4 CNN and AutoEncoders

We use CNNs and auto encoders to perform image classification on MNIST datase

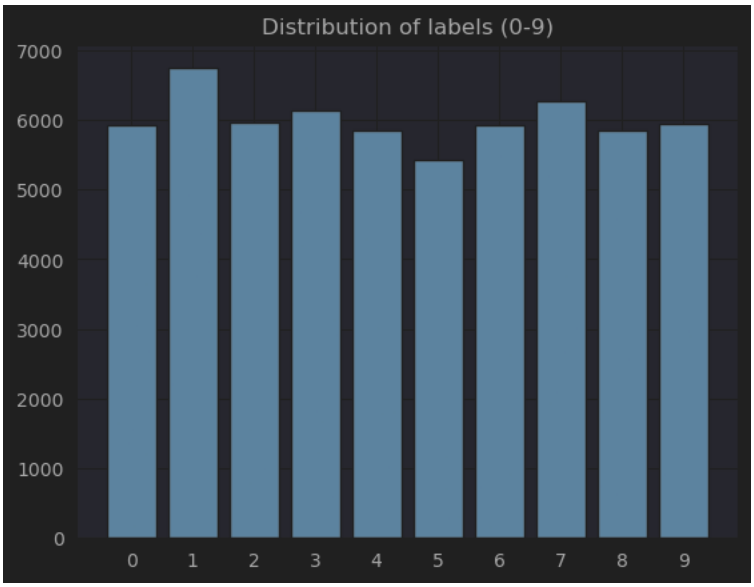
Arnav Negi

Data Exploration

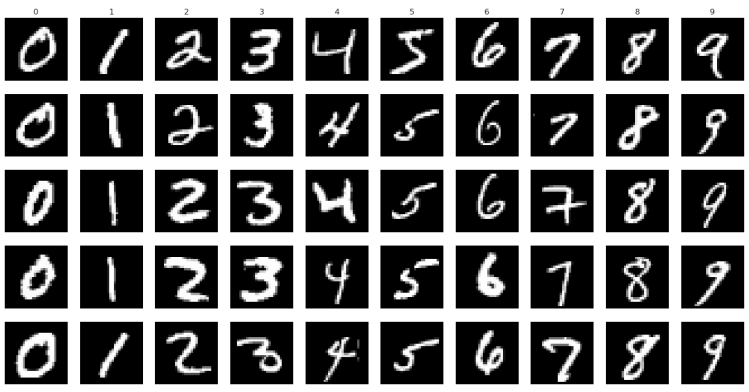
MNIST stands for the "Modified National Institute of Standards and Technology" dataset. It is a collection of handwritten digits (0-9) that have been preprocessed and resized to 28x28 pixels.

The MNIST dataset is often used as a benchmark for testing and evaluating machine learning algorithms, particularly in the context of image classification and digit recognition.

Label distribution:



Samples visualized:



CNNs and feature maps

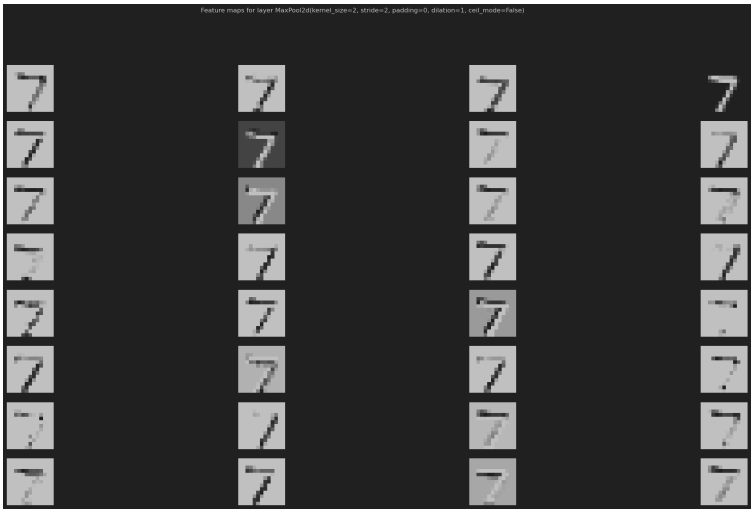
Convolutional Neural Networks (CNNs), often referred to as ConvNets, are a class of deep learning models specifically designed for processing structured grid data, such as images and video.

CNNs are widely used in computer vision tasks, including image classification, object detection, facial recognition, and image generation. They are also applied in fields such as natural language processing and speech recognition.

Feature Maps

Feature maps are two-dimensional grids of numbers generated as the output of convolutional layers in a CNN. Each feature map represents the activation of specific learned filters (also called kernels) applied to the input data. They are helpful in visualizing outputs of CNNs

Feature maps for our CNN model on MNIST data:



1st convolutional layer

2nd convolutional layer

Model Training and Tuning

Our CNN model is made with pytorch and uses Conv2D, MaxPool and Dropout layers for the MNIST task.

We have a CNN model for tuning and a baseline CNN model as provided in assignment.

Baseline CNN performance on val set:

```
Epoch: 1
Val set: Average loss: 0.0465, Accuracy: 9759/10000 (98%)

Training: 3754it [01:46,  3.69it/s, epoch=3, loss=1.48]

Epoch: 2
Val set: Average loss: 0.0464, Accuracy: 9793/10000 (98%)

Training: 5633it [02:33,  6.53it/s, epoch=4, loss=1.48]

Epoch: 3
Val set: Average loss: 0.0464, Accuracy: 9789/10000 (98%)

Training: 7506it [03:27,  4.95it/s, epoch=5, loss=1.49]

Epoch: 4
Val set: Average loss: 0.0464, Accuracy: 9778/10000 (98%)

Training: 9375it [04:21, 35.84it/s, epoch=5, loss=1.48]

Epoch: 5
Val set: Average loss: 0.0463, Accuracy: 9816/10000 (98%)

Finished Training
```

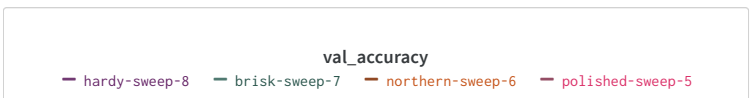
On test set:

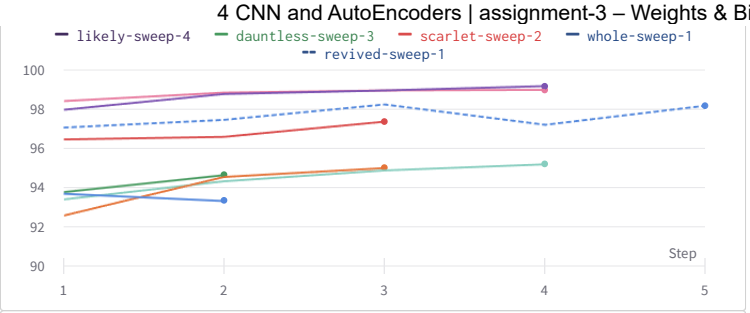
0	0.97	0.99	0.98	980
1	1.00	0.99	0.99	1135
2	0.96	0.99	0.98	1032
3	0.98	0.99	0.98	1010
4	0.99	0.98	0.99	982
5	0.99	0.97	0.98	892
6	1.00	0.96	0.98	958
7	0.97	0.98	0.98	1028
8	1.00	0.96	0.98	974
9	0.97	0.99	0.98	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

Classification report

Now we use the CNN class for hyperparameter tuning. The hyperparameter are as follows:

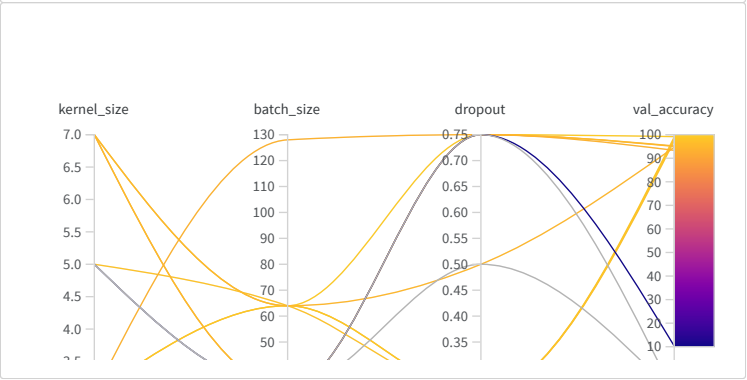
- kernel size
- dropout rate
- learning rate
- epochs
- batch size
- stride




likely-sweep-4
dauntless-sweep-3
scarlet-sweep-2
whole-sweep-1
revived-sweep-1
100
98
96
94
92
90
Step
1
2
3
4
5


f1
hardy-sweep-8
brisk-sweep-7
northern-sweep-6
polished-sweep-5
likely-sweep-4
dauntless-sweep-3
scarlet-sweep-2
whole-sweep-1
revived-sweep-1
1
0.98
0.96
0.94
0.92
0.9
Step
1
2
3
4
5


val_loss
hardy-sweep-8
brisk-sweep-7
northern-sweep-6
polished-sweep-5
likely-sweep-4
dauntless-sweep-3
scarlet-sweep-2
whole-sweep-1
revived-sweep-1
0.06
0.04
0.02
0
Step
1
2
3
4
5


kernel_size
batch_size
dropout
val_accuracy
7.0
6.5
6.0
5.5
5.0
4.5
4.0
130
120
110
100
90
80
70
60
50
0.75
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
100
90
80
70
60
50
40
30
20
10

Run set 11

The best performing model has the following hyperparameters:

- kernel size = 3
- epochs = 4
- dopout rate = 0.25
- learning rate = 0.001
- batch size = 64

Comparison between with dropout and without dropout

The following was the performance of CNNs in both cases:

```
Epoch: 1
Val set: Average loss: 0.0018, Accuracy: 9815/10000 (98%)
Epoch: 2
Val set: Average loss: 0.0012, Accuracy: 9866/10000 (99%)
Epoch: 3
Val set: Average loss: 0.0011, Accuracy: 9880/10000 (99%)
Epoch: 4
Val set: Average loss: 0.0010, Accuracy: 9910/10000 (99%)
Epoch: 5
Val set: Average loss: 0.0011, Accuracy: 9899/10000 (99%)

Finished Training
```

CNN with dropout layer of 0.25

```
Epoch: 1
Val set: Average loss: 0.0014, Accuracy: 9851/10000 (99%)
Epoch: 2
Val set: Average loss: 0.0013, Accuracy: 9870/10000 (99%)
Epoch: 3
Val set: Average loss: 0.0012, Accuracy: 9882/10000 (99%)
Epoch: 4
Val set: Average loss: 0.0013, Accuracy: 9885/10000 (99%)
Epoch: 5
Val set: Average loss: 0.0012, Accuracy: 9886/10000 (99%)

Finished Training
```

CNN with no dropout

Since the complexity of these models is low, dropout's effect is there but is not significant.

Model Evaluation

The model is evaluated on the test set with the following hyperparams:

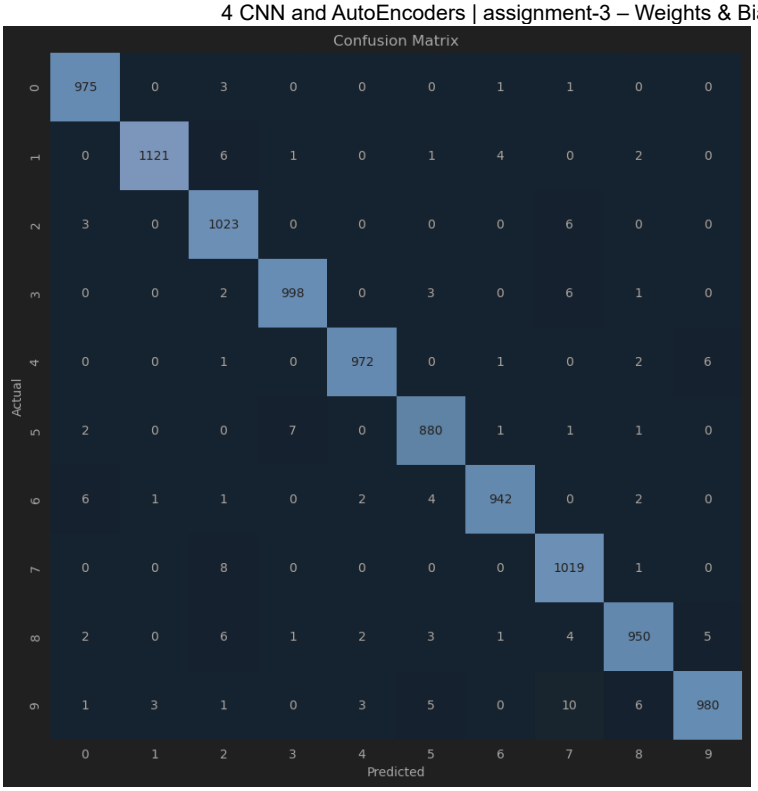
- kernel size = 3
- stride = 1
- dropout = 0.25

```
Accuracy: 98.60
Class 0: 99.49
Class 1: 98.77
Class 2: 99.13
Class 3: 98.81
Class 4: 98.98
Class 5: 98.65
Class 6: 98.33
Class 7: 99.12
Class 8: 97.54
Class 9: 97.13
```

Class wise accuracies

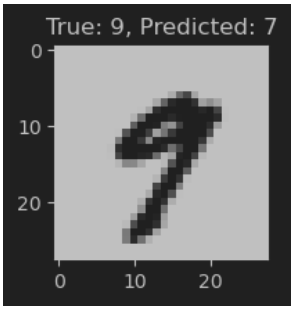
```
Classification report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99       980
           1       1.00      0.99      0.99      1135
           2       0.97      0.99      0.98      1032
           3       0.99      0.99      0.99      1010
           4       0.99      0.99      0.99       982
           5       0.98      0.99      0.98       892
           6       0.99      0.98      0.99       958
           7       0.97      0.99      0.98      1028
           8       0.98      0.98      0.98       974
           9       0.99      0.97      0.98      1009

    accuracy                           0.99     10000
   macro avg       0.99      0.99      0.99     10000
weighted avg       0.99      0.99      0.99     10000
```

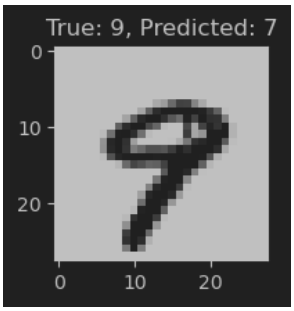
Classification report (multiclass)



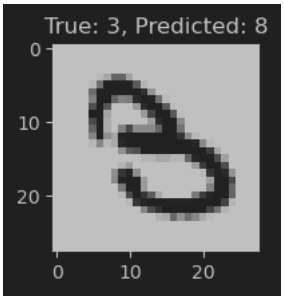
Confusion matrix

Misclassification analysis





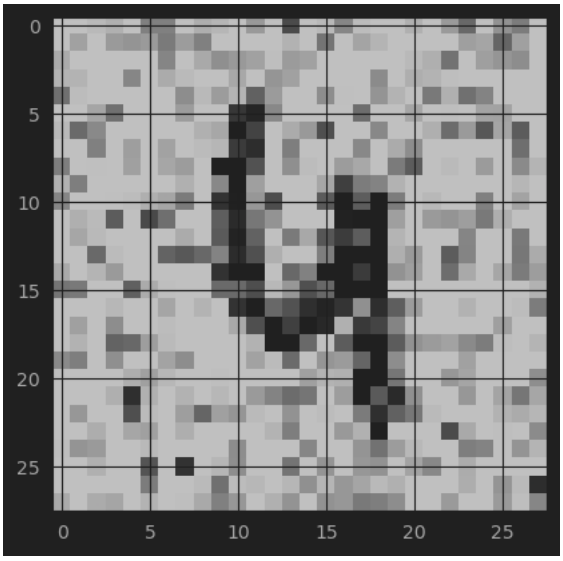
Two of the examples are slanted 9s which the model mistook for similar looking number 7.



The third example is a 3 which is rounded and hence looks like an 8.

Noisy MNIST

We use the noisy mnist dataset with AWGN (Additive White Gaussian Noise) to train the CNN.



Example of a noisy image

Performance of previous model trained on noisy data:

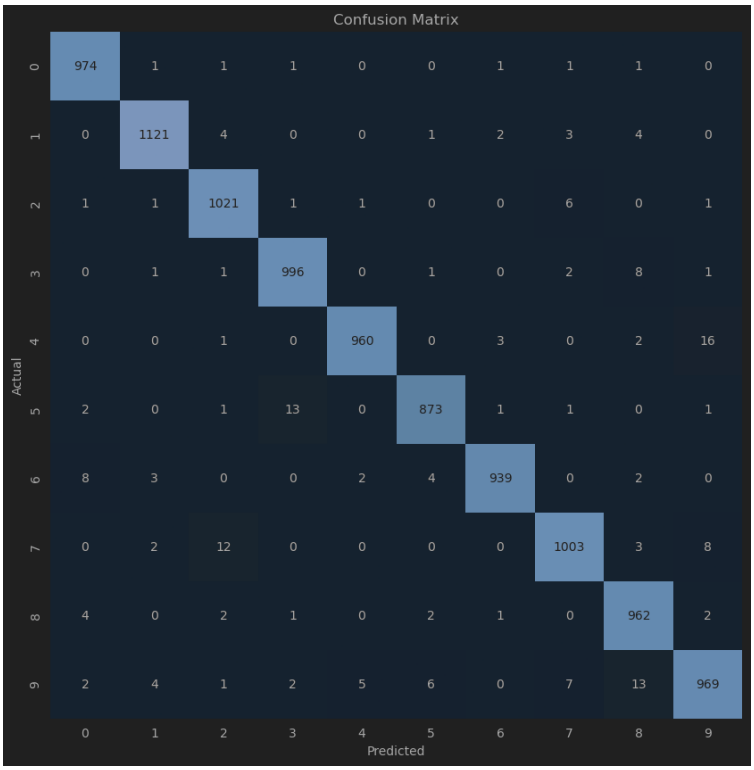
```
Training: 1879it [01:26,  2.54it/s, epoch=2, loss=0.12]

Epoch: 1
Val set: Average loss: 0.0057, Accuracy: 9484/10000 (95%)

Training: 3752it [03:09,  1.26it/s, epoch=3, loss=0.0228]

Epoch: 2
Val set: Average loss: 0.0028, Accuracy: 9710/10000 (97%)

Training: 5628it [04:49,  1.90it/s, epoch=4, loss=0.0874]

Epoch: 3
Val set: Average loss: 0.0034, Accuracy: 9691/10000 (97%)

Training: 7503it [06:26,  1.88it/s, epoch=5, loss=0.0239]

Epoch: 4
Val set: Average loss: 0.0031, Accuracy: 9765/10000 (98%)

Training: 9375it [07:55, 19.71it/s, epoch=5, loss=0.0491]

Epoch: 5
Val set: Average loss: 0.0035, Accuracy: 9757/10000 (98%)

Finished Training
```

Training and validation results for noisy data

It gives 97.57% accuracy at the end on the validation set

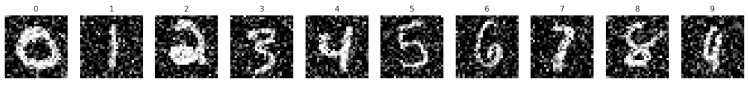
The test set results are

0	0.98	0.99	0.99	980
1	0.99	0.99	0.99	1135
2	0.98	0.99	0.98	1032
3	0.98	0.99	0.98	1010
4	0.99	0.98	0.98	982
5	0.98	0.98	0.98	892
6	0.99	0.98	0.99	958
7	0.98	0.98	0.98	1028
8	0.97	0.99	0.98	974
9	0.97	0.96	0.97	1009
accuracy			0.98	10000
macro avg	0.98	0.98	0.98	10000
weighted avg	0.98	0.98	0.98	10000

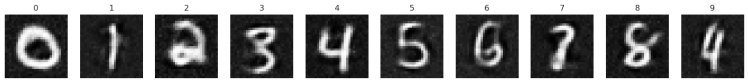


Denoising with auto encoders

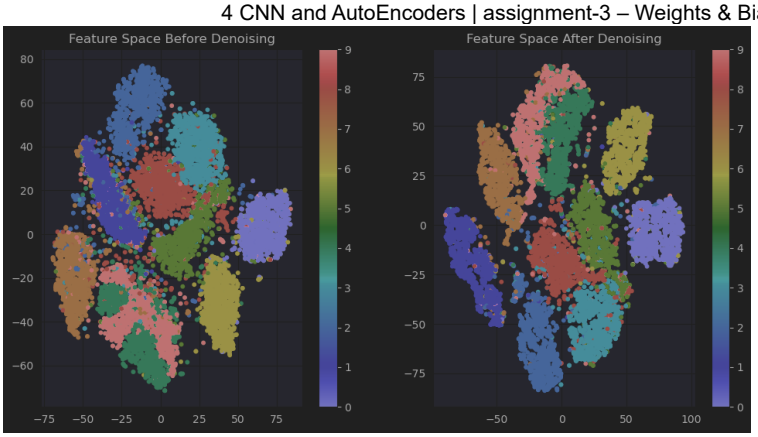
Using autoencoders we can denoise the images



Original, noisy



Denoised



t-SNE analysis of the images

This gives improvements in accuracy

0	0.99	1.00	1.00	980
1	1.00	1.00	1.00	1135
2	1.00	1.00	1.00	1032
3	1.00	0.99	0.99	1010
4	0.99	0.99	0.99	982
5	0.99	0.99	0.99	892
6	1.00	1.00	1.00	958
7	0.99	1.00	0.99	1028
8	0.99	0.99	0.99	974
9	0.99	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Accuracy jumps to 99% and in some classes 100% after denoising.

Comparison

CNNs on denoised samples take longer to train, however give improved accuracies.