1 Question 1

- Filter all samples representing digits '0' or '1' from the MNIST datasets.
- Randomly split the training data into a training set (80% training samples) of a validation set (20% training samples).
- Define an MLP with 1 hidden layer and train the MLP to classify the digits '0' vs '1'. Report your MLP design and training details (which optimizer, number of epochs, learning rate, etc.)
- Keep other hyper-parameters the same, and train the model with different batch sizes: 2, 16, 128, 1024. Report the time cost, training, validation and test set accuracy of your mode

Solution:

1.1 Model for binary classificaiton

```
print(model)

SimpleMLP(
    (fc1): Linear(in_features=784, out_features=5, bias=True)
    (activation): Sigmoid()
    (fc2): Linear(in_features=5, out_features=2, bias=True)
)
```

Effect of batch performance on the model performance and train time

Batch size	Training Time (s)	Train Acc	Val Acc	Test Acc
2	16.724894	99.911173	99.684169	99.669031
16	5.800792	99.960521	99.842084	99.952719
128	4.362510	99.891433	99.881563	99.905437
1024	3.937332	99.595341	99.526253	99.574468

Table 1: Comparing the performance of SimpleMLP model for different batch sizes 2, 16, 128, 1024

□

2 Question 2

- Implement the training loop and evaluation section. Report the hyper-parameters you choose
- Experiment with different numbers of neurons in the hidden layer and note any changes in performance.
- Write a brief analysis of the model's performance, including any challenges faced and how they were addressed

<u>**Solution:**</u>

2.1 Model for multi-class (10 digits) classificaiton

```
print(model)
hidden_dim = int(np.sqrt(28*28*10)) # changed in hyperopt

MulticlassMLP(
    (fc1): Linear(in_features=784, out_features=88, bias=True)
    (activation): Sigmoid()
    (fc2): Linear(in_features=88, out_features=10, bias=True)
)

# criterion = nn.CrossEntropyLoss()
```

2.2 Hyper parameter optimization for Multi-Class Optimization

I first experimented with device to see the most optimal device for training between cpu and the gpu on my Mac-M1 Pro. The results are given in the appendix for the model with sigmoid. For each of them we found that **device** = **'cpu'** was faster than **device** = **'mps'** (mac gpu in M1 Pro).

2.2.1 Results:

Two models were optimized for this task, one with sigmoid activation and the other with ReLU activation.

1. **Sigmoid Activation Layer**

```
print(model)
hidden_dim = int(np.sqrt(28*28*10))
# hidden_dim changed in hyperopt later
MulticlassMLP(
    (fc1): Linear(in_features=784, out_features=88,
    bias=True)
    (activation): Sigmoid()
    (fc2): Linear(in_features=88, out_features=10,
    bias=True)
)
# criterion = nn.CrossEntropyLoss()
```

The hyper-parameters tested are as follows

```
batch_sizes = [64, 128, 1024]
optimizers = ['adam', 'sgd']
# learning_rates= [1e-4, 1e-3, 1e-2, 1e-1]
hidden_dims = [4, 32, 64, 128]
```

Opt#	Batch	Opt	LR	HidDim	Training Time	Train Acc	Val Acc	Test Acc
0	64	adam	0.001	4	20.822481	81.116667	80.50	81.34
1	64	adam	0.001	32	21.138930	95.725	94.583333	94.83
2	64	adam	0.001	64	22.073092	97.208333	95.566667	96.09
3	64	adam	0.001	128	22.682774	98.10	96.591667	96.87
4	64	sgd	0.01	4	19.016016	69.941667	70.083333	70.49
5	64	sgd	0.01	32	19.240644	89.883333	89.666667	90.33
6	64	sgd	0.01	64	19.526299	90.029167	89.866667	90.57
7	64	sgd	0.01	128	20.448700	90.037500	89.85	90.69
8	128	adam	0.001	4	18.587870	74.941667	73.933333	75.06
9	128	adam	0.001	32	18.807421	95.029167	94.341667	94.47
10	128	adam	0.001	64	19.121340	96.412500	95.416667	95.75
11	128	adam	0.001	128	20.281624	97.404167	96.433333	96.45
12	128	sgd	0.01	4	17.898255	63.181250	63.475	64.90
13	128	sgd	0.01	32	17.976289	87.287500	87.291667	88.10
14	128	sgd	0.01	64	18.428340	87.922917	87.775	88.92
15	128	sgd	0.01	128	18.753883	87.927083	87.90	88.73
16	1024	adam	0.001	4	17.380155	52.65	52.416667	53.09
17	1024	adam	0.001	32	17.465012	90.939583	90.625	91.21
18	1024	adam	0.001	64	17.708344	92.270833	91.866667	92.49
19	1024	adam	0.001	128	18.143819	93.297917	92.766667	93.21
20	1024	sgd	0.01	4	17.723570	20.575	22.00	23.66
21	1024	sgd	0.01	32	17.807459	62.589583	63.075	65.17
22	1024	sgd	0.01	64	18.086973	67.327083	66.508333	68.73
23	1024	sgd	0.01	128	18.751791	70.679167	70.291667	71.88

Table 2: Hyperopt results for different optmizers, learning rate, batch size, and hidden dimension of the MulticlassMLP Network with sigmoid activation layer

2. **ReLU Activation Layer**

```
class MulticlassMLP(nn.Module):
    def __init__(self, in_dim, hidden_dim, out_dim):
        super(MulticlassMLP, self).__init__()
        self.fc1 = nn.Linear(in_dim, hidden_dim)
        self.activation = nn.ReLU()
        self.fc2 = nn.Linear(hidden_dim, out_dim)

    def forward(self, x):
        # Your code goes here
        x = self.fc1(x)
        x = self.activation(x)
```

```
        x = self.fc2(x)

        return x

# criterion = nn.CrossEntropyLoss()
```

Opt#	Batch	Opt	LR	HidDim	Training Time	Train Acc	Val Acc	Test Acc
0	64	adam	0.001	4	22.202067	33.362500	33.991667	34.13
1	64	adam	0.001	32	22.753937	95.647917	94.566667	95.12
2	64	adam	0.001	64	23.678958	97.191667	96.525	96.80
3	64	adam	0.001	128	24.853068	98.108333	96.425	96.53
4	64	sgd	0.01	4	21.199240	82.979167	82.091667	82.65
5	64	sgd	0.01	32	23.314560	92.931250	92.458333	92.94
6	64	sgd	0.01	64	24.677200	93.55	93.05	93.49
7	64	sgd	0.01	128	23.671364	93.885417	93.608333	94.10
8	128	adam	0.001	4	20.069331	65.714583	65.108333	66.38
9	128	adam	0.001	32	21.104044	93.787500	92.925	93.34
10	128	adam	0.001	64	22.600291	96.437500	95.45	95.97
11	128	adam	0.001	128	24.154394	97.735417	96.30	96.72
12	128	sgd	0.01	4	20.192016	81.056250	80.933333	81.43
13	128	sgd	0.01	32	19.052993	91.445833	91.433333	92.05
14	128	sgd	0.01	64	19.173184	91.566667	91.241667	91.92
15	128	sgd	0.01	128	20.031956	91.764583	91.458333	92.31
16	1024	adam	0.001	4	17.754258	40.585417	41.825	41.90
17	1024	adam	0.001	32	17.710765	91.395833	91.466667	91.97
18	1024	adam	0.001	64	18.001581	93.037500	92.708333	93.05
19	1024	adam	0.001	128	18.118424	94.625	94.158333	94.60
20	1024	sgd	0.01	4	17.477235	53.318750	53.15	54.65
21	1024	sgd	0.01	32	17.538370	85.191667	84.941667	85.86
22	1024	sgd	0.01	64	17.841422	86.214583	86.075	87.12
23	1024	sgd	0.01	128	18.008393	86.645833	86.491667	87.38

Table 3: Hyperopt results for different optmizers, learning rate, batch size, and hidden dimension of the MulticlassMLP Network with ReLU activation layer

2.2.2 Conclusion

From 2 and 3, we see that the highlighted rows.
The **yellow** rows highlight the highest performing hyper-parameters for 'adam', highest performing hyper-parameters for 'sgd' are highlighted in **orange**, and **red** highligts the worse performances across the two optimizers.

Observations:

- Sigmoid Activation
  - For batch size, optimizer, learning rate, hidden dimension, training time as follows 64, adam, 0.00, 128, 22.682, we get a train accuracy of 98.10%, validation accuracy of 96.59%, and the test accuracy of 96.87%. For batch size, optimizer, learning rate, hidden dimen-

sion, training time as follows 64, sgd, 0.01, 128, 20.4487, we get a train accuracy of 90.04%, validation accuracy of 89.85%, and the test accuracy of 90.69%.

Across this we note that 'adam' gets slightly higher test accuracy than 'sgd'. We also see that 128 hidden neurons do slightly better across the test set for both optimizers. This highlights that it is better able to not only capture the train data patterns, but also does well with test data. This shows that the given range of 16 64 might not be best suited for our multi class problem.

For batch size, optimizer, learning rate, hidden dimension, training time as follows 128, adam, 0.001, 128, 20.2816, wget a train accuracy of 97.40%, validation accuracy of 96.43%, and the test accuracy of 96.45%.

- Here we see that the batch size of 64 does slightly better than the batch size of 128, while also being slightly faster. However, the difference in performance is not statistically enough.

- In the red highlighted rows of 2, we see that Hidden dimension of 4 does worst across the board, which makes sense given the number of neurons are much lesser than what we expect to capture the complexity of the 10 class classification problem.

- ReLU Activation
  - For batch size, optimizer, learning rate, hidden dimension, training time as follows 64, adam, 0.00, 128, 22.68, we get a train accuracy of 97.19%, validation accuracy of 96.52%, and the test accuracy of 96.80%.

    For batch size, optimizer, learning rate, hidden dimension, training time as follows 64, sgd, 0.01, 128, 23.6713, we get a train accuracy of 93.88%, validation accuracy of 93.60%, and the test accuracy of 94.10%.

    For batch size, optimizer, learning rate, hidden dimension, training time as follows 128, adam, 0.001, 128, 24.154, we get a train accuracy of 97.73%, validation accuracy of 93.60%, and the test accuracy of 96.72%.

    Across this we note that 'adam' gets slightly higher test accuracy than 'sgd'. We also see that 128 hidden neurons do slightly worse across the test set for both optimizers, but better for training accuracy. This highlights that we might be over fitting on the training dataset for adam.

    - We also see that ReLU activation does slightly worse than sigmoid which is not I had expected. This hightlights that there is no universally best performing activation function. However, again the difference is not what we will call statistically significant.

    - Here we see that the batch size of 64 does slightly better than the batch size of 128, while also being slightly faster. However, the difference in performance is not statistically enough.

    - In the red highlighted rows of 3, we see that Hidden dimension of 4 does worst across the board, which makes sense given the number of neurons are much lesser than what we expect to capture the complexity of the 10 class classification problem.
- Challenges: As such there were no specific challenges other than the compute time that this grid based hyper parameter optimization took. To reduce the timing I tested both cpu and

gpu times and to my surprise, cpu was significally faster. After choosing the faster device, the whole process for the given number of hyper parameters took around 15 minutes for each network. So in total around 30 for both sigmoid and relu networks combined together.

□

3 Question 3

- Create the imbalance datasets with all "0" digits and only 1% "1" digits
- Implement the training loop and evaluation section (implementing the F_1 metric)
- Ignore the class imbalance problem and train the MLP. Report your hyper-parameter details and the F_1 score performance on the test set (as the baseline)
- Explore modifications to improve the performance of the class imbalance problem. Report your modifications and the F_1 scores performance on the test set.
- Can you propose new ways for the class imbalance problem and achieve stable and satisfactory performance for large N = 500, 1000, ...?

<u>**Solution:**</u>

Extra credit (exploring the performance of the network for large N is done with the small N of 100, look at the tables for more details)

Model used was the same as the one used in question 1 for binary classifciation.

```
SimpleMLP(
  (fc1): Linear(in_features=784, out_features=4, bias=True)
  (activation): ReLU()
  (fc2): Linear(in_features=4, out_features=2, bias=True)
)
```

3.0.1 Creation of Imbalanced Dataset where we sample every N^{th} point

We vary N from 100, then from 250 to 2000 with increments of 250 each. Thus the list $\mathrm{N}_{\mathrm{list}}$ varies from 100, 250, 500,..., 2000

```
train_0_original = [data for data in mnist if data[1] == 0]
train_1_original = [data for data in mnist if data[1] == 1]

# List of Ns (we sample every Nth point from list of 1s)
N_list = [100] + [250*(i+1) for i in range(8)]

for N in N_list:
    train_0 = train_0_original.copy()
    train_1 =  train_1_original.copy()
    random.shuffle(train_1)
    train_1 = train_1[:len(train_1) // N]
    print(N, 'Train set (before sparsing)',
     len(train_0), len(train_1), len(train_1) + len( train_0) )

    # Split training data (1s)into training and validation sets
    train_1len = int(len(train_1) *.8)
    val_1len = len(train_1) - train_1len
    train1_set, val1_set = random_split(train_1, [train_1len, val_1len])
```

```
    # Split training data (0s) into training and validation sets
    train_0len = int(len(train_0) *.8)
    val_0len = len(train_0) - train_0len
    train0_set, val0_set = random_split(train_0, [train_0len, val_0len])

    # combining 0 and 1s to get train and val sets
    train_set = train0_set + train1_set
    val_set = val0_set + val1_set
    len(train_set), len(val_set)

    # creating test set
    test_0 = [data for data in mnist_test if data[1] == 0]
    test_1 = [data for data in mnist_test if data[1] == 1]
    print(N,'Test set (before sparsing)',
    len(test_0), len(test_1), len(test_1) + len( test_0) )

    test_1 = test_1[:len(test_1) // N]
    print(N,'Test set (after sparsing)'
    ,len(test_0), len(test_1), len(test_1) + len( test_0) )
    test_set = test_0 + test_1
    print('\n')

    # Define DataLoaders to access data in batches
    train_loader = DataLoader(train_set, batch_size=64, shuffle=True)
    val_loader = DataLoader(val_set, batch_size = 64, shuffle=False)
    test_loader = DataLoader(test_set, batch_size = 64, shuffle=False)
```

3.0.2 Implement F_1 metric

F_1 Function

```
def precision_score(labels, predictions):
    predictions, labels = np.array(labels), np.array(predictions)
    predictions_1 = np.sum(predictions==1)
    correct_1 = np.sum( (predictions==1) & (labels==1))
    precision = correct_1/ predictions_1 if predictions_1 > 0 else 1e-6
    return precision

def recall_score(labels, predictions):
    predictions, labels = np.array(labels), np.array(predictions)
    correct_1 = np.sum( (predictions==1) & (labels==1))
    labels_1 = np.sum(labels==1)
    recall = correct_1/ labels_1 if labels_1 > 0 else 1e-6
    return recall

def f1_score(labels, predictions):
    precision = precision_score(labels, predictions)
```

```
    recall = recall_score(labels, predictions)
    f1 = (2 * (recall * precision)) / (precision + recall)
    return f1
```

Now we implement this in the val and test loops:

Validation Loop

```
# validation
val_loss = count = 0
correct = total = 0
val_preds = []; val_labels=[]
for data, target in val_loader:
    data, target = data.to(device), target.to(device)
    data = data.view(data.size(0), -1)
    output = model(data)
    val_loss += criterion(output, target).item()
    count += 1
    pred = output.argmax(dim=1)
    correct += (pred == target).sum().item()
    total += data.size(0)
    val_preds.append(pred)
    val_labels.append(target)
    # print(type(target))

# concat preds and true labels across all batches
val_preds = torch.cat(val_preds).numpy()
val_labels = torch.cat(val_labels).numpy()
assert len(val_preds) == len(val_set)

val_loss = val_loss / count
val_acc = 100. * correct / total
# print(f'Validation loss: {val_loss:.2f}, accuracy: {val_acc:.2f}%')
f1_validation = f1_score(labels = val_labels, predictions = val_preds)
# print(f'F1 score validation: {f1_validation:.2f}')
```

Test Loop

```
# test
model.eval()
correct = total = 0
test_preds = []; test_labels=[]

with torch.no_grad():
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)
        data = data.view(data.size(0), -1)
        output = model(data)
```

```
        pred = output.argmax(dim=1)
        correct += (pred == target).sum().item()
        total += data.size(0)
        test_preds.append(pred)
        test_labels.append(target)

# concat preds and true labels across all batches
test_preds = torch.cat(test_preds).numpy()
test_labels = torch.cat(test_labels).numpy()
assert len(test_preds) == len(test_set)
test_acc = 100. * correct / total
# print(f'Test Accuracy: {test_acc:.2f}%')
# print(f'Validation loss: {val_loss:.2f}, accuracy: {val_acc:.2f}%')
f1_test = f1_score(labels = test_labels, predictions =test_preds)
# print(f'F1 score test: {f1_test:.2f}')
```

3.0.3 Analysis of the model performance for different degrees of sparsity (larger N means more sparse dataset)

Structure:

For testing the performance, we use two different data sets. One is the original (unsparsed test dataset), the other sparsed data set (where we sample every Nth datapoint). The model is trained and validated on the sparse datasets but we test on the different datasets.

1. Performance on sparsed test data

	N	Batch size	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	0.489757	100.000000	99.916597	100.000000	0.962963	1.000000
1	250	64	0.375478	100.000000	100.000000	99.898374	1.000000	0.888889
2	500	64	0.357813	100.000000	100.000000	100.000000	1.000000	1.000000
3	750	64	0.361555	100.000000	100.000000	100.000000	1.000000	1.000000
4	1000	64	0.351796	100.000000	100.000000	100.000000	1.000000	1.000000
5	1250	64	0.411107	99.915647	100.000000	100.000000	1.000000	0.000001
6	1500	64	0.365109	99.957815	99.915683	100.000000	0.000000	0.000001
7	1750	64	0.350900	99.957806	99.915683	100.000000	0.000000	0.000001
8	2000	64	0.349054	99.957806	99.915683	100.000000	0.000000	0.000001

Figure 1: no modifications, test: sparsted data where N shows every Nth data point from the train, validation, and test datasets were sampled

2. Performance on original/unsparsed test data

	N	Batch size	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	0.411620	100.000000	100.000000	98.203310	1.0	0.982975
1	250	64	0.402712	100.000000	100.000000	98.392435	1.0	0.984794
2	500	64	0.390831	99.978939	100.000000	96.879433	1.0	0.970054
3	750	64	0.380769	100.000000	100.000000	94.562648	1.0	0.946636
4	1000	64	0.373118	100.000000	100.000000	93.900709	1.0	0.939748
5	1250	64	0.440504	99.915647	99.915683	46.335697	0.0	0.000000
6	1500	64	0.373520	100.000000	99.915683	55.082742	0.0	0.280303
7	1750	64	0.377439	99.957806	99.915683	46.335697	0.0	0.000000
8	2000	64	0.381415	99.957806	99.915683	46.335697	0.0	0.000000

Figure 2: no modifications, here N shows every Nth data point from the train, validation were sampled. Test data remained unchanged

3. Observations We see that the model despite sparsity in Fig 1 that eveen for sparsity of upto 1000, the model does really well in terms of F_1 score. $N = 250$ is an exception but it was just this specific run, meaning pictures that model found hard were selected which increased the average loss count compared to other runs.
   Despite having a very different distribution to the train and validation, in the case of the original unsparsed test data, we see in Fig 2 that the F_1 score is lesser (which is what we would expect given it no more follows the sparsed dataset distribution that our model is trained on). However, it is still acceptable for $N = 750$ hovering around $.95$.
   In both cases, we see that for $N > 1000$, the F_1 score drops considerably. In the case of sparsed dataset it drops down to 0 because there is no 1 in the test dataset. In the case of original dataset too we see a drop, that is somehow back up again for $N = 1500$.
   Thus, we see a huge bottle neck for the two methods at the 1000 threshold.

3.0.4 Adjusted Class Weights in the Loss Function: Aalysis of the model performance for different degrees of sparsity for different loss weights

Structure:

For testing the performance, we use two different data sets. One is the original (unsparsed test dataset), the other sparsed data set (where we sample every Nth datapoint). The model is trained and validated on the sparse datasets but we test on the different datasets.

Adjusting the weight in the loss function

```
# reweight_factor = weight[1]/ weight[0]
model = SimpleMLP(in_dim=28 * 28,
hidden_dim=hidden_dim,
out_dim=2).to(device)
criterion = nn.CrossEntropyLoss(weight = weight)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
num_epochs = 10
```

In the table, the $Weight$ means how much more the sparse class (**1**) was over weighted in the loss function in comparsion to 0.
For each of the N, four different weights were tried: $\left[1, \frac{N}{10}, \frac{N}{2}, \frac{len(train0)}{len(train1)}\right]$

1. Performance on sparsed test data

	N	Batch size	Weight	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	1.000000	0.441928	100.000000	100.000000	100.000000	1.000000	1.000000
1	100	64	10.000000	0.444450	99.979128	100.000000	100.000000	1.000000	1.000000
2	100	64	50.000000	0.425864	100.000000	100.000000	99.899092	1.000000	0.956522
3	100	64	89.396225	0.429034	100.000000	100.000000	100.000000	1.000000	1.000000
4	250	64	1.000000	0.421473	100.000000	99.916037	100.000000	0.909091	1.000000
5	250	64	25.000000	0.408722	100.000000	99.832074	100.000000	0.800000	1.000000
6	250	64	125.000000	0.438926	99.978983	99.916037	100.000000	0.909091	1.000000
7	250	64	236.899994	0.412361	99.831862	99.832074	99.796748	0.833333	0.800000
8	500	64	1.000000	0.411593	99.957877	99.831650	100.000000	0.500000	1.000000
9	500	64	50.000000	0.403737	100.000000	100.000000	100.000000	1.000000	1.000000
10	500	64	250.000000	0.469201	100.000000	99.831650	100.000000	0.500000	1.000000
11	500	64	473.799988	0.495480	99.957877	100.000000	100.000000	1.000000	1.000000
12	750	64	1.000000	0.460369	99.957841	99.831508	99.898063	0.000000	0.000000
13	750	64	75.000000	0.411340	100.000000	100.000000	100.000000	1.000000	1.000000
14	750	64	375.000000	0.432767	99.915683	100.000000	100.000000	1.000000	1.000000
15	750	64	789.666687	0.446825	99.873524	99.831508	99.898063	0.000000	0.000000
16	1000	64	1.000000	0.440238	99.915647	99.831508	99.898063	0.000000	0.000000
17	1000	64	100.000000	0.409171	99.978912	99.915754	100.000000	0.666667	1.000000
18	1000	64	500.000000	0.436480	99.810207	100.000000	99.898063	1.000000	0.666667
19	1000	64	1184.500000	0.456940	99.599325	100.000000	100.000000	1.000000	1.000000
20	1250	64	1.000000	0.486057	100.000000	99.915683	100.000000	0.000000	0.000001
21	1250	64	125.000000	0.412337	99.936736	100.000000	100.000000	1.000000	0.000001
22	1250	64	625.000000	0.428437	99.768030	100.000000	100.000000	1.000000	0.000001
23	1250	64	1184.500000	0.473218	99.852383	100.000000	99.897959	1.000000	0.000000
24	1500	64	1.000000	0.413333	100.000000	99.915683	100.000000	0.000000	0.000001
25	1500	64	150.000000	0.402057	100.000000	99.915683	100.000000	0.000000	0.000001
26	1500	64	750.000000	0.445018	100.000000	99.915683	100.000000	0.000000	0.000001
27	1500	64	1579.333374	0.461400	99.978907	99.915683	100.000000	0.000000	0.000001
28	1750	64	1.000000	0.456237	100.000000	100.000000	100.000000	1.000000	0.000001
29	1750	64	175.000000	0.409596	100.000000	100.000000	100.000000	1.000000	0.000001
30	1750	64	875.000000	0.445411	99.978903	99.831366	100.000000	0.500000	0.000001
31	1750	64	2369.000000	0.448272	100.000000	100.000000	100.000000	1.000000	0.000001
32	2000	64	1.000000	0.465735	99.957806	99.915683	100.000000	0.000000	0.000001
33	2000	64	200.000000	0.443340	100.000000	100.000000	100.000000	1.000000	0.000001
34	2000	64	1000.000000	0.429006	100.000000	100.000000	100.000000	1.000000	0.000001
35	2000	64	2369.000000	0.510838	100.000000	100.000000	100.000000	1.000000	0.000001

Figure 3: Adjusted Class weights in the Loss Function, test: sparsted data where N shows every Nth data point from the train, validation, and test datasets were sampled

2. Performance on original/unsparsed test data

	N	Batch size	Weight	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	1.000000	0.619131	98.893759	98.832360	46.335697	0.000000	0.000000
1	100	64	10.000000	0.482844	100.000000	100.000000	99.243499	1.000000	0.992902
2	100	64	50.000000	0.481757	98.893759	98.832360	46.335697	0.000000	0.000000
3	100	64	89.396225	0.439864	99.979128	100.000000	99.432624	1.000000	0.994690
4	250	64	1.000000	0.435156	100.000000	100.000000	97.966903	1.000000	0.980692
5	250	64	25.000000	0.391420	100.000000	99.916037	99.527187	0.923077	0.995575
6	250	64	125.000000	0.389516	100.000000	99.832074	99.243499	0.857143	0.992902
7	250	64	236.899994	0.395516	99.936948	99.916037	99.574468	0.923077	0.996019
8	500	64	1.000000	0.430612	100.000000	99.915825	93.522459	0.800000	0.935771
9	500	64	50.000000	0.424280	100.000000	100.000000	97.825059	1.000000	0.979317
10	500	64	250.000000	0.438843	99.978939	100.000000	98.439716	1.000000	0.985248
11	500	64	473.799988	0.409818	100.000000	100.000000	98.486998	1.000000	0.985702
12	750	64	1.000000	0.533480	99.873524	99.831508	46.335697	0.000000	0.000000
13	750	64	75.000000	0.532144	100.000000	100.000000	97.825059	1.000000	0.979317
14	750	64	375.000000	0.437707	99.873524	99.831508	46.335697	0.000000	0.000000
15	750	64	789.666687	0.527594	99.957841	100.000000	97.919622	1.000000	0.980234
16	1000	64	1.000000	0.464308	100.000000	99.831508	70.165485	0.000000	0.615009
17	1000	64	100.000000	0.387924	100.000000	99.915754	87.044917	0.666667	0.862725
18	1000	64	500.000000	0.405910	99.978912	99.915754	78.014184	0.666667	0.742382
19	1000	64	1184.500000	0.425349	99.831295	99.915754	99.621749	0.800000	0.996466
20	1250	64	1.000000	0.427798	100.000000	100.000000	61.371158	1.000000	0.437715
21	1250	64	125.000000	0.470915	100.000000	100.000000	63.971631	1.000000	0.494695
22	1250	64	625.000000	0.412594	100.000000	99.915683	79.810875	0.666667	0.768313
23	1250	64	1184.500000	0.442972	100.000000	99.915683	73.475177	0.666667	0.671738
24	1500	64	1.000000	0.423154	99.936722	99.915683	46.335697	0.000000	0.000000
25	1500	64	150.000000	0.542563	100.000000	100.000000	93.617021	1.000000	0.936768
26	1500	64	750.000000	0.501302	100.000000	100.000000	90.354610	1.000000	0.901258
27	1500	64	1579.333374	0.471869	99.831259	100.000000	99.338061	1.000000	0.993794
28	1750	64	1.000000	0.427371	99.957806	99.915683	46.335697	0.000000	0.000000
29	1750	64	175.000000	0.386647	100.000000	99.915683	96.973995	0.666667	0.970988
30	1750	64	875.000000	0.385367	100.000000	100.000000	80.189125	1.000000	0.773636
31	1750	64	2369.000000	0.405178	100.000000	100.000000	81.513002	1.000000	0.791911
32	2000	64	1.000000	0.449110	99.957806	99.915683	46.335697	0.000000	0.000000
33	2000	64	200.000000	0.385847	99.873418	99.578415	99.763593	0.285714	0.997798
34	2000	64	1000.000000	0.399684	100.000000	100.000000	71.347518	1.000000	0.635817
35	2000	64	2369.000000	0.408426	99.240506	100.000000	74.751773	1.000000	0.692396

Figure 4: Adjusted Class weights in the Loss Function, test: sparsted data where N shows every Nth data point from the train, validation. Test datasets was left as original test data set

3. Observations

   Across both the sparsed and the unsparsed dataset we see huge improvements in the F_1 scores for the validation dataset, which shows that the weighting works well. We note that the weight of 1 for each class does as expected (from previous Fig 3 till $N = 750$. But it drops down to 0 at $N \geqslant 1000$, because there is no data point belonging to 1 class in test dataset.

   We also note that in Fig 3, higher weights of $\frac{N}{10}, \frac{\mathtt{len(train0)}}{\mathtt{len(train1)}}$ for the sparse classes do decently well for $N = 1000$. They suffer the same problem for $N > 1000$ because the test set has no 1s Despite having a very different distribution to the train and validation, in the case of the original unsparsed test data, we see in Fig 4 that the F_1 score is lesser for Ns upto 1000 (which is what we would expect given it no more follows the sparsed dataset distribution that our model is trained on). However, we see that the higher weights do decently well till 1500. Beyond that we see that the best performing weight of $\frac{len(train0)}{len(train1)}$ becomes too large for it to do well, and we see the $\frac{N}{10}$ weight factor does better with the F_1 score for test.

3.0.5 Resampling in the Data Loader: Aalysis of the model performance for different degrees of sparsity for different resampling weights

Structure:

For testing the performance, we use two different data sets. One is the original (unsparsed test dataset), the other sparsed data set (where we sample every Nth datapoint). The model is trained and validated on the sparse datasets but we test on the different datasets.

Adjusting the weight in the loss function

```
train_set = train0_set + train1_set
val_set = val0_set + val1_set
random.shuffle(train_set)
random.shuffle(val_set)
len(train_set), len(val_set)

# creating test set
test_0 = [data for data in mnist_test if data[1] == 0]
test_1 = [data for data in mnist_test if data[1] == 1]
test_1 = test_1[:len(test_1) // N] # comment this out for the unsparsed
test_set = test_0 + test_1

test_loader = DataLoader(test_set, batch_size=64, shuffle=False)

compensation = int(train_0len/ train_1len)
weight_factors = [1, int(N/10), int(N/2), compensation]
batch_size = 64
results = []

for weight_factor in weight_factors:
    weights = np.array( [1.0 if data[1] == 0
    else weight_factor for data in train_set])
    weights = torch.from_numpy(weights)
    sampler = WeightedRandomSampler(weights, num_samples=len(weights),
    replacement=True)
    train_loader = DataLoader(train_set, batch_size=64, sampler=sampler)
    val_loader = DataLoader(val_set, batch_size=64, shuffle=False)
```

Note no shuffling in the false, this was inspired from **?**

In the table, the Weight means how much more the sparse class ($\mathbf{1}$) was over weighted in the loss function in comparsion to $\mathsf{0}$. For each of the N, four different weights were tried: $\left[1, \frac{N}{10}, \frac{N}{2}, \frac{len(train0)}{len(train1)}\right]$

1. Performance on sparsed test data

	N	Batch Size	Weight	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	1	0.413916	99.853893	99.833194	99.798184	0.923077	0.900000
1	100	64	10	0.453152	100.000000	99.916597	99.899092	0.962963	0.952381
2	100	64	50	0.399540	100.000000	100.000000	100.000000	1.000000	1.000000
3	100	64	89	0.443699	100.000000	100.000000	99.899092	1.000000	0.956522
4	250	64	1	0.402522	99.936948	99.916037	99.796748	0.909091	0.666667
5	250	64	25	0.469228	100.000000	100.000000	100.000000	1.000000	1.000000
6	250	64	125	0.377950	99.957966	100.000000	99.796748	1.000000	0.800000
7	250	64	236	0.404280	100.000000	100.000000	100.000000	1.000000	1.000000
8	500	64	1	0.471906	99.957877	99.915825	100.000000	0.800000	1.000000
9	500	64	50	0.491204	100.000000	100.000000	100.000000	1.000000	1.000000
10	500	64	250	0.412180	100.000000	100.000000	100.000000	1.000000	1.000000
11	500	64	473	0.554037	99.978939	100.000000	99.898167	1.000000	0.800000
12	750	64	1	0.524819	99.873524	99.831508	99.898063	0.000000	0.000000
13	750	64	75	0.536037	100.000000	100.000000	100.000000	1.000000	1.000000
14	750	64	375	0.478645	99.978921	100.000000	100.000000	1.000000	1.000000
15	750	64	789	0.361053	100.000000	100.000000	100.000000	1.000000	1.000000
16	1000	64	1	0.559330	99.810207	99.831508	99.898063	0.000000	0.000000
17	1000	64	100	0.439239	100.000000	100.000000	100.000000	1.000000	1.000000
18	1000	64	500	0.421760	100.000000	100.000000	100.000000	1.000000	1.000000
19	1000	64	1184	0.386407	100.000000	100.000000	100.000000	1.000000	1.000000
20	1250	64	1	0.481794	99.957824	99.915683	100.000000	0.000000	0.000001
21	1250	64	125	0.373254	100.000000	99.915683	100.000000	0.000000	0.000001
22	1250	64	625	0.383272	100.000000	99.915683	100.000000	0.000000	0.000001
23	1250	64	1184	0.401946	100.000000	99.915683	100.000000	0.666667	0.000001
24	1500	64	1	0.410053	99.873444	99.915683	100.000000	0.000000	0.000001
25	1500	64	150	0.395911	100.000000	100.000000	100.000000	1.000000	0.000001
26	1500	64	750	0.426304	99.978907	100.000000	99.897959	1.000000	0.000000
27	1500	64	1579	0.444462	100.000000	100.000000	100.000000	1.000000	0.000001
28	1750	64	1	0.421583	99.957806	99.915683	100.000000	0.000000	0.000001
29	1750	64	175	0.579675	100.000000	100.000000	100.000000	1.000000	0.000001
30	1750	64	875	0.450617	100.000000	100.000000	100.000000	1.000000	0.000001
31	1750	64	2369	0.410894	100.000000	100.000000	100.000000	1.000000	0.000001
32	2000	64	1	0.419797	99.978903	99.915683	100.000000	0.000000	0.000001
33	2000	64	200	0.513345	100.000000	99.915683	100.000000	0.000000	0.000001
34	2000	64	1000	0.508458	100.000000	100.000000	99.897959	1.000000	0.000000
35	2000	64	2369	0.455658	99.978903	100.000000	100.000000	1.000000	0.000001

Figure 5: Adjusted Class weights in the Loss Function, test: sparsted data where N shows every Nth data point from the train, validation, and test datasets were sampled

2. Performance on original/unsparsed test data

	N	Batch Size	Weight	Train Time	Train Acc	Val Acc	Test Acc	F1-Val	F1-Test
0	100	64	1	0.388137	99.979128	99.916597	98.817967	0.962963	0.988864
1	100	64	10	0.374013	100.000000	100.000000	99.716312	1.000000	0.997350
2	100	64	50	0.408951	100.000000	100.000000	99.858156	1.000000	0.998679
3	100	64	89	0.422724	100.000000	100.000000	99.479905	1.000000	0.995131
4	250	64	1	0.416720	99.600673	99.496222	46.335697	0.000000	0.000000
5	250	64	25	0.389752	100.000000	100.000000	97.541371	1.000000	0.976555
6	250	64	125	0.512834	100.000000	100.000000	99.763593	1.000000	0.997792
7	250	64	236	0.399923	99.978983	100.000000	99.763593	1.000000	0.997792
8	500	64	1	0.454007	99.831508	99.747475	46.335697	0.000000	0.000000
9	500	64	50	0.412547	100.000000	99.915825	95.366430	0.800000	0.954880
10	500	64	250	0.547272	100.000000	100.000000	99.338061	1.000000	0.993794
11	500	64	473	0.512915	99.894693	100.000000	99.763593	1.000000	0.997794
12	750	64	1	0.438797	99.873524	99.831508	46.335697	0.000000	0.000000
13	750	64	75	0.407191	100.000000	99.915754	76.548463	0.666667	0.720406
14	750	64	375	0.373476	100.000000	100.000000	97.494090	1.000000	0.976094
15	750	64	789	0.362313	100.000000	99.915754	92.482270	0.666667	0.924680
16	1000	64	1	0.385908	99.894559	99.831508	46.335697	0.000000	0.000000
17	1000	64	100	0.360009	100.000000	99.915754	81.323877	0.666667	0.789333
18	1000	64	500	0.417010	100.000000	100.000000	98.014184	1.000000	0.981166
19	1000	64	1184	0.453959	100.000000	100.000000	91.489362	1.000000	0.913876
20	1250	64	1	0.397826	99.957824	99.915683	46.335697	0.000000	0.000000
21	1250	64	125	0.391967	100.000000	100.000000	88.794326	1.000000	0.883424
22	1250	64	625	0.420383	100.000000	100.000000	99.290780	1.000000	0.993348
23	1250	64	1184	0.402556	100.000000	100.000000	99.716312	1.000000	0.997352
24	1500	64	1	0.436392	99.978907	99.915683	46.335697	0.000000	0.000000
25	1500	64	150	0.368916	100.000000	100.000000	75.839243	1.000000	0.709494
26	1500	64	750	0.334540	100.000000	100.000000	88.605201	1.000000	0.881222
27	1500	64	1579	0.341033	99.978907	100.000000	91.773050	1.000000	0.916985
28	1750	64	1	0.428044	99.957806	99.915683	46.335697	0.000000	0.000000
29	1750	64	175	0.398755	100.000000	100.000000	81.087470	1.000000	0.786096
30	1750	64	875	0.421275	100.000000	100.000000	96.690307	1.000000	0.968182
31	1750	64	2369	0.388738	100.000000	100.000000	96.643026	1.000000	0.967713
32	2000	64	1	0.421667	99.915612	99.915683	46.335697	0.000000	0.000000
33	2000	64	200	0.374887	100.000000	99.915683	59.952719	0.000000	0.404779
34	2000	64	1000	0.349644	100.000000	99.915683	85.248227	0.000000	0.840654
35	2000	64	2369	0.370446	100.000000	99.915683	93.853428	0.000000	0.939252

Figure 6: Adjusted Class weights in the Loss Function, test: sparsted data where N shows every Nth data point from the train, validation. Test datasets was left as original test data set

3. Observations

Across both the sparsed and the unsparsed dataset we see huge improvements in the F_1 scores for the validation dataset over no modifications, which shows that the resampling works well. We note that resampling for lower weights does better that adjusting weights in the loss.
We also note that in Fig 5, higher weights of for the sparse classes do decently well for $N = 1000$. They suffer the same problem for $N > 1000$ because the test set has no $1s$. However, over the weight adjustment for the loss function, we do not see a huge difference in the performance.
However, we do see a decent difference in the performance in resampling over loss weighting for large Ns in Fig 6. $N = 1750$ offers a good comparison where we see the validation F_1 of 1 and test F_1 of around $.97$ for a factor of $\frac{\mathtt{len(train0)}}{\mathtt{len(train1)}}$, whereas we had a test validation of $.80$ in the case of loss weighting. However, see that the validation score was very low for unsparsed split of $N = 2000$, it might be that the validation set did not have a 1 class.
However, see still see that the $F_1(\mathtt{test})$ for $N = 2000$ is still considerably good at $.94$.
Remark: We note that both weighting in the loss function and resampling in the Data loader offer considerable improvement over no modificaiton in the case of both original and sparsed test data sets. However, we see that the performance of a weighting factor in the data loader through resampling is a lot more consistent across different Ns than the weighting factor in the loss function. In the loss fuction, weighting the highest weights start off well, but we see that their performance drops off for large Ns. The intermediate weighting factors of $\frac{N}{2}, \frac{N}{10}$ start to perform better for more sparse data.
This makes sense given that weighting the loss by a very large number can make the optimization unstable as we weight a specific class a lot more in the optimization. On the other hand, resampling offers a smoother alternative to weighting the loss function, especially as the sparsity grows too large.

$\square$

4 Question 4

- Define an MLP with only one hidden layer and set the hidden layer dimension as 50. Train the MLP to reconstruct input images from all 10 digits
- Report the Mean Squared Error on the training, validation and test set. Report your hyper- parameter details.
- Pick 5 images for each digit from the test set. Visualize the original images and the reconstructed images using the MLP.

Solution:

4.1 Model for Reconstruction

```
RegressionMLP(
  (fc1): Linear(in_features=784, out_features=50, bias=True)
  (activation): ReLU()
  (fc2): Linear(in_features=50, out_features=784, bias=True)
  (activation_output): Tanh()
)
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
```

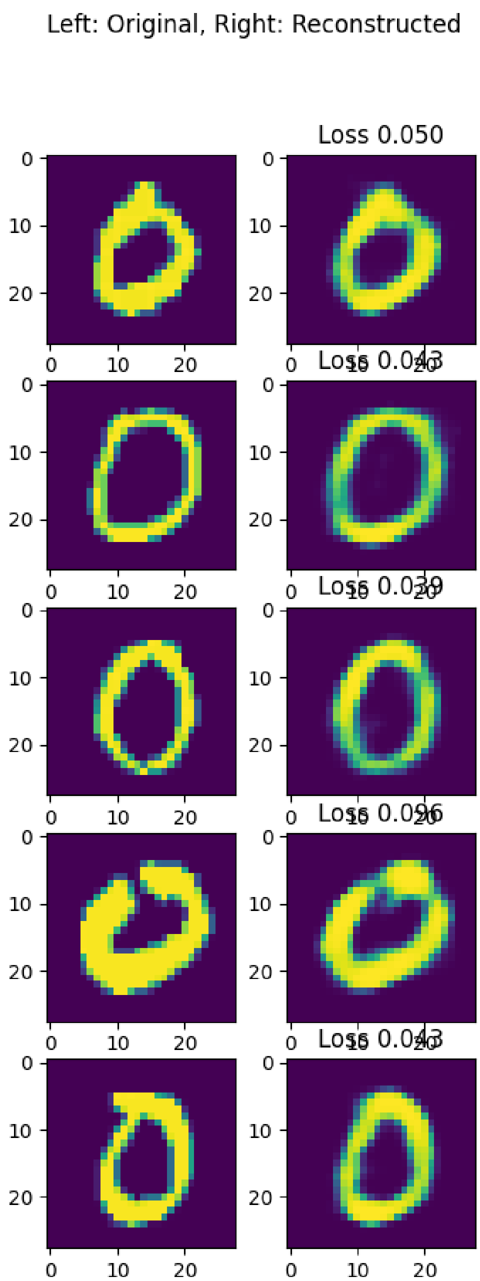
4.2 Reporting Train, Val, and Test MSE Loss

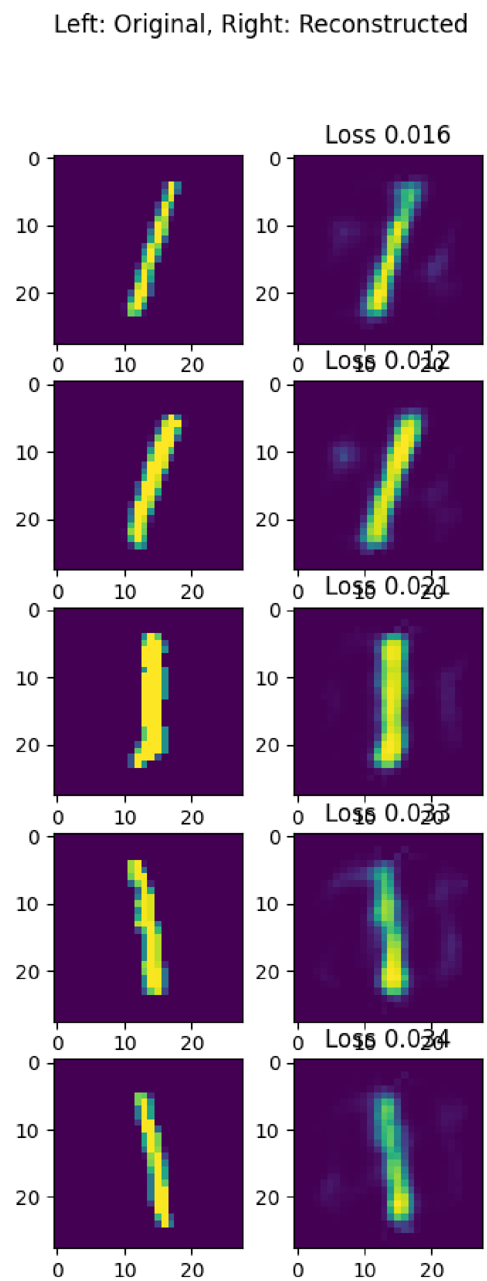
- Train Loss:
  Epoch 1, Loss: 0.0991
  Epoch 2, Loss: 0.0804
  Epoch 3, Loss: 0.0737
  Epoch 4, Loss: 0.0771
  Epoch 5, Loss: 0.0730
  Epoch 6, Loss: 0.0712
  Epoch 7, Loss: 0.0752
  Epoch 8, Loss: 0.0686
  Epoch 9, Loss: 0.0668
  Epoch 10, Loss: 0.0684
  Epoch 11, Loss: 0.0681
  Epoch 12, Loss: 0.0696
  Epoch 13, Loss: 0.0711
  Epoch 14, Loss: 0.0697
  Epoch 15, Loss: 0.0751
  Epoch 16, Loss: 0.0754
  Epoch 17, Loss: 0.0641
  Epoch 18, Loss: 0.0672
  Epoch 19, Loss: 0.0669

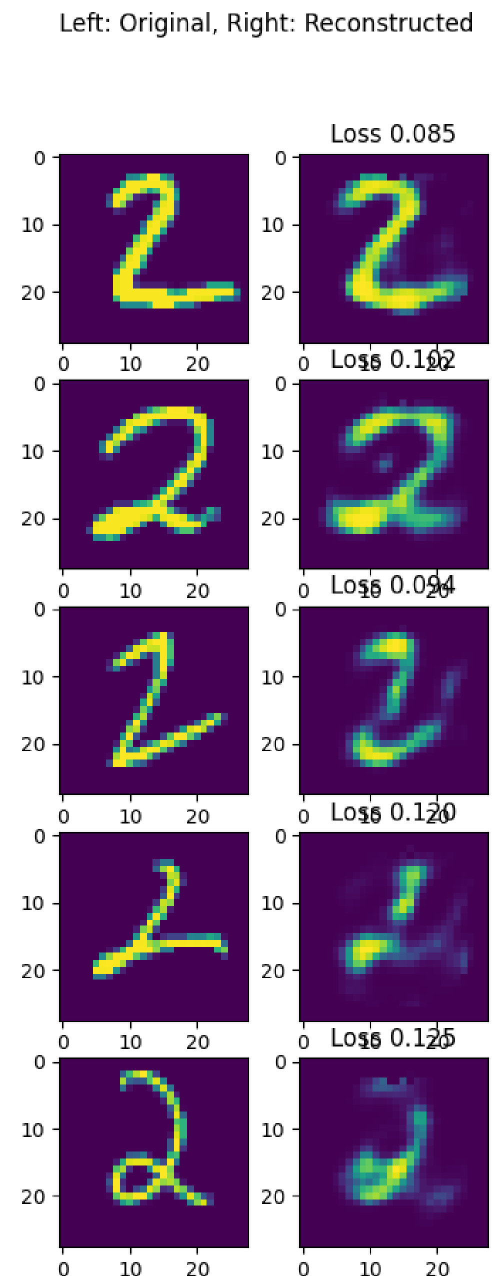
Epoch 20, Loss: 0.0687

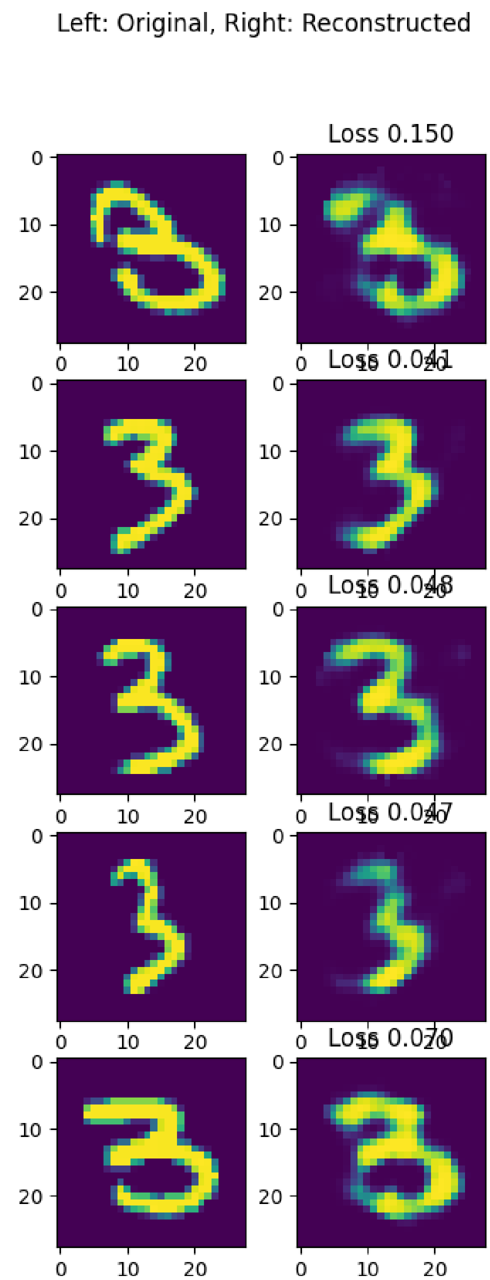
- Validation Loss: 0.07
- Test Loss: 0.07

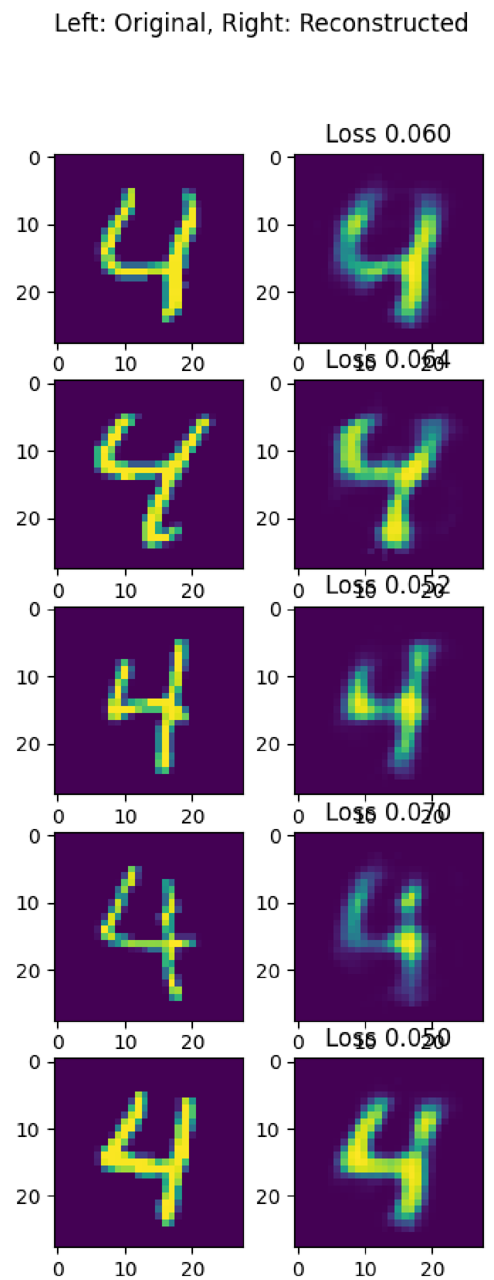
4.3 Reconstructed Images vs Original Images for Digits 0 to 9

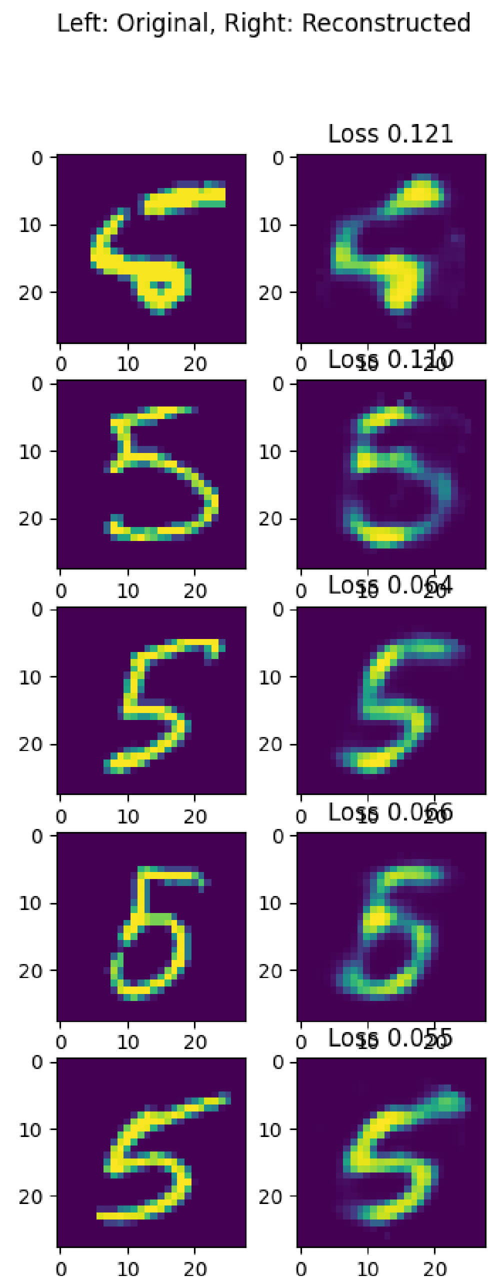


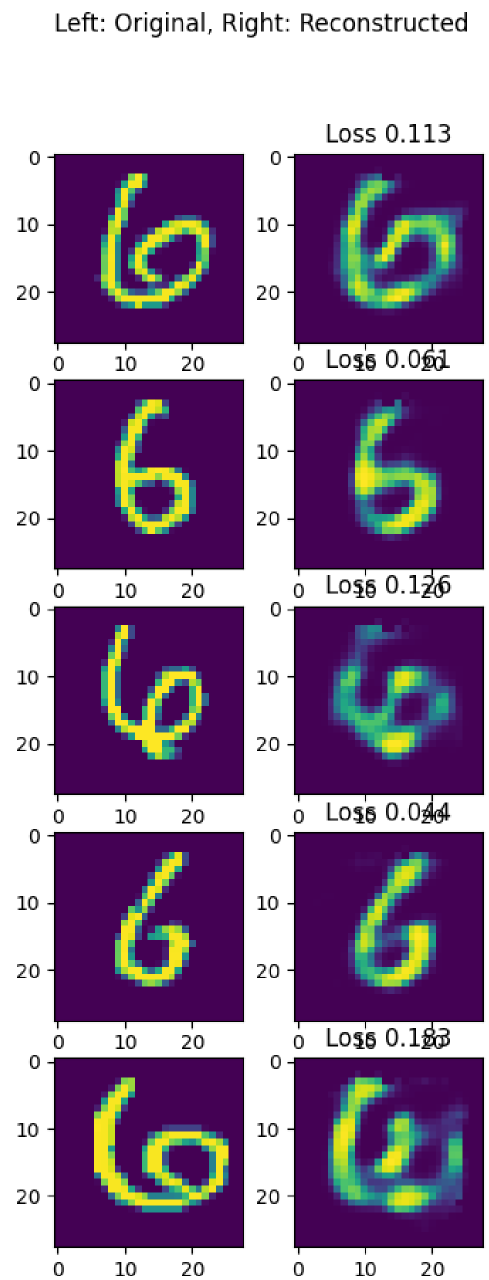


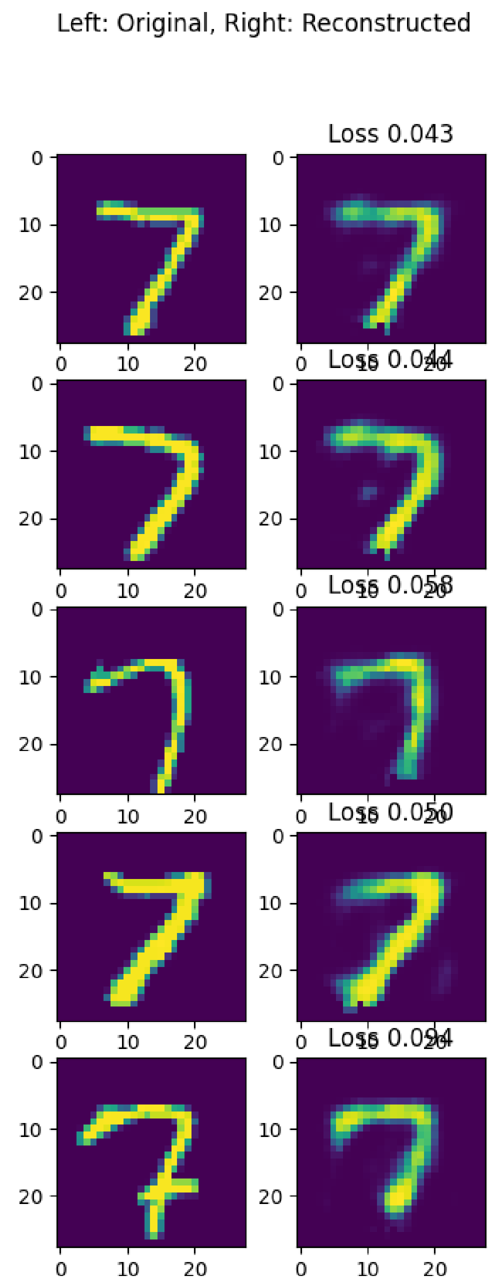

Left: Original, Right: Reconstructed
Loss 0.085
Loss 0.102
Loss 0.094
Loss 0.120
Loss 0.125

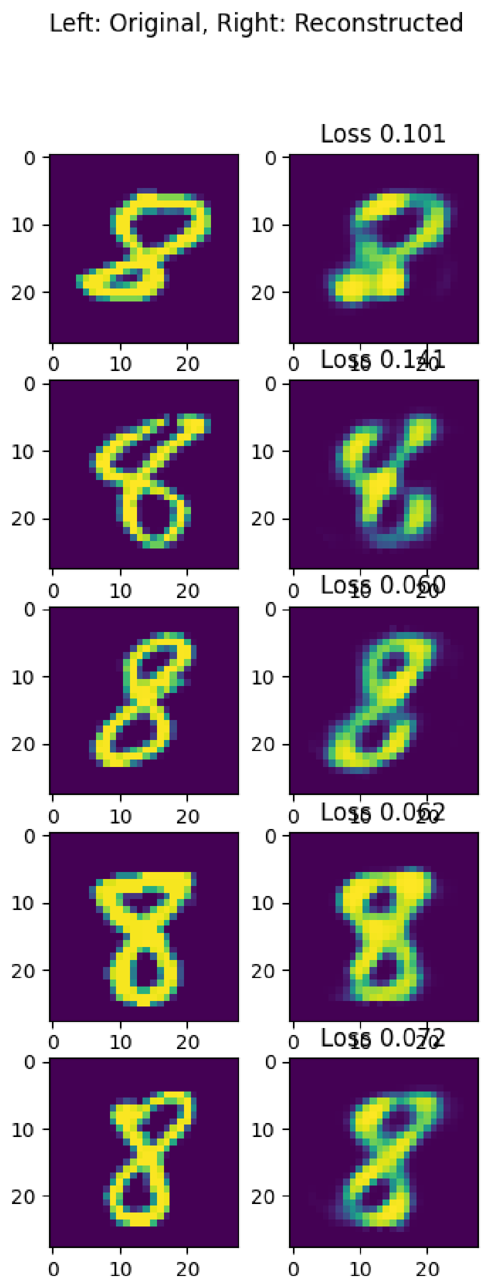

Left: Original, Right: Reconstructed
Loss 0.150
Loss 0.041
Loss 0.048
Loss 0.047
Loss 0.070

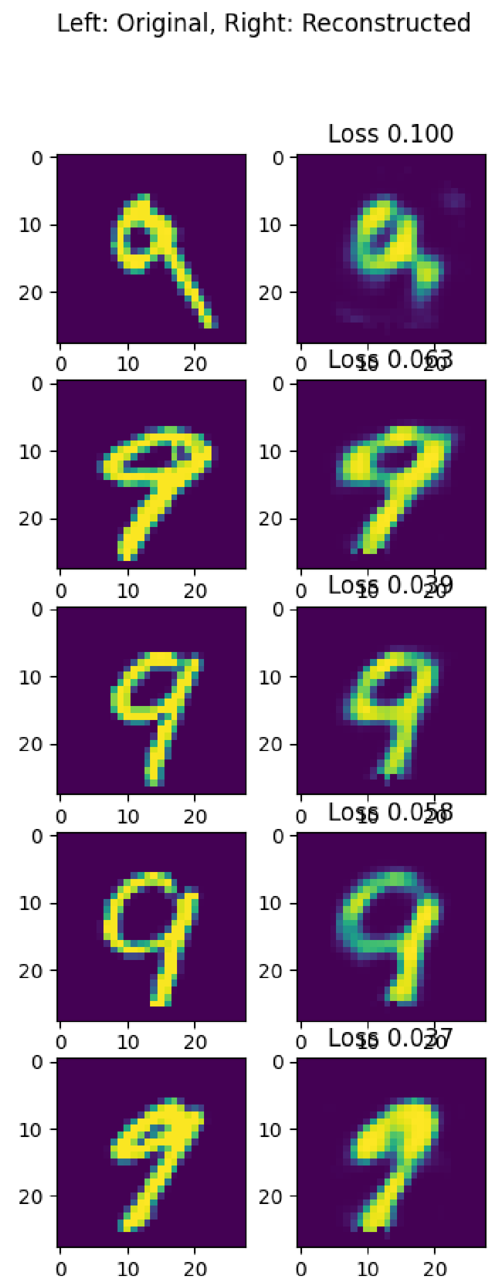

Left: Original, Right: Reconstructed
Loss 0.060
Loss 0.064
Loss 0.052
Loss 0.070
Loss 0.050


Left: Original, Right: Reconstructed
Loss 0.121
Loss 0.110
Loss 0.064
Loss 0.066
Loss 0.055


Left: Original, Right: Reconstructed
Loss 0.113
Loss 0.061
Loss 0.126
Loss 0.044
Loss 0.183


Left: Original, Right: Reconstructed
Loss 0.043
Loss 0.044
Loss 0.058
Loss 0.050
Loss 0.094


Left: Original, Right: Reconstructed
Loss 0.101
Loss 0.141
Loss 0.060
Loss 0.062
Loss 0.072


Left: Original, Right: Reconstructed
Loss 0.100
Loss 0.063
Loss 0.039
Loss 0.058
Loss 0.037

□