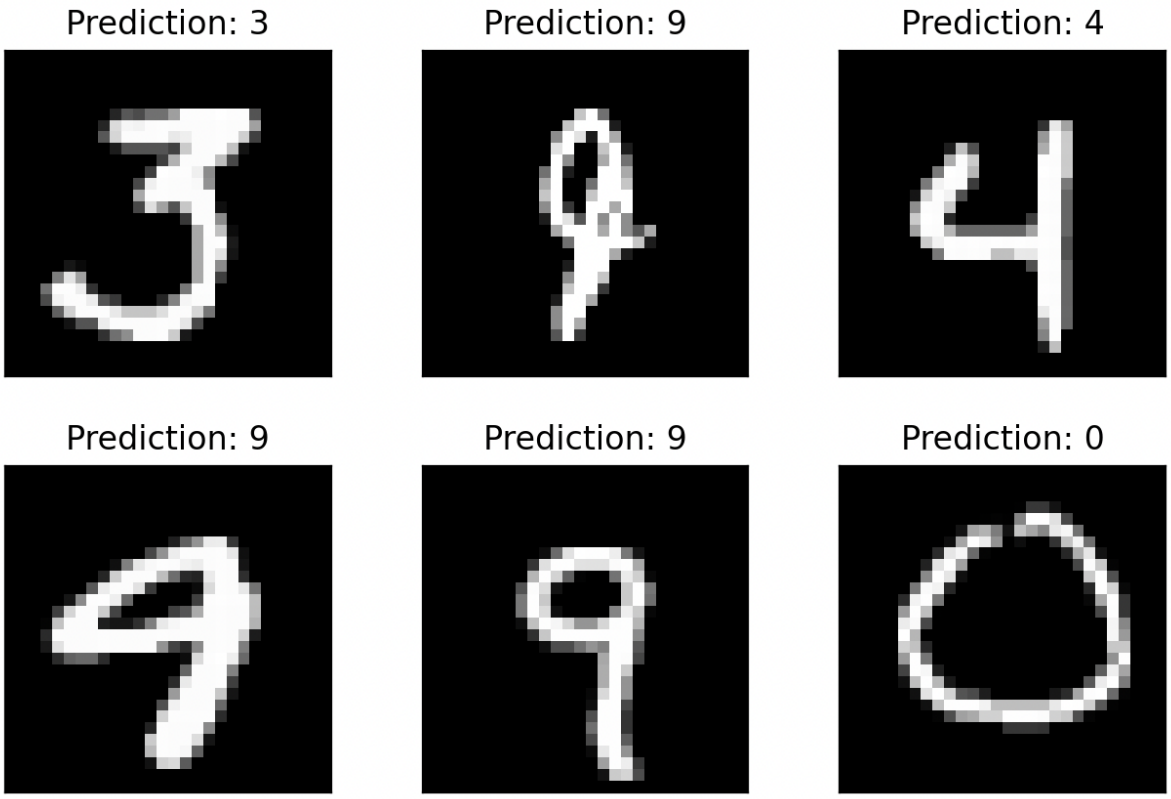
Printout of the network:

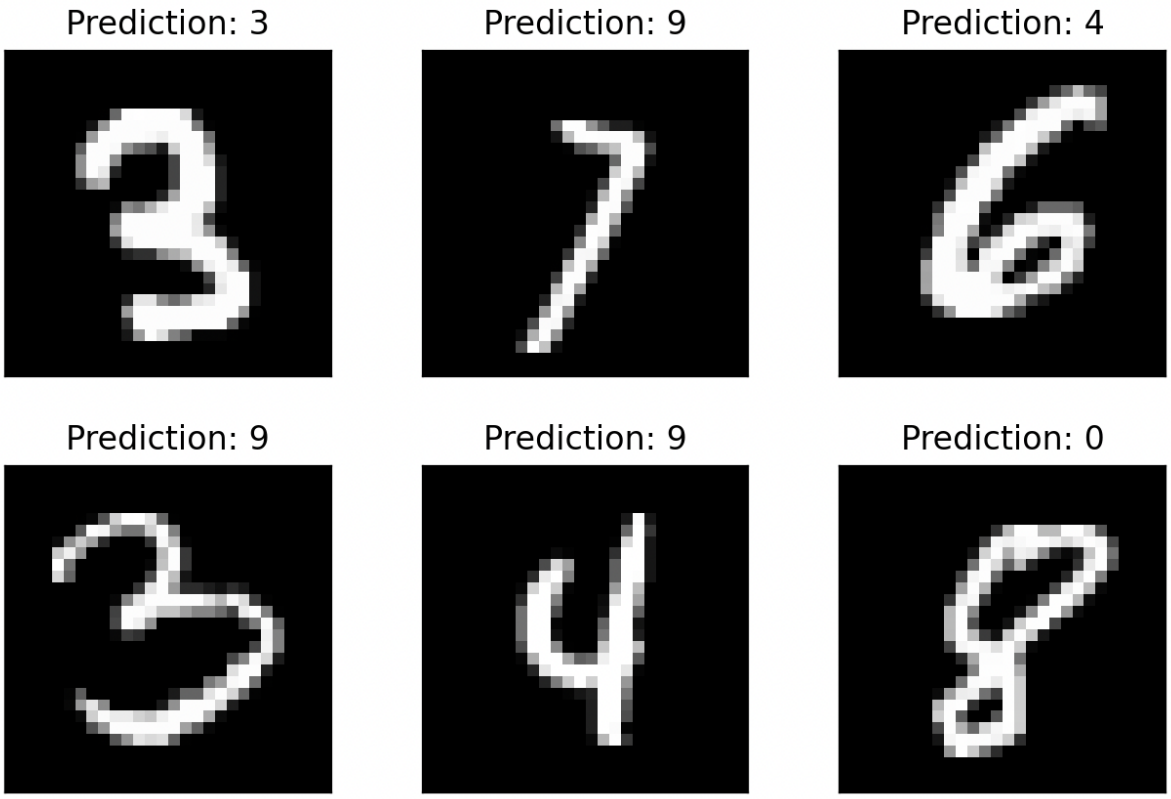
```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Training and Testing Loss:



Example outputs:





Task 2: Experiment with network variations

In this task, we are experimenting with different dimensions on the **MNIST Fashion dataset.** And after careful considerations, we decided to test 6 different dimensions.

Dimensions to adjust:

- **The number of epochs of training**
- **dropout rate**
- **Learning rate**
- **The number of convolution filters in a layer**
- **The batch size while training**
- **The number of convolution layers**

Develop a plan:

To automate the process, we decide to pass all dimensions as parameters into the function.

To test each dimension individually, we used a linear search strategy to hold all other dimensions constant and test one dimension at a time, and memorize the optimal dimension in each test.
And after this round, we then test some dimension with the other optimal dimensions, and gradually find all optimal dimensions.

Default values:

DEFAULT_N_EPOCH = 3
DEFAULT_BATCH_SIZE = 64
DEFAULT_DROPOUT_RATE = 0.5
DEFAULT_LEARNING_RATE = 0.01
DEFAULT_NUM_FILTERS_1 = 10
DEFAULT_NUM_FILTERS_2 = 20
DEFAULT_NUM_LAYERS = 2

Testing range:

Number of epochs: set to 20 and see when the accuracy stops increasing or even starts to drop.
Batch size: starting from 64 and double the size each time.
Dropout rate: [0.01, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5]
Learning rate: 0.01 ~ 0.15, increase 0.01 at a time.
Number of filters: 10, and double the number each time.
Number of convolutional layers: [2, 3]

predictions on the results for each dimension:

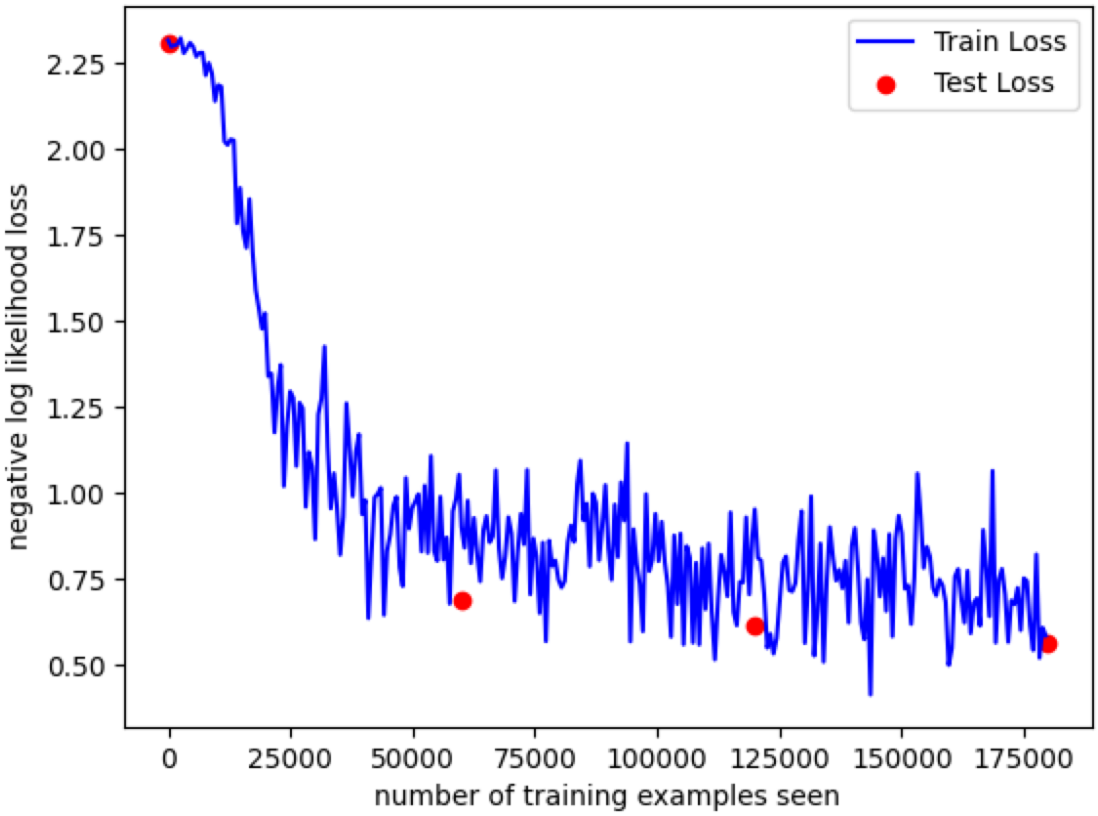
- **The number of epochs of training**: by increasing the number of epochs, the training time might increase, but the accuracy and model complexity will also increase. **We decide to set the number of epochs to a fixed value** (e.g., 10, 50, or 100) and then monitor the performance of the network on a validation set. If the validation performance plateaus or starts to decrease, then it may be necessary to reduce the number of epochs to prevent overfitting.
- **dropout rate**: Dropout is a regularization technique to prevent overfitting. Dropout randomly drops out a proportion of the neurons in a layer during training. Increasing the dropout rate may prevent overfitting, but it may also reduce the accuracy with too much dropout. We need to find the sweet spot.
- **Learning rate**: A high learning rate can result in the model overshooting the minimum, while a low learning rate can make the training process slow, we need to find a sweet spot for both accuracy and time spent.

- **The number of convolution filters in a layer**: The number of filters in each convolutional layer determines the number of features that the network can learn. A higher number of filters can capture more complex patterns, but it can also require more computational resources, and might cause overfitting. A good starting point could be 32 or 64.
- **The batch size while training:** A larger batch size can lead to faster training times because the model can process more data in a single pass, but it also requires more memory and can make the optimization process more unstable.
- **The number of convolution layers**: there is no good start point. It can increase the model complexity and accuracy but might also increase the risk of overfitting and make the network slower to train.

Result:

Result with all DEFAULT dimensions:

Training and testing loss:



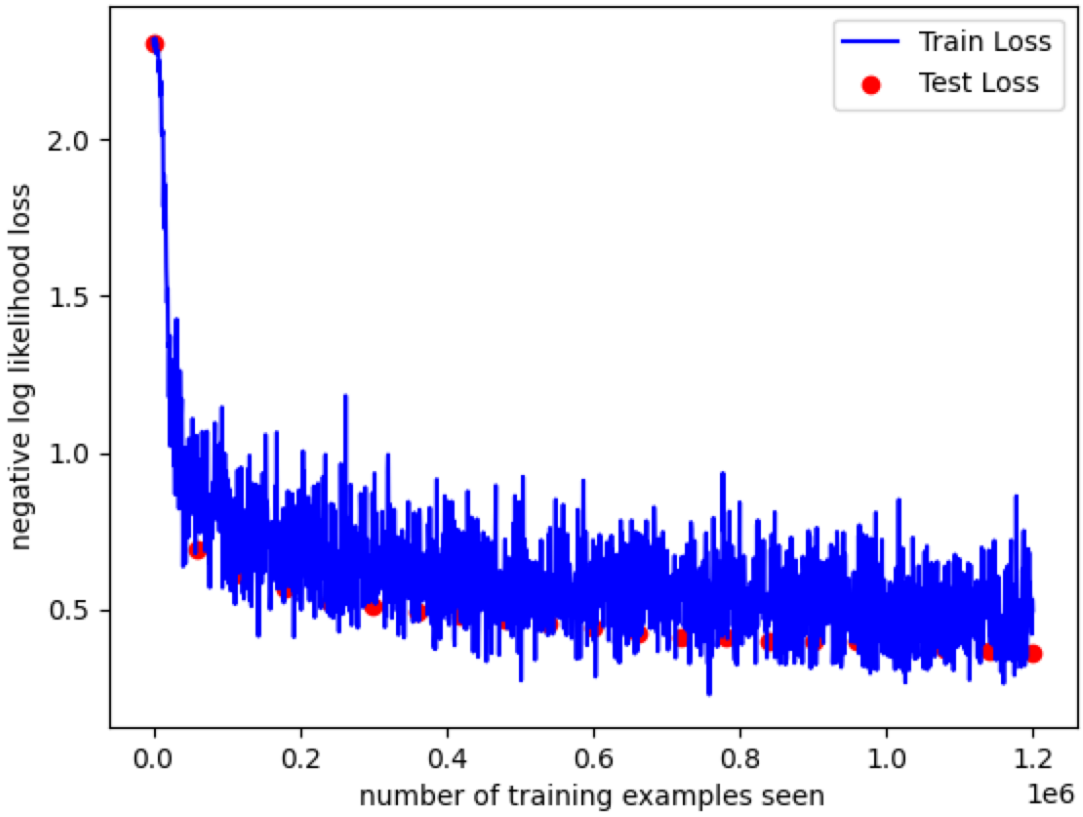

```
The result for using all default values, accuracy is: 78.0%, time spent is: 67
```

As we can see from the above results, the accuracy is only 78%, which is not a very satisfactory result, and time spent is 67s, which is acceptable.

Result for epoch testing with all other dimensions DEFAULT:

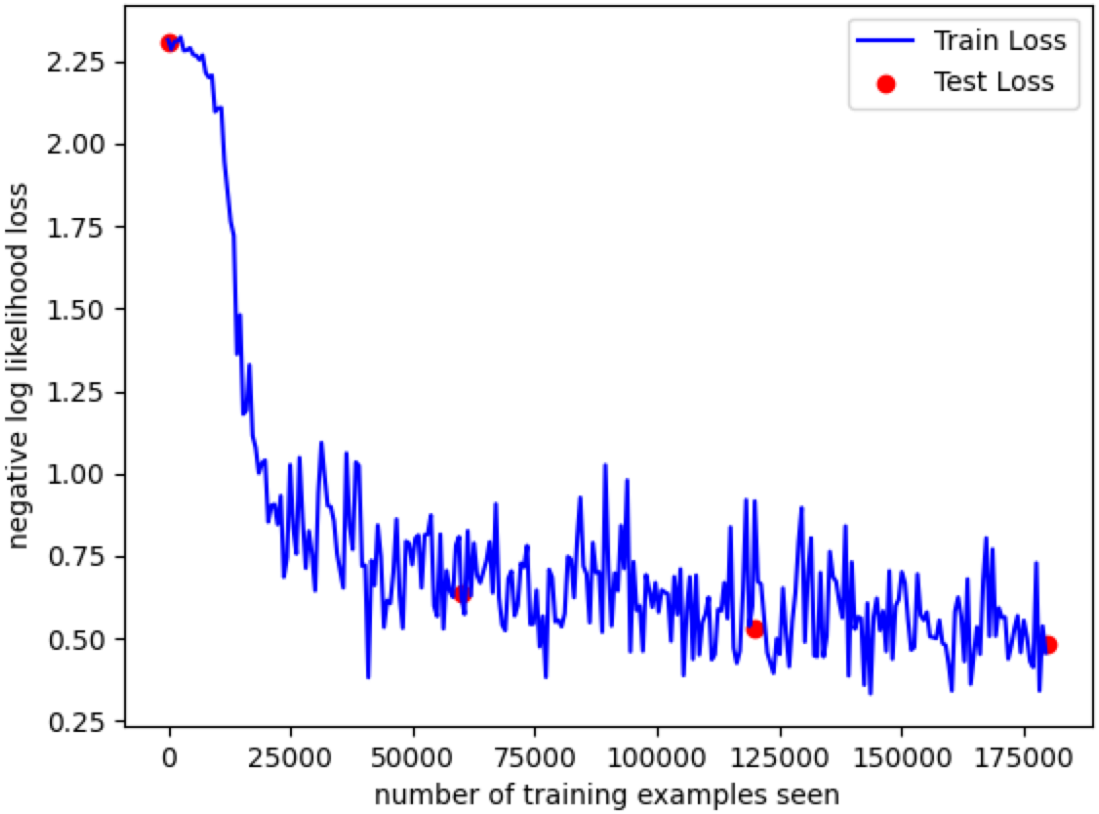
```
Train Epoch: 1
Test set: Avg. loss: 0.6868, Accuracy: 7402/10000 (74%)

Train Epoch: 2
Test set: Avg. loss: 0.6116, Accuracy: 7619/10000 (76%)

Train Epoch: 3
Test set: Avg. loss: 0.5583, Accuracy: 7773/10000 (78%)

Train Epoch: 4
Test set: Avg. loss: 0.5325, Accuracy: 7898/10000 (79%)

Train Epoch: 5
Test set: Avg. loss: 0.5093, Accuracy: 7939/10000 (79%)

Train Epoch: 6
Test set: Avg. loss: 0.4910, Accuracy: 7996/10000 (80%)

Train Epoch: 7
Test set: Avg. loss: 0.4806, Accuracy: 8146/10000 (81%)

Train Epoch: 8
Test set: Avg. loss: 0.4670, Accuracy: 8212/10000 (82%)

Train Epoch: 9
Test set: Avg. loss: 0.4497, Accuracy: 8288/10000 (83%)

Train Epoch: 10
Test set: Avg. loss: 0.4401, Accuracy: 8367/10000 (84%)

Train Epoch: 11
Test set: Avg. loss: 0.4296, Accuracy: 8407/10000 (84%)

Train Epoch: 12
Test set: Avg. loss: 0.4141, Accuracy: 8514/10000 (85%)

Train Epoch: 13
Test set: Avg. loss: 0.4141, Accuracy: 8496/10000 (85%)

Train Epoch: 14
Test set: Avg. loss: 0.3982, Accuracy: 8544/10000 (85%)
```

Training and testing loss:



In this test, we gave the number of epochs a fixed number: 20.
As we can see from the above result, the accuracy improves pretty slowly after the 15th epoch. So the best number of epochs is 15.

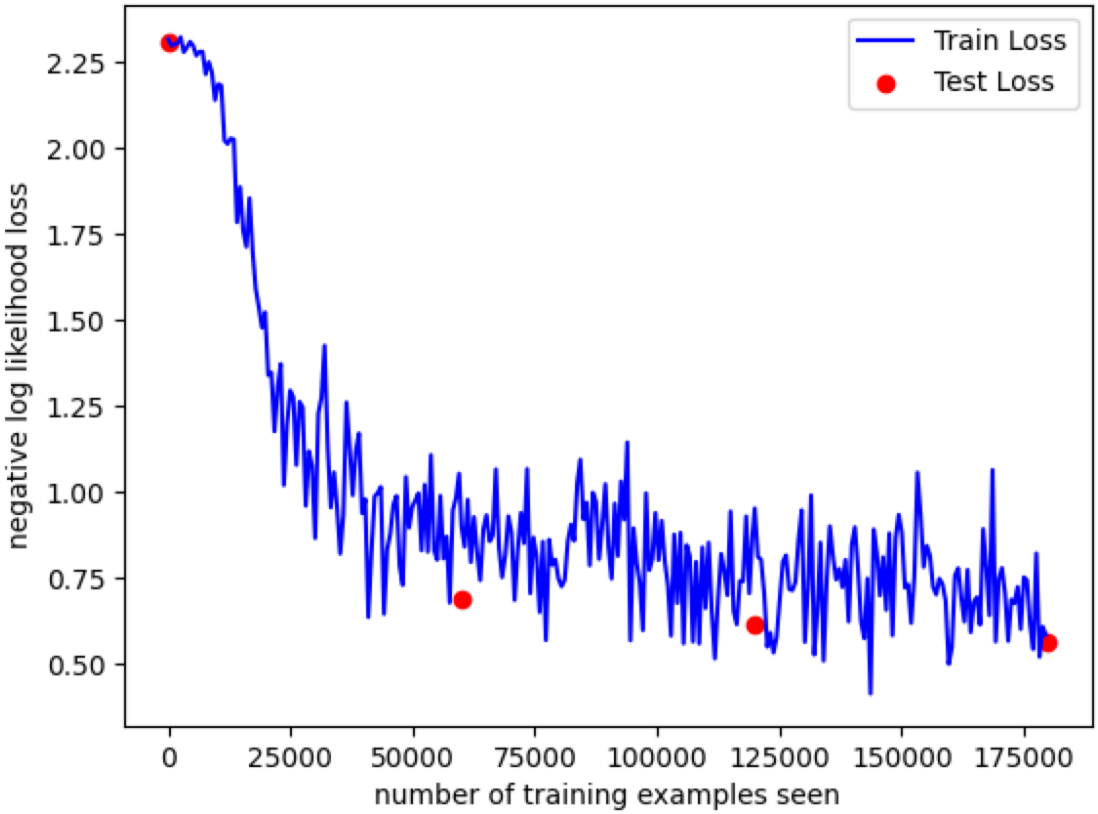
Result for dropout rate testing with all other dimensions DEFAULT:
Training and testing loss:




```
the best dropout rate is: 0.01, and the accuracy is: 82.0%
```

In this test, we gave the dropout rate a range of [0.01, 0.1, 0.2, 0.25, 0.3, 0.4, 0.5] and kept all other dimensions as DEFAULT.
As we can see in the result above, the best dropout rate is 0.01, with the accuracy of 82%, which has improved a lot compared to the result of all default dimensions.

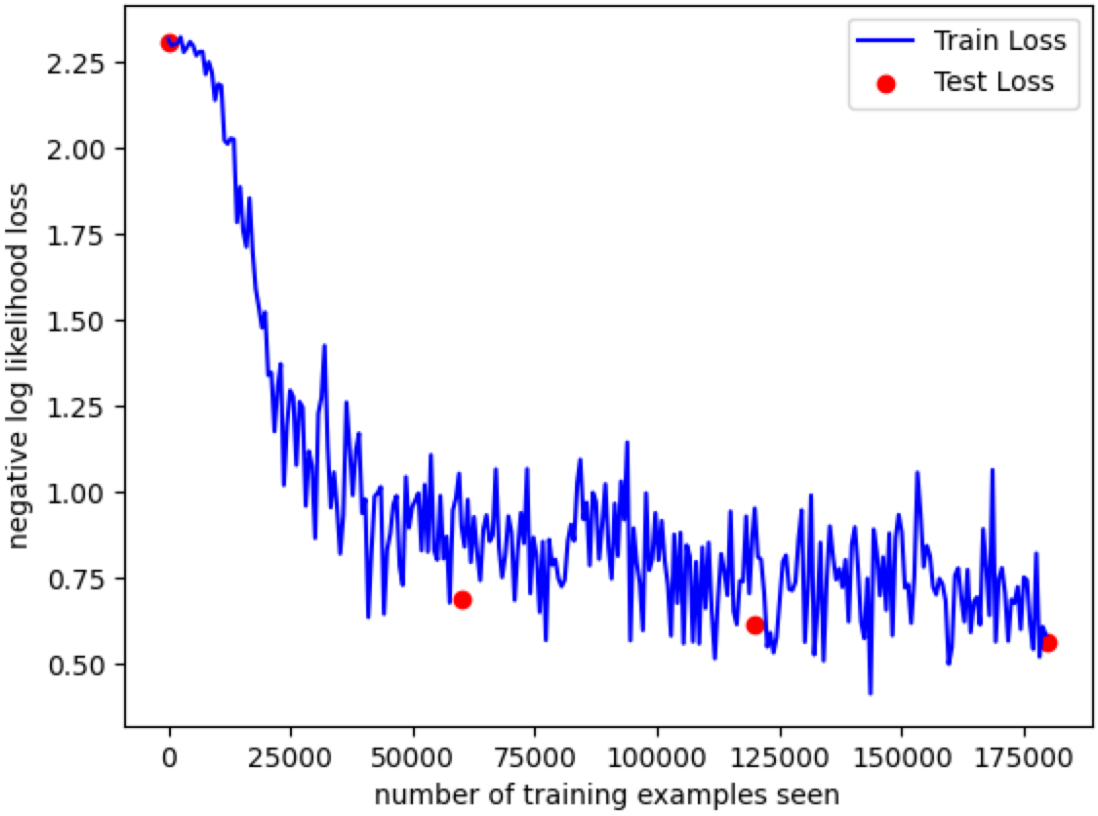
Result for learning rate testing with all other dimensions DEFAULT:
Training and testing loss:




```
the best learning rate is: 0.07, and the accuracy is: 84.0%
```

In this test, we gave the learning rate in a range of 0.01 to 0.15, with 0.01 improvement at a time, and all other dimensions DEFAULT.
As we can see from the result above, the best learning rate is 0.07, with the accuracy of 84%, which has improved a lot compared to the result of all default dimensions.

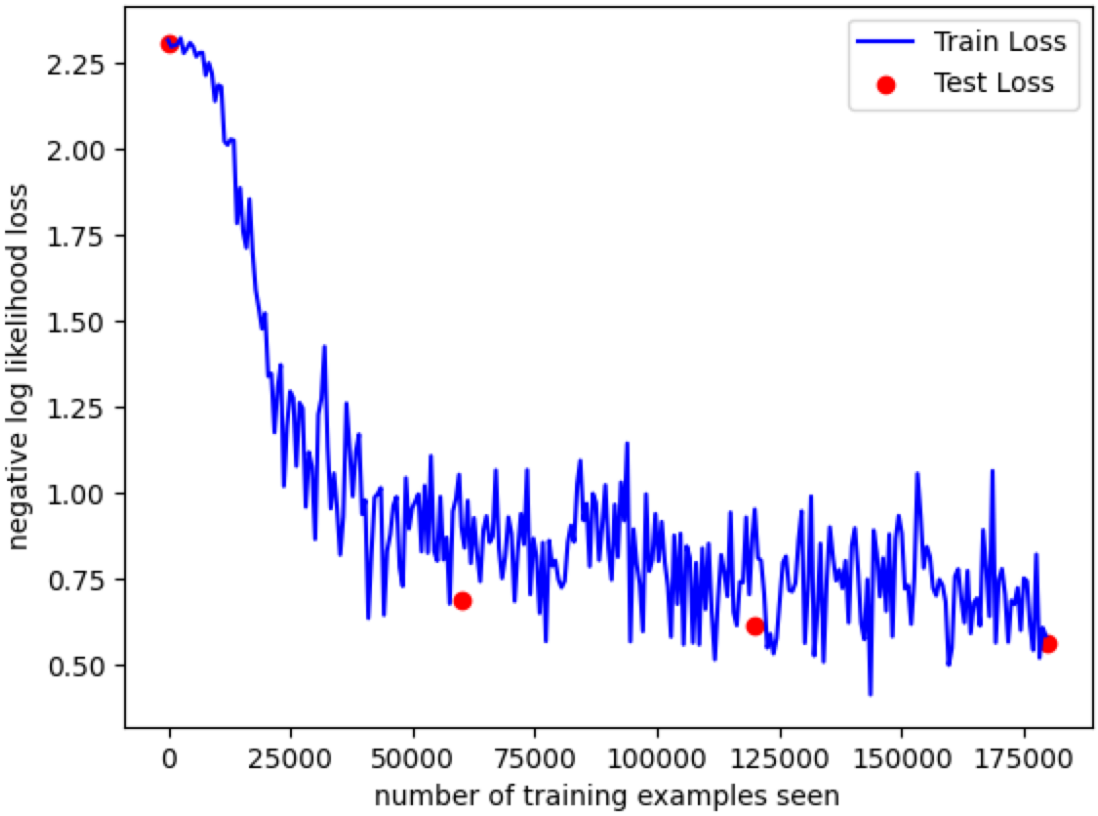
Result for learning rate testing with all other dimensions DEFAULT:
Training and testing loss:




```
the best batch size is: 64
```

In this test, we started the batch size with 64, and increased the batch size by 2 each time. As we can see, when the batch size improves, the time spent drops, and the accuracy stays the same.
However, when the time improvement is not significant, or the accuracy starts to drop, or the program crashes, we will stop increasing the batch size.
The best batch size, in this case, is 64.

Result for number of filters testing with all other dimensions DEFAULT:
Training and testing loss:




```
the best number of filter is: 40 and 80, and the best accuracy is 81.0%
```

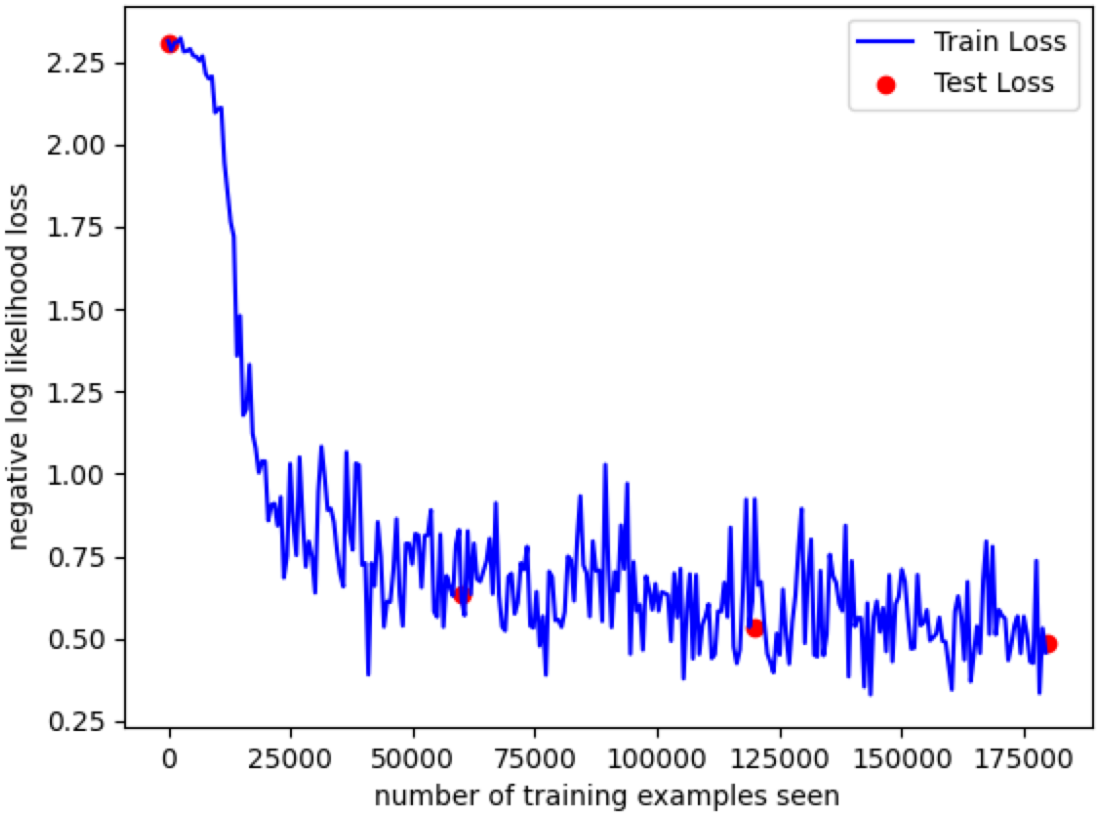
In this test, we started with 10 and 20 (for first and second convolutional layers), and doubled the number each time.

When the time spent is more than 300, or the accuracy stops increasing, we stop increasing the number of filters.

In this case, the best number of filters are 40 and 80 (for first and second convolutional layers)

Result for learning rate testing with optimal dropout rate and all other dimensions DEFAULT:

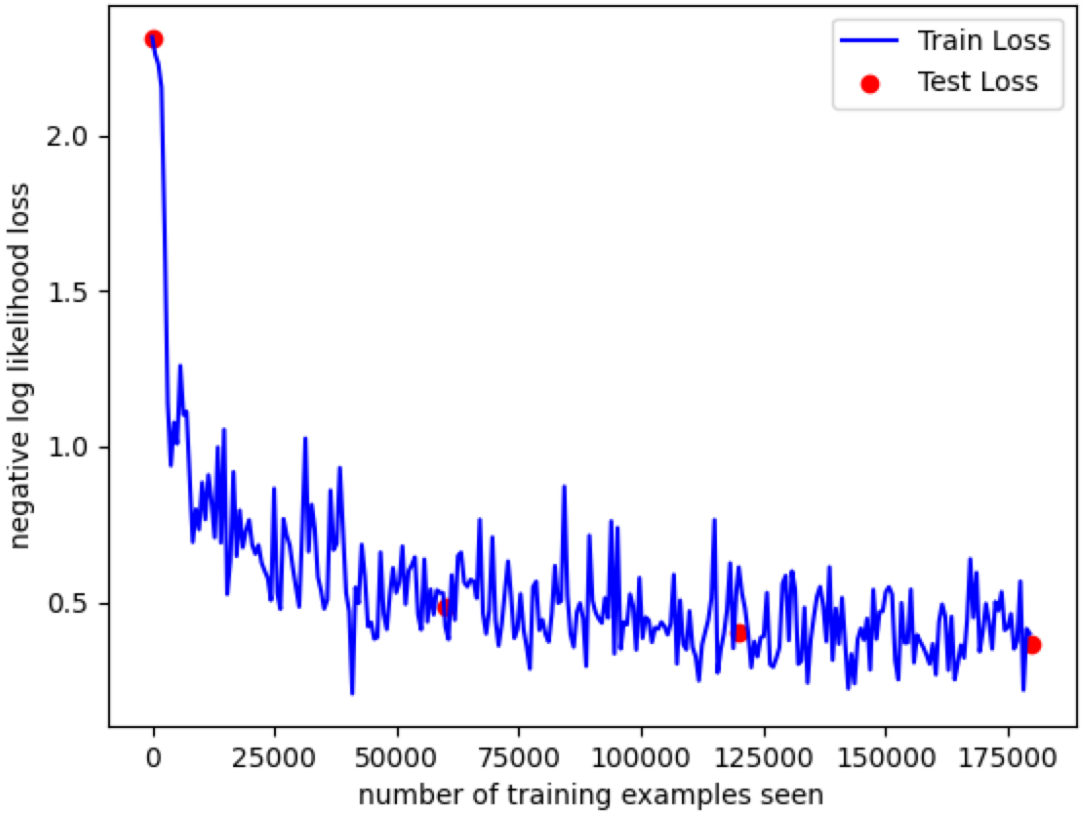
Training and testing loss:




```
the best learning rate with optimal dropout rate is: 0.06, and the accuracy is: 87.0%
```

In this test, we are testing the optimal learning rate ranging from 0.01 to 0.15, with the optimal dropout rate and all other dimensions as DEFAULT.
We are keeping track of the best accuracy, and as we can see from the result above, the best learning rate with optimal dropout rate is 0.06, and the accuracy is 87%, which is a relatively satisfactory result.

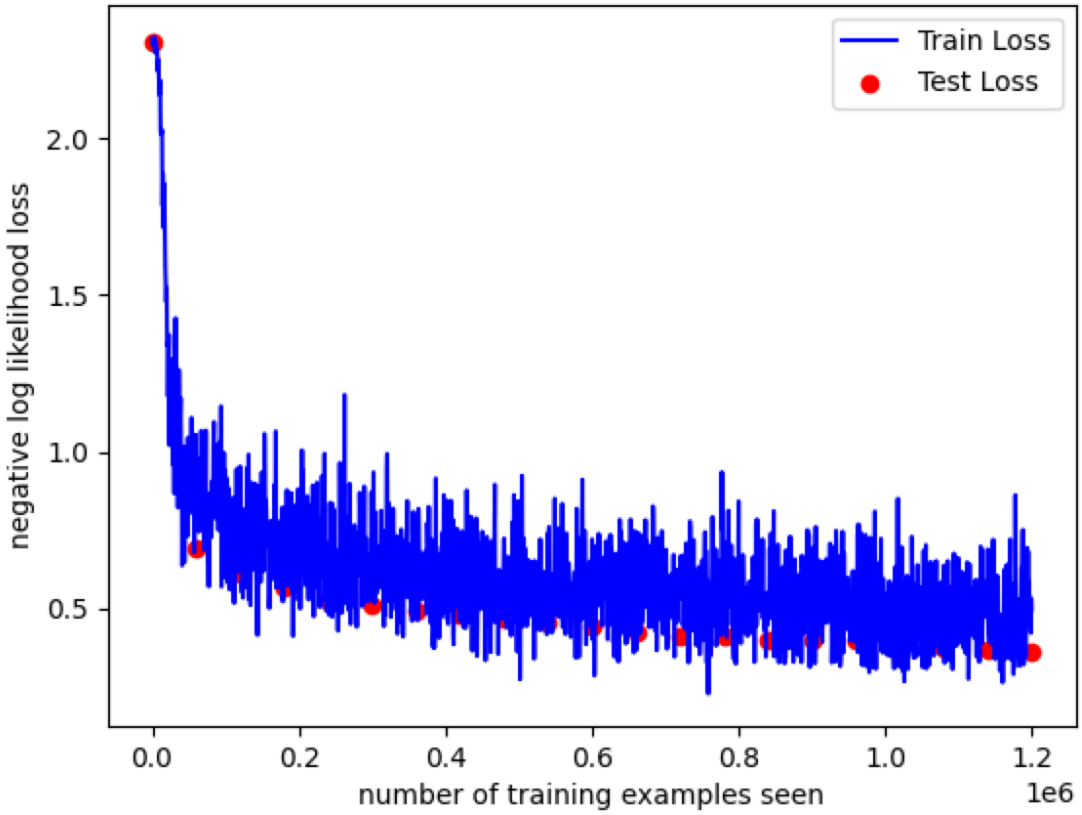
Result for number of filters testing with optimal dropout rate and learning rate:
Training and testing loss:




```
the best number of filter with optimal dropout rate and learning rate is: 40 and 80, and the best accuracy is 89.0%
```

In this test, we are testing the optimal number of filters starting from 10 and 20 (first and second convolutional layer), with the optimal dropout rate, optimal learning rate and all other dimensions as DEFAULT.
We are keeping track of the best accuracy, and as we can see from the result above, the best number of filters with optimal dropout rate and learning rate is 40 and 80, and the accuracy is 89%, which is relatively satisfactory.

Result for all best dimensions with 2 convolutional layers:
Training and testing loss:

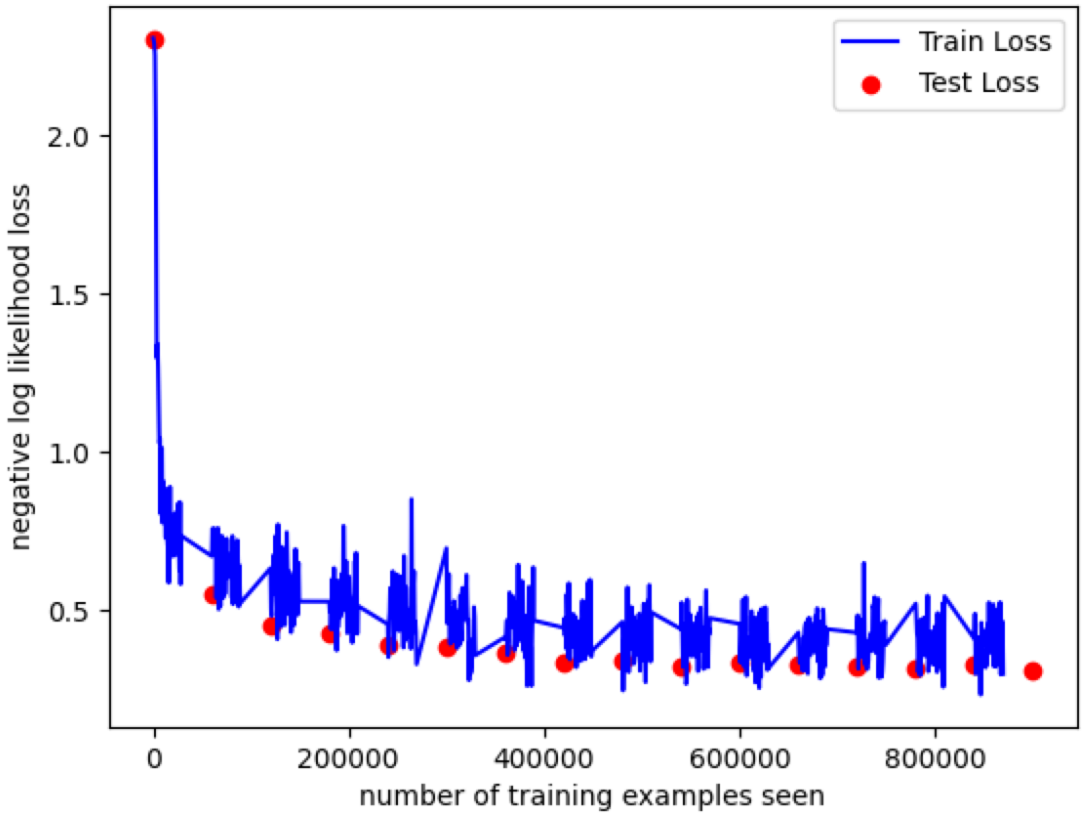



```
the best combined result: accuracy is 91.0, and time spent is 1248s
```

In this test, we are using all best dimensions with 2 convolutional layers:
Number of epochs: 15
Batch size: 64
Dropout rate: 0.01
Learning rate: 0.06
Number of filters: 40 and 80

As we can see from the result above, the accuracy is 91%, which is the highest accuracy so far, however, the time spent is 1248s, which is relatively long. We need to balance time and accuracy.

Result for all best dimensions with 3 convolutional layers:
Training and testing loss:




```
the best combined result with 3 layers: accuracy is 89.0, and time spent is 271s
```

In this test, we are using all best dimensions with 3 convolutional layers:

As we can see from the result above, the accuracy is 89%, which is 2% lower than the result with 2 convolutional layers, but the time spent is much less.

Summary:

In the tests above, the changing curve for each dimension is the same as we predicted. And we found the best dimensions as follows:
Number of epochs: 15
Batch size: 64
Dropout rate: 0.01
Learning rate: 0.06
Number of filters: 40 and 80
And with the best dimensions we found, we can achieve **91%** accuracy with 2 convolutional layers, but time spent is 1248s. With 3 convolutional layers, we can achieve **89%** accuracy, time spent is only 271s.
To determine which set of dimensions we are going to use, it depends on our needs, whether we need higher accuracy or faster results.

Task 3: Transfer Learning on Greek Letters

In this task, We applied transfer learning on 3 Greek Letters using the model weights from a pre-trained model in task 1.

Original network printout:

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

Modified network printout:

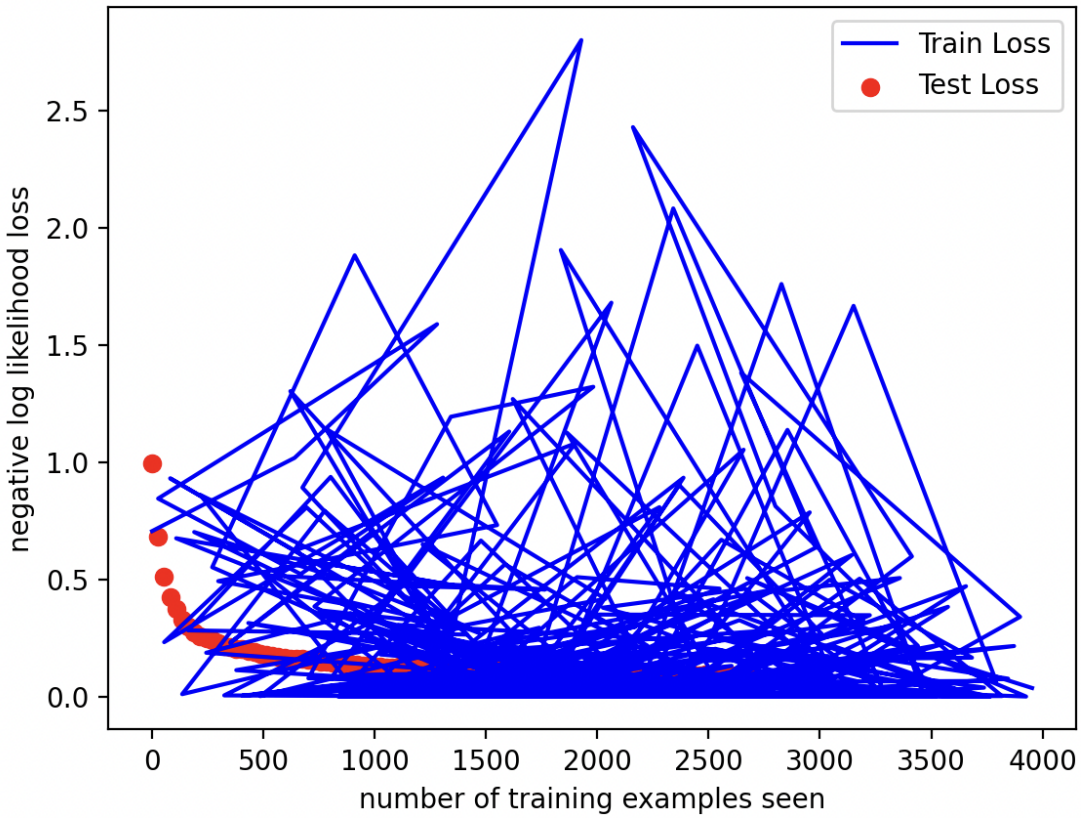
```
torch.Size([9, 1, 28, 28])
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(5, 5), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=320, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=3, bias=True)
)
```

Results:

Epochs with learning rate = 0.01

```
Test set: Avg. loss: 0.9932, Accuracy: 17/27 (63%)

Train Epoch: 1
Test set: Avg. loss: 0.6823, Accuracy: 24/27 (89%)

Train Epoch: 2
Test set: Avg. loss: 0.5120, Accuracy: 25/27 (93%)

Train Epoch: 3
Test set: Avg. loss: 0.4244, Accuracy: 25/27 (93%)

Train Epoch: 4
Test set: Avg. loss: 0.3739, Accuracy: 25/27 (93%)

Train Epoch: 5
Test set: Avg. loss: 0.3289, Accuracy: 26/27 (96%)

Train Epoch: 6
Test set: Avg. loss: 0.2990, Accuracy: 26/27 (96%)

Train Epoch: 7
Test set: Avg. loss: 0.2760, Accuracy: 26/27 (96%)

Train Epoch: 8
Test set: Avg. loss: 0.2557, Accuracy: 26/27 (96%)

Train Epoch: 9
Test set: Avg. loss: 0.2498, Accuracy: 26/27 (96%)

Train Epoch: 10
Test set: Avg. loss: 0.2409, Accuracy: 26/27 (96%)

Train Epoch: 11
Test set: Avg. loss: 0.2267, Accuracy: 26/27 (96%)

Train Epoch: 12
Test set: Avg. loss: 0.2163, Accuracy: 26/27 (96%)
```

Training and testing loss for epoch = 100 with learning rate = 0.01



According to the result above, in the 5th epoch, the accuracy reaches the highest 96%, and then stays the same. So it takes us 5 epochs to perfectly identify the letters.

Modifications on the dimensions:

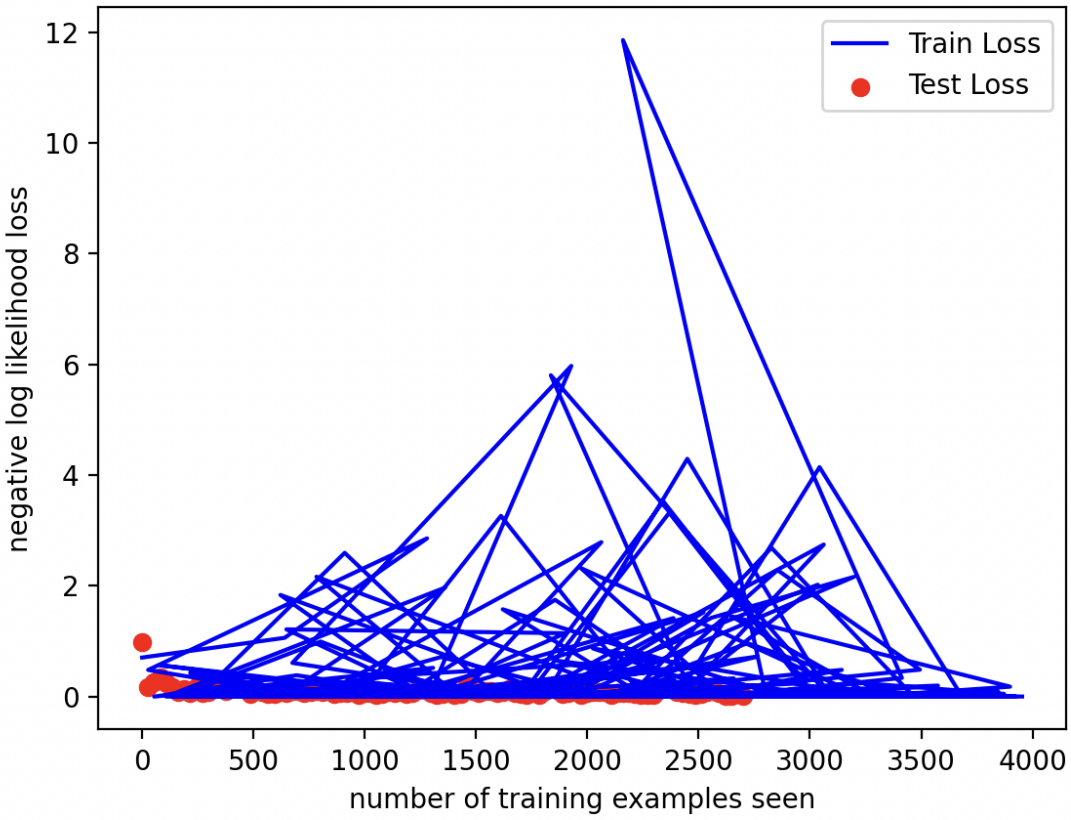
Epochs with learning rate = 0.1

```
Test set: Avg. loss: 0.9932, Accuracy: 17/27 (63%)

Train Epoch: 1
Test set: Avg. loss: 0.1781, Accuracy: 26/27 (96%)

Train Epoch: 2
Test set: Avg. loss: 0.2529, Accuracy: 24/27 (89%)

Train Epoch: 3
Test set: Avg. loss: 0.3418, Accuracy: 23/27 (85%)

Train Epoch: 4
Test set: Avg. loss: 0.2397, Accuracy: 25/27 (93%)

Train Epoch: 5
Test set: Avg. loss: 0.1679, Accuracy: 25/27 (93%)

Train Epoch: 6
Test set: Avg. loss: 0.0863, Accuracy: 26/27 (96%)

Train Epoch: 7
Test set: Avg. loss: 0.1293, Accuracy: 26/27 (96%)

Train Epoch: 8
Test set: Avg. loss: 0.0737, Accuracy: 27/27 (100%)

Train Epoch: 9
Test set: Avg. loss: 0.1757, Accuracy: 25/27 (93%)
```

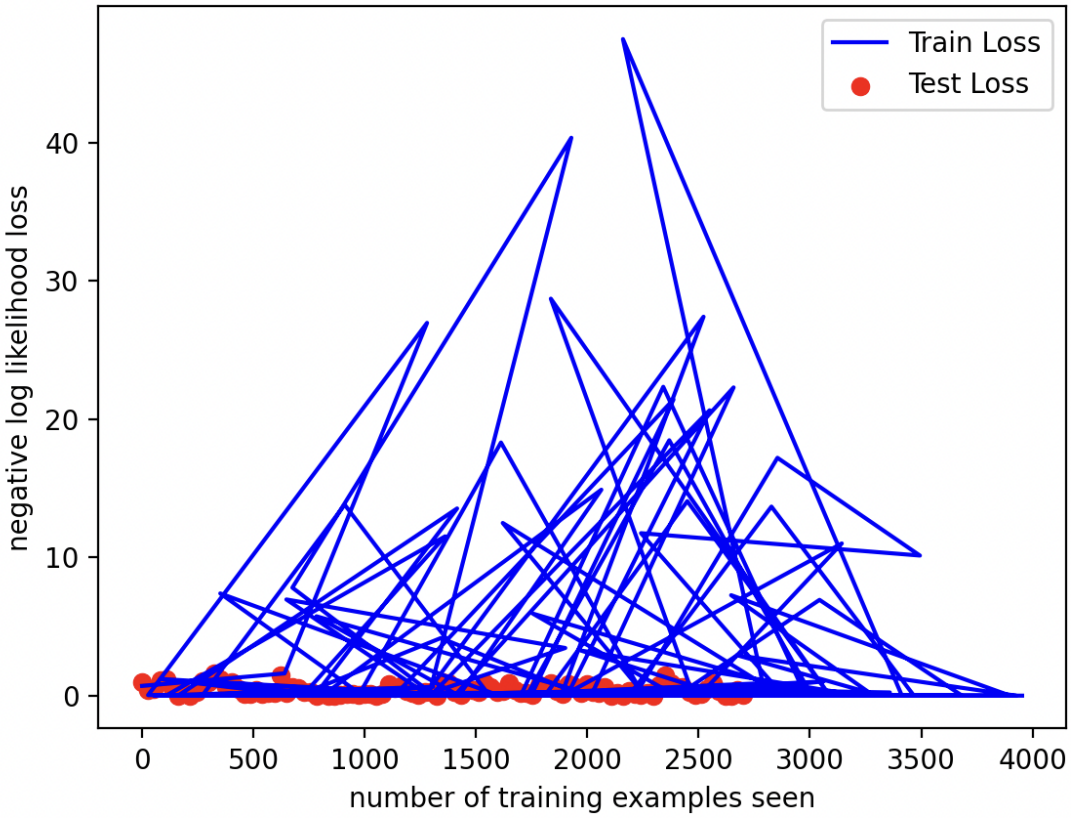
Epochs with learning rate = 0.5

```
Test set: Avg. loss: 0.9932, Accuracy: 17/27 (63%)

Train Epoch: 1
Test set: Avg. loss: 0.3780, Accuracy: 26/27 (96%)

Train Epoch: 2
Test set: Avg. loss: 0.4697, Accuracy: 26/27 (96%)

Train Epoch: 3
Test set: Avg. loss: 1.1208, Accuracy: 23/27 (85%)

Train Epoch: 4
Test set: Avg. loss: 1.1953, Accuracy: 23/27 (85%)

Train Epoch: 5
Test set: Avg. loss: 0.5478, Accuracy: 26/27 (96%)

Train Epoch: 6
Test set: Avg. loss: 0.0178, Accuracy: 27/27 (100%)

Train Epoch: 7
Test set: Avg. loss: 0.1892, Accuracy: 26/27 (96%)

Train Epoch: 8
Test set: Avg. loss: 0.0176, Accuracy: 27/27 (100%)

Train Epoch: 9
Test set: Avg. loss: 0.2937, Accuracy: 26/27 (96%)
```

Training and testing loss for epoch = 100 with learning rate = 0.1



Training and testing loss for epoch = 100 with learning rate = 0.5



According to the results above, the best accuracy we can achieve is 100% when epoch = 8 with learning rate = 0.1, and epoch = 6 and 8 when learning rate = 0.5, but it is not stable, and drops as epoch increases, which might have overfit.

Additional test data with the model:

Result using 15 samples

```
Train Epoch: 73
Test set: Avg. loss: 7.2477, Accuracy: 31/41 (76%)

Train Epoch: 74
Test set: Avg. loss: 7.0500, Accuracy: 31/41 (76%)

Train Epoch: 75
Test set: Avg. loss: 5.1432, Accuracy: 31/41 (76%)

Train Epoch: 76
Test set: Avg. loss: 6.1609, Accuracy: 32/41 (78%)

Train Epoch: 77
Test set: Avg. loss: 6.4282, Accuracy: 32/41 (78%)

Train Epoch: 78
Test set: Avg. loss: 4.9346, Accuracy: 33/41 (80%)

Train Epoch: 79
Test set: Avg. loss: 4.8209, Accuracy: 33/41 (80%)

Train Epoch: 80
Test set: Avg. loss: 5.5498, Accuracy: 33/41 (80%)

Train Epoch: 81
Test set: Avg. loss: 4.5083, Accuracy: 31/41 (76%)

Train Epoch: 82
Test set: Avg. loss: 4.7895, Accuracy: 33/41 (80%)

Train Epoch: 83
Test set: Avg. loss: 5.7183, Accuracy: 33/41 (80%)

Train Epoch: 84
Test set: Avg. loss: 5.2161, Accuracy: 33/41 (80%)

Train Epoch: 85
Test set: Avg. loss: 5.5365, Accuracy: 34/41 (83%)

Train Epoch: 86
Test set: Avg. loss: 5.8077, Accuracy: 29/41 (71%)
```

```
Train Epoch: 87
Test set: Avg. loss: 6.3931, Accuracy: 31/41 (76%)

Train Epoch: 88
Test set: Avg. loss: 6.6462, Accuracy: 30/41 (73%)

Train Epoch: 89
Test set: Avg. loss: 5.6921, Accuracy: 31/41 (76%)

Train Epoch: 90
Test set: Avg. loss: 6.1086, Accuracy: 31/41 (76%)

Train Epoch: 91
Test set: Avg. loss: 6.8042, Accuracy: 31/41 (76%)

Train Epoch: 92
Test set: Avg. loss: 6.7386, Accuracy: 32/41 (78%)

Train Epoch: 93
Test set: Avg. loss: 7.4794, Accuracy: 32/41 (78%)

Train Epoch: 94
Test set: Avg. loss: 6.1820, Accuracy: 32/41 (78%)

Train Epoch: 95
Test set: Avg. loss: 6.2647, Accuracy: 32/41 (78%)

Train Epoch: 96
Test set: Avg. loss: 6.8389, Accuracy: 32/41 (78%)

Train Epoch: 97
Test set: Avg. loss: 5.9892, Accuracy: 33/41 (80%)

Train Epoch: 98
Test set: Avg. loss: 6.0275, Accuracy: 33/41 (80%)

Train Epoch: 99
Test set: Avg. loss: 6.2048, Accuracy: 33/41 (80%)

Train Epoch: 100
Test set: Avg. loss: 6.7189, Accuracy: 33/41 (80%)
```

In this task, we are testing 3 alphabets with 15 samples each, the samples are handwritten by ourselves and saved as png.
According to the result above, it might be due to the poor shape and resolution of our test images, the highest accuracy we can achieve is 80%.

Task 4: Heart disease prediction using ANN

In this task, we are experimenting with 4 different architectures to predict heart disease using the dataset from project 4.
2 of the architectures are **feedforward neural networks**, and another 2 of them are **recurrent neural networks.**

Firstly, we did some researches on the ANNs we can use:

1. **Feedforward neural networks**: These are the most common type of ANNs and consist of a series of layers where each neuron in one layer is connected to every neuron in the next layer. Feedforward networks are commonly used for tasks such as classification, regression, and pattern recognition.
2. **Recurrent neural networks (RNNs)**: These networks have feedback connections that allow information to be passed from one time step to the next. RNNs are often used for tasks involving sequential data, such as language modeling and speech recognition.
3. **Convolutional neural networks (CNNs)**: These networks use convolutional layers to extract features from images and other types of grid-like data. CNNs are commonly used for tasks such as image classification and object detection.
4. **Autoencoder neural networks**: These networks are used for tasks such as data compression and image denoising. They consist of an encoder network that maps the input data to a lower-dimensional representation, and a decoder network that maps the lower-dimensional representation back to the original input data.
5. **Deep belief networks (DBNs)**: These networks consist of multiple layers of hidden units that are trained in an unsupervised manner using restricted Boltzmann machines. DBNs are used for tasks such as image recognition, speech recognition, and natural language processing.

Feedforward neural network is perfect for simple classifications, which is a good fit to our dataset.
RNN also works well on classifications with sequential data, which also works for our task.

CNN is better for image classification, **and autoencoder neural networks**, **DBN**s are unsupervised learning, which works on processing the data, so we will not use these 3 this time.

Printout of the network using free_forward architecture with 2 hidden layers:

```
Layer 1 :
Name: dense
Input shape: (None, 11)
Output shape: (None, 32)
Activation: <function relu at 0x13fe5c0d0>
Layer 2 :
Name: dense_1
Input shape: (None, 32)
Output shape: (None, 16)
Activation: <function relu at 0x13fe5c0d0>
Layer 3 :
Name: dense_2
Input shape: (None, 16)
Output shape: (None, 1)
Activation: <function sigmoid at 0x13fe5c5e0>
```

Result:

```
Test loss: 0.5163913369178772
Test accuracy: 0.8349999785423279
7/7 [==============================] - 0s 745us/step
confusion matrix:
[[73 22]
 [11 94]]
```

In this network, there are 2 fully connected layers with ReLU function, and A final fully connected dense layer with a single output unit and a sigmoid activation function which is perfect for binary classification.

And the accuracy is 83.5%, which is lower than what we had in the previous project.

Printout of network using free_forward architecture with 3 hidden layers and dropout rate with 0.2:

```
Layer 1 :
Name: dense
Input shape: (None, 11)
Output shape: (None, 32)
Activation: <function relu at 0x1489f1160>
Layer 2 :
Name: dense_1
Input shape: (None, 32)
Output shape: (None, 16)
Activation: <function relu at 0x1489f1160>
Layer 3 :
Name: dense_2
Input shape: (None, 16)
Output shape: (None, 8)
Activation: <function relu at 0x1489f1160>
Layer 4 :
Name: dropout
Input shape: (None, 8)
Output shape: (None, 8)
```

Result:

```
Test loss: 0.38252395391464233
Test accuracy: 0.8450000286102295
7/7 [==============================] - 0s 706us/step
confusion matrix:
[[74 21]
 [10 95]]
```

In this network, there are 3 fully connected layers with ReLU function, and A final fully connected dense layer with a single output unit and a sigmoid activation function which is perfect for binary classification.
And the accuracy is 84.5%, which is slightly higher than the one with 2 hidden layers.

Printout of network using RNN with simpleRNN:

```
Layer 1 :
Name: simple_rnn
Input shape: (None, 1, 11)
Output shape: (None, 64)
Activation: <function relu at 0x1467be160>
Layer 2 :
Name: dense
Input shape: (None, 64)
Output shape: (None, 1)
Activation: <function sigmoid at 0x1467be670>
```

Result:

```
Test loss: 0.41815099120140076
Test accuracy: 0.8500000238418579
7/7 [==============================] - 0s 1ms/step
confusion matrix:
[[78 17]
 [13 92]]
```

In this network, there is A Simple RNN layer with 64 units and a ReLU activation function. This layer takes an input of shape (1, X_train.shape[2]), which means it expects input data with a sequence length of 1 and the number of features defined by the third dimension of the input data.
Also A fully connected dense layer with a single output unit and a sigmoid activation function. Which is perfect for binary classification.
And the accuracy is 85%, which is slightly higher than the one with 3 hidden layers.

Printout of network using RNN with LSTM:

```
Layer 1 :
Name: lstm
Input shape: (None, 1, 11)
Output shape: (None, 64)
Activation: <function relu at 0x13d84d160>
Layer 2 :
Name: dense
Input shape: (None, 64)
Output shape: (None, 1)
Activation: <function sigmoid at 0x13d84d670>
```

Result:

```
Test loss: 0.4498875141143799
Test accuracy: 0.8399999737739563
7/7 [==============================] - 0s 885us/step
confusion matrix:
[[77 18]
 [14 91]]
```

In this network, we are using the LSTM layer instead of the Simple RNN layer, with the rest of them staying the same.
And the accuracy is 84%, which is slightly lower than the one with the SimpleRNN layer.

Extension task 1: ImageNet Transfer Learning tutorial

In this part, we followed the imageNet transfer learning tutorial, and used a pre-trained network and then froze the weights for all the network layers except for the final layer. Then we trained the last layer in order to increase the accuracy.
So the final network model will fit into a new problem with a new layer.

Results:

```
Epoch 17/24
----------
train Loss: 0.2547 Acc: 0.8893
val Loss: 0.1573 Acc: 0.9346

Epoch 18/24
----------
train Loss: 0.2121 Acc: 0.9139
val Loss: 0.1643 Acc: 0.9216

Epoch 19/24
----------
train Loss: 0.3045 Acc: 0.8730
val Loss: 0.1544 Acc: 0.9281

Epoch 20/24
----------
train Loss: 0.2514 Acc: 0.8689
val Loss: 0.1715 Acc: 0.9216

Epoch 21/24
----------
train Loss: 0.2485 Acc: 0.8934
val Loss: 0.1764 Acc: 0.9085

Epoch 22/24
----------
train Loss: 0.2234 Acc: 0.8975
val Loss: 0.1676 Acc: 0.9281

Epoch 23/24
----------
train Loss: 0.2600 Acc: 0.8975
val Loss: 0.1617 Acc: 0.9346

Epoch 24/24
----------
train Loss: 0.2739 Acc: 0.8566
val Loss: 0.2198 Acc: 0.8954
```

```
Epoch 18/24
----------
train Loss: 0.3145 Acc: 0.8320
val Loss: 0.1797 Acc: 0.9477

Epoch 19/24
----------
train Loss: 0.3292 Acc: 0.8443
val Loss: 0.1942 Acc: 0.9477

Epoch 20/24
----------
train Loss: 0.4325 Acc: 0.8115
val Loss: 0.1769 Acc: 0.9412

Epoch 21/24
----------
train Loss: 0.2741 Acc: 0.8566
val Loss: 0.1978 Acc: 0.9477

Epoch 22/24
----------
train Loss: 0.3274 Acc: 0.8566
val Loss: 0.2016 Acc: 0.9346

Epoch 23/24
----------
train Loss: 0.2930 Acc: 0.8689
val Loss: 0.1719 Acc: 0.9542

Epoch 24/24
----------
train Loss: 0.3566 Acc: 0.8320
val Loss: 0.1961 Acc: 0.9477

Training complete in 45m 6s
Best val Acc: 0.960784
```

Extension task 2: Network for recognizing alphabet letters

Network printout

```
Net(
  (conv1): Conv2d(1, 10, kernel_size=(5, 5), stride=(1, 1))
  (conv2): Conv2d(10, 20, kernel_size=(6, 6), stride=(1, 1))
  (conv2_drop): Dropout2d(p=0.5, inplace=False)
  (fc1): Linear(in_features=720, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
)
```

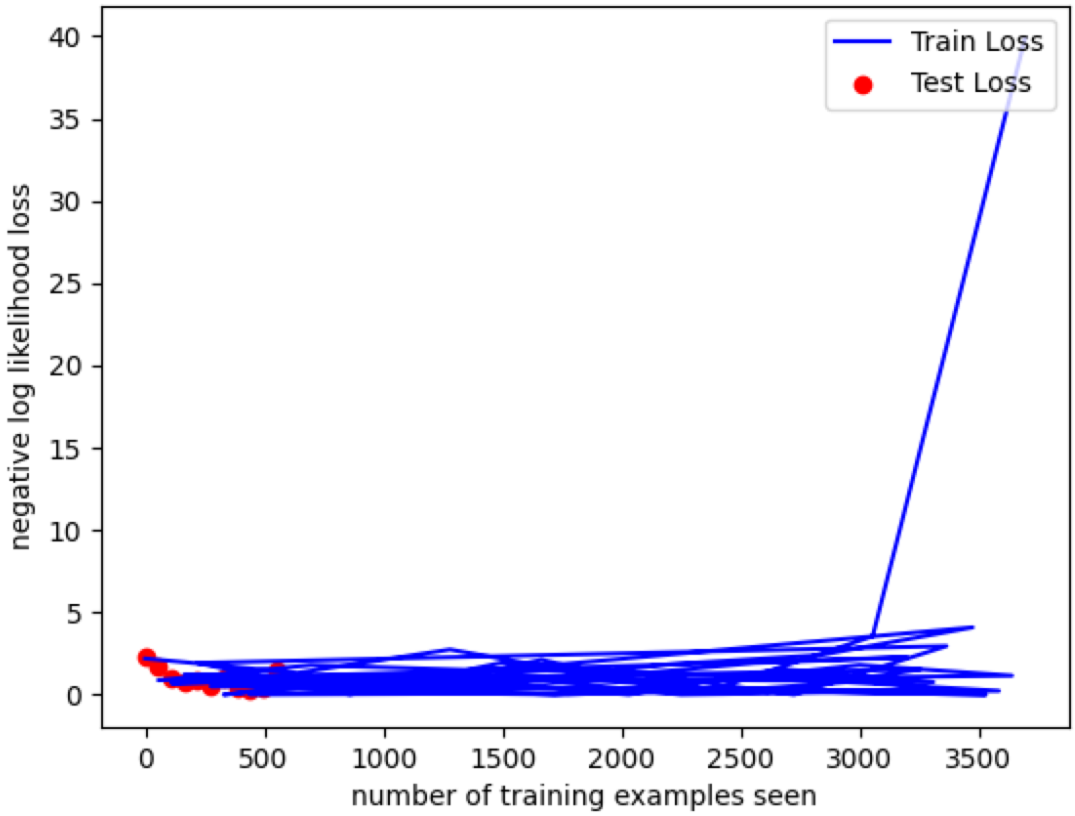
Try creating network to recognize alphabet letters (3 layers get 92% accuracy top then start overfit):
Try 1: start 300*300, add 4 layers into 16*16, 25% accuracy
Try 2: start 300*300, add 5 layers into 4*4, 25% accuracy
Try 3: start 144*144, add 3 layers into 2*2, 25% accuracy
Try 4: start 144*144, add 3 layers into 4*4, 55% accuracy
Try 5: start 144*144, add 2 layers into 6*6, 92% accuracy

Based on the results, I found out that because the alphabet line in my training set is too thin and the whole picture size too large, causing it hard for the model to focus on the features of the alphabet. So I started cropping the alphabet from the whole picture with 144*144, and the final pixels can't be too small because it will lose the feature again. The conclusion is to crop the picture to make the alphabet fill the window and do not add too many layers to make the final picture too small.

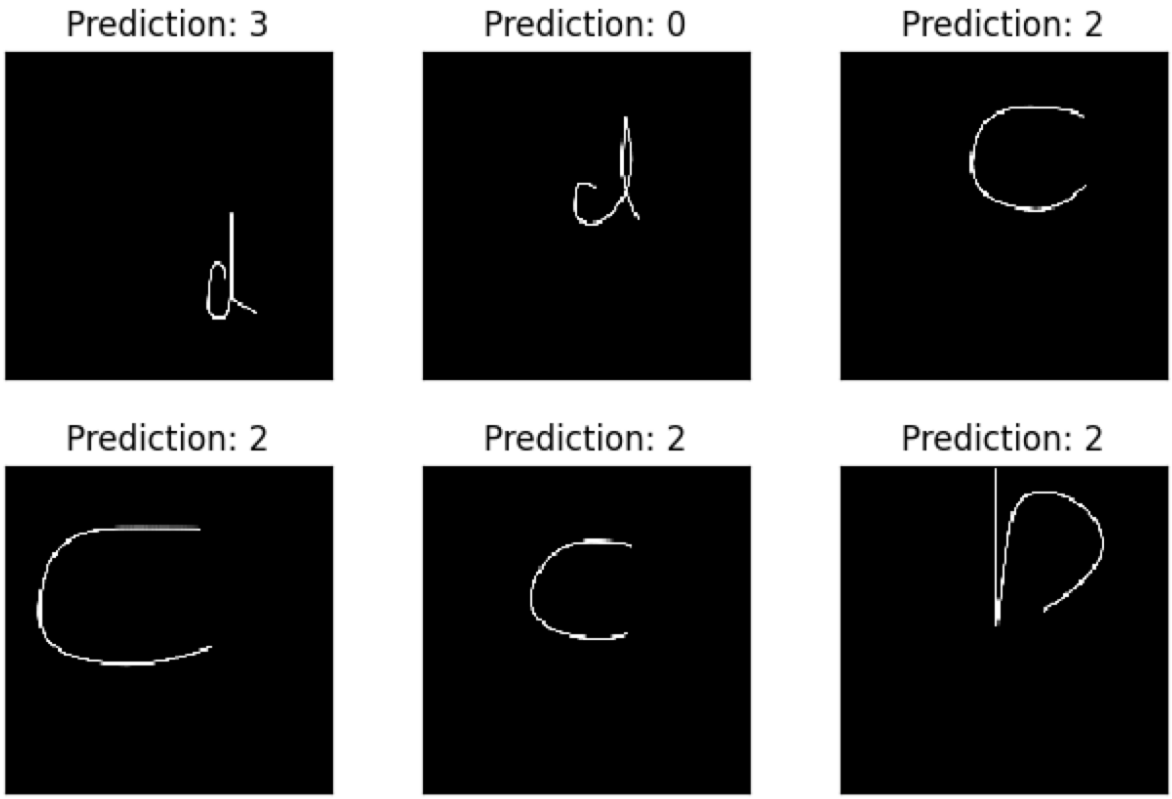
Training epochs

```
Train Epoch: 8 [0/55 (0%)]  Loss: 0.781461
Train Epoch: 8 [10/55 (18%)]    Loss: 0.214252
Train Epoch: 8 [20/55 (36%)]    Loss: 2.111169
Train Epoch: 8 [30/55 (55%)]    Loss: 0.078840
Train Epoch: 8 [40/55 (73%)]    Loss: 0.761601
Train Epoch: 8 [50/55 (91%)]    Loss: 0.232247

Test set: Avg. loss: 0.2691, Accuracy: 55/60 (92%)

Train Epoch: 9 [0/55 (0%)]  Loss: 0.000001
Train Epoch: 9 [10/55 (18%)]    Loss: 0.379323
Train Epoch: 9 [20/55 (36%)]    Loss: 0.000000
Train Epoch: 9 [30/55 (55%)]    Loss: 0.798983
Train Epoch: 9 [40/55 (73%)]    Loss: 1.803125
Train Epoch: 9 [50/55 (91%)]    Loss: 1.157155

Test set: Avg. loss: 0.4352, Accuracy: 54/60 (90%)

Train Epoch: 10 [0/55 (0%)] Loss: 0.611668
Train Epoch: 10 [10/55 (18%)]   Loss: 0.117609
Train Epoch: 10 [20/55 (36%)]   Loss: 0.000000
Train Epoch: 10 [30/55 (55%)]   Loss: 0.446217
Train Epoch: 10 [40/55 (73%)]   Loss: 3.547745
Train Epoch: 10 [50/55 (91%)]   Loss: 39.885586

Test set: Avg. loss: 1.4156, Accuracy: 31/60 (52%)
```

Training and testing loss



Example output



Summary

In this project, we have learned several important concepts and techniques in working with convolutional neural networks (CNNs) and other artificial neural network (ANN) architectures. We have gone through the basic process of running CNNs and ANNs in Python using PyTorch and have experimented with different dimensions and architectures to optimize the performance of the networks for various tasks.
Some of the key takeaways from this project are:

- Understanding the importance of selecting appropriate dimensions and hyperparameters for the networks. By experimenting with different dimensions, we were able to improve the performance of our models, find the optimal balance between accuracy and training time, and prevent overfitting.
- Gaining experience in transfer learning, which allowed us to leverage the knowledge from pre-trained models to improve the performance of our networks on new tasks, such as Greek letter recognition.
- Exploring different ANN architectures and their applications in various tasks, including feedforward networks for classification, recurrent networks for sequential data, and convolutional networks for image processing.

- Implementing data augmentation techniques and preprocessing strategies to improve the performance of our networks on new tasks, such as alphabet recognition.

Overall, this project has provided us with a comprehensive understanding of the principles and techniques involved in working with CNNs and ANNs. It has also demonstrated the importance of experimentation, optimization, and adaptability in achieving high performance in various tasks. Going forward, these insights will be invaluable in applying deep learning techniques to solve real-world problems and challenges.