Intro to Graphic Models & Deep Learning
Final Project

Matan Goldfarb, ID: 314623174

Talya Yermiahu, ID: 207594193

Alon Meirovich, ID: 330181470

August 2024

Abstract

In the scope of this course we learned the basic techniques and underlying theory that is the founding stone of deep learning models.
This final project is the Open Set Recognition (OSR) based on the MNIST dataset and evaluated over some open set that holds MNIST and more pictures.
In this project summary, we'll guide you through our learning process: how and what we looked for in our theoretical research, what was the evolution of our model over time and trail & error, and what conclusions we derive from those experiences. We hope you enjoy it.

Contents

1 Introduction **3**

2 Project strategy **3**
2.1 How should we start? . . . 3
2.2 General outline & planning . . . 3
2.3 Theoretical research . . . 4
2.3.1 Convolutional layers for image processing . . . 4
2.3.2 Fully connected layers in our model . . . 4
2.3.3 Autoencoders (AEs) . . . 4
2.3.4 Key tools for image processing . . . 5

3 The chosen model **5**
3.1 Model testing . . . 5
3.1.1 FC model . . . 5
3.1.2 Sole Convolutional model . . . 6
3.1.3 Integrated model . . . 7
3.2 Accuracy oriented programming . . . 8
3.2.1 Architecture fine tuning . . . 8
3.2.2 Hyper-Parameters implementation & fine tuning . . . 10

4 Final results **10**

5 Limitations **11**
5.1 Complex datasets testing . . . 11

6 Conclusions **12**

7 Bibliography **13**

1 Introduction

In the realm of Deep learning, Open Set Recognition stands as a central paradigm addressing the limitations of traditional closed-set classification systems. In conventional classification tasks, models are trained on a fixed dataset and evaluated on test data drawn from the same distribution. However, real-world applications often encounter novel or previously unseen instances that are not part of the original training set. This is where OSR becomes a practical solution.

OSR involves training a deep learning model on a given dataset, such as MNIST, and then evaluating its performance on an open set. An open set contains not only data from the training set but also additional, unfamiliar data. The core objective of OSR is to develop models capable of two primary tasks: accurately classifying known instances and effectively identifying and labeling unfamiliar instances as "unknown".

This dual capability is crucial for enhancing the robustness and applicability of AI systems in dynamic and unpredictable environments. By distinguishing between known and unknown data, OSR models can improve decision-making processes and provide more reliable performance in real-world scenarios.

2 Project strategy

2.1 How should we start?

The first intuitive reaction we had after reading the project's guidelines was that the first and more important decision to make is what model to choose.

Choosing the right model for OSR is paramount due to its direct impact on the system's ability to accurately identify the data provided in test as well as precise performance when using it in real world applications.

The right model ensures a balanced performance, minimizing false positives (misclassifying unknowns as known) and false negatives (misclassifying knowns as unknown), thereby enhancing its reliability.

Additionally, the model's capacity to adapt to new and evolving data distributions can significantly influence the overall success of an OSR system in practical deployments, making the choice of model a critical decision in the development of resilient solutions.

2.2 General outline & planning

After defining the main objective of our project, we realized that the most effective way to achieve it is by outlining and following detailed steps.

To address these considerations, we developed a comprehensive action plan that included the following steps:

1. Conducting a theoretical review, in order to gather relevant information on OSR.
2. Understanding the general structure of our model, along with auxiliary functions that could enhance it.
3. Implementing the model and its auxiliary functions through coding.
4. Training the model on the MNIST dataset.
5. Evaluating the model's performance and analyzing reasons for its success or shortcomings, fixing relevant issues.
6. Testing the model on an open set and analyzing its performance.
7. Fine-tuning hyperparameters with a focus on open set performance.
8. Analyzing overall accuracy and performance metrics.
9. Ensuring compliance with all project requirements.
10. Conducting a final revision and correction of the code, summarizing findings, and submitting the project.

Throughout the development of our project, we encountered various challenges such as low accuracy in model predictions, high training loss, good accuracy on the MNIST evaluation but poor accuracy on the open set; attempts to improve the model by adding complexity which resulted in worse performance and more. Issues addressed by thoroughly reviewing and researching deep learning theory; applying our newfound knowledge; recognizing that not all deep learning models are suited for image processing; understanding that increased complexity does not necessarily lead to higher accuracy; and other insights.

2.3 Theoretical research

The main source of information for this project was the pdfs, slides and recordings of lectures and practical sessions of *Intro to Graphical Models & Deep Learning* at BGU.

Also, we looked for information in research papers, Wikipedia, used ChatGPT and other LLM for counseling and aid, researched for information in StackExchange, GeeksForGeeks and other forums, as well as at the Pytorch documentations.

2.3.1 Convolutional layers for image processing

Over our research we found that the most wide-used and best parameters-cost to accuracy ratio are the Convolutional Neural Networks (CNNs).

From the initial accurate models for OSR such as LeNet [1] or other popular ones like AlexNet [2], through examples of Deep Learning tutorials of forums such as GeeksforGeeks or libraries documentation like Keras, ususally the used architecture is CNNs.

This architecture is very efficient at learning hierarchical feature representations from raw data, they capture spatial hierarchies through their layers, what is an important technique when recognizing images.

The convolutional layers in CNNs combine local connectivity and weight sharing, making them versatile when variations occur in the input, such as translations and distortions. This property helps in recognizing new patterns that may belong to unknown classes.

From a personal standpoint, we understood that a key factor for improved efficiency and accuracy is the use of a **kernel**, particularly a 2D kernel [3]. This approach enhances the model's ability to comprehend neighboring pixels around each one it analyzes, thereby improving its pattern recognition capabilities.

2.3.2 Fully connected layers in our model

The first architectural model introduced in this course was the Multilayer Perceptron (MLP), which consists of fully connected (FC) layers.

The ability of MLPs to generalize data and recognize patterns more broadly than convolutional layers led us to hypothesize that our model's performance could improve by incorporating convolutional learning on a smaller scale, and complementing them with dense FC layers in a more general sense.

This approach enabled the model to learn patterns from neighboring pixels within a specific area, enhancing its ability to identify them. Subsequently, using FC layers provided a more comprehensive understanding of the image, integrating these patterns in both linear and non-linear manners. This combined approach facilitated more accurate and efficient identification of the MNIST dataset.

2.3.3 Autoencoders (AEs)

As discussed in lectures and practical sessions, the use of Autoencoders (AEs) was considered for our model architecture.

However, our research indicated that AEs are primarily utilized for image denoising rather than image recognition. The majority of relevant literature and examples we reviewed did not employ AEs for image recognition tasks. Additionally, the inherent data loss associated with AEs was a concern.

Consequently, we decided against incorporating AEs into our model.

Had we not achieved optimal results with other techniques, we might have considered implementing an Autoencoder model and evaluating its performance.

2.3.4 Key tools for image processing

1. **Optimizer:**
   While developing of our model, it was clear that the use of a optimizer is a useful tool, given that it updates the weights according to the gradient found. We found that Adam is a good choice for our task. Adam accelerates the gradient descent algorithm, in order to reach the global minimum faster. *We found Adam to be robust and well-suited to a wide range of non-convex optimization problems in the field machine learning* [4].

2. **Criterion:**
   The criterion on which a model analyses data is an integral part of Deep Learning. We found that Cross Entropy Loss provides a way to measure how well the predicted probabilities align with the true labels. Its ability to handle multiple classes, numerical stability, and ease of integration with neural networks make it practical and effective for training our model.

3. **Threshold:**
   For our model to make accurate predictions on unknown data, and tag it at the correspondent class, threshold is an important component. We understood that if we can achieve a high accuracy model, it can be "more confident" when tagging familiar data, and thus we can interpret a less confident prediction as unknown data. We saw too, that as our model gets more accuracy, we can set a higher threshold in order to get a higher unknown identification accuracy.

4. **Soft-max:**
   The SoftMax function normalizes the model's outputs into a probability distribution [5], allowing us to set a higher threshold for more accurate predictions on new data. Without SoftMax, the model's confidence in its predictions might be too low, potentially leading to false negatives. This low confidence, relative to a specific run, becomes less meaningful after the data undergoes multiple transformations during analysis.

5. **Pooling:**
   Pooling layers abstract and compress the features extracted by convolutional layers. Max-pooling can lead to faster convergence, select superior invariant features, and improve generalization[6], while Avg-Pooling provides a more generalized feature summary. This abstraction helps the model focus on the most important features of the given data.

3 The chosen model

We shall explore in this section how the model's development process took part, and what insights and difficulties we encountered along the way.

3.1 Model testing

3.1.1 FC model

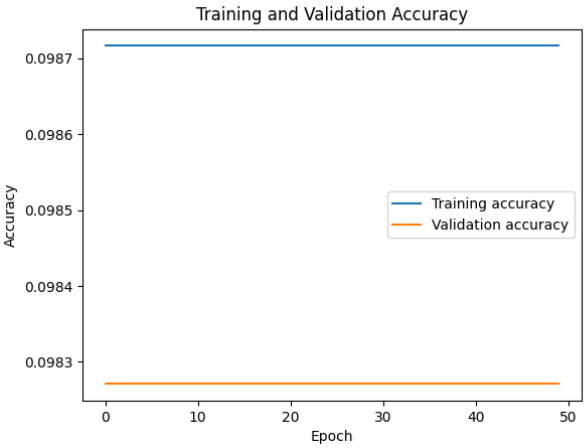
A starting thought was that a model using this architecture may be a good idea. And so we implemented a basic model, similar to the one used in MNIST recognition of HW1, composed by the first layer of one input channel and 128 output channels, followed by 3 layers of 128x128 channels, and finally with a layer of 128x1 channels, with the output as the prediction the model made. Each layer with a ReLU activation function, given its advantages over other activation functions (such as simplicity, convergence, gradient propagation and more).

We saw that that's not the case.

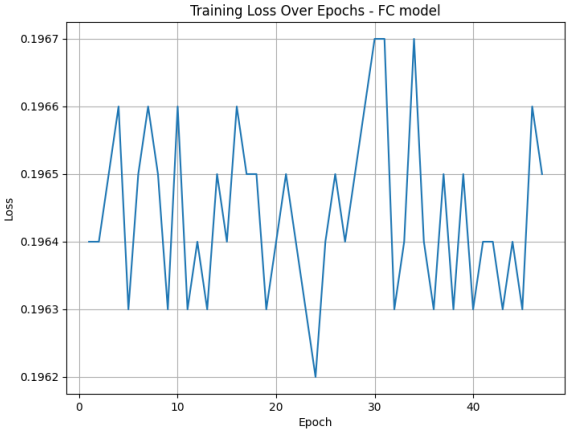
Our first model performed poorly compared to our expectations, with less than 10% accuracy, 0.2 training loss and more or less 45 minutes of train run time, for 50 epochs:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                 [-1, 128]         100,480
              ReLU-2                 [-1, 128]               0
            Linear-3                 [-1, 128]          16,512
              ReLU-4                 [-1, 128]               0
            Linear-5                 [-1, 128]          16,512
              ReLU-6                 [-1, 128]               0
            Linear-7                 [-1, 128]          16,512
              ReLU-8                 [-1, 128]               0
            Linear-9                   [-1, 1]             129
================================================================
Total params: 150,145
Trainable params: 150,145
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.01
Params size (MB): 0.57
Estimated Total Size (MB): 0.58
```

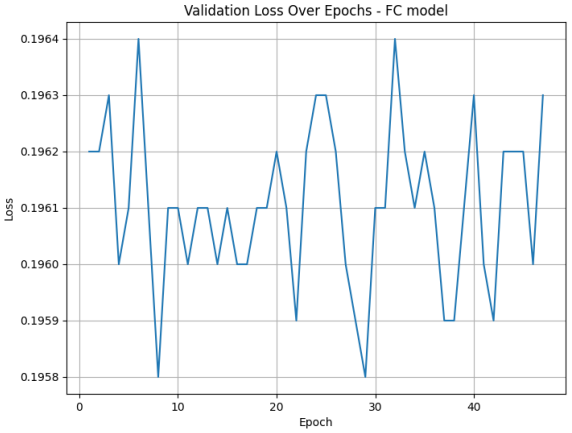
(a) Model summary, with more than 150,000 parameters.



(b) Model accuracy.



(c) Train loss.



(d) Validation loss.

Figure 1: Data and performance of the FC model.

Given these results, we got the impression that our model cannot be linear, given the poor accuracy and loss variation it achieved. As we can see in the graphs, the FC model didn't seem to learn from the training.

And so we thought of the possibility that our model's core learning has to be convolutional.

3.1.2 Sole Convolutional model

Based on our previous learnings, we experimented with a model consisting solely of three convolutional layers with Avg-Pooling, what enabled our model to achieve improved recognition of features in the handwritten data:

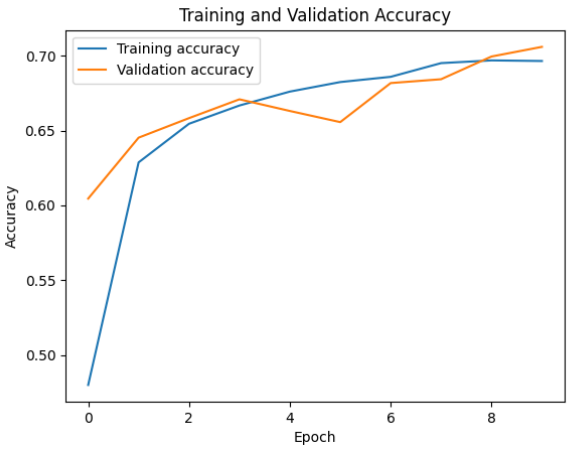
```python
# Model Class
class CNN(nn.Module):
    def __init__(self, channels, classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(channels,  out_channels: 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d( in_channels: 32,  out_channels: 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d( in_channels: 64, classes, kernel_size=3, padding=1)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = F.relu(self.conv2(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = F.relu(self.conv3(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = torch.flatten(x, 1)
        return x
```

Figure 2: Our sole-convolutional model architecture.

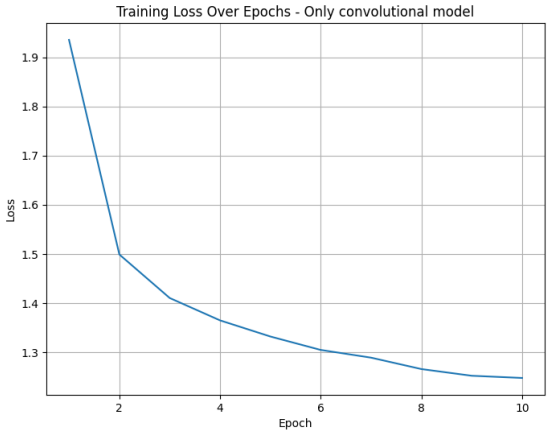
its performance was much better:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 28, 28]             320
            Conv2d-2          [-1, 64, 14, 14]          18,496
            Conv2d-3           [-1, 10, 7, 7]           5,770
================================================================
Total params: 24,586
Trainable params: 24,586
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.29
Params size (MB): 0.09
Estimated Total Size (MB): 0.39
----------------------------------------------------------------
```

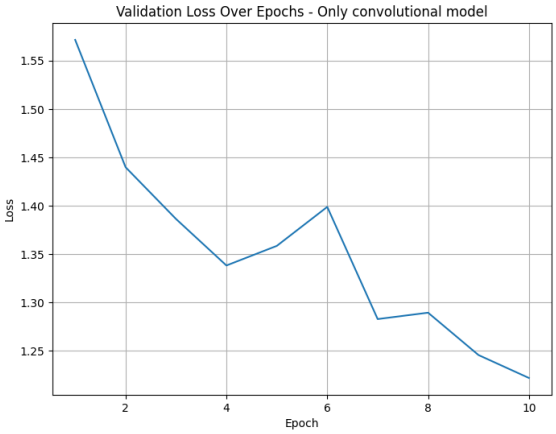
(a) Model summary, with 25,000 parameters.



(b) Model's accuracy. We see a big improvement.



(c) Train loss.



(d) Validation loss.

Figure 3: Data and performance of the only-convolutional model.

We saw that this model reached a better accuracy (around 70%) with less runs and almost half run time, than the FC model. However, it failed when testing:

```
Test Loss: 2.2520, Test Accuracy: 0.3859, Unknown Ratio: 0.5099, Time elapsed in test: 00:00:37
```

Figure 4: Our first test, with 38% accuracy.

We needed to understand better the topic for defining our model's final architecture.

In any case, an important conclusion from this model is that the core learning has to be convolutional, given that it learns more rapidly (better accuracy with less epochs) using less resources.

3.1.3 Integrated model

After conducting research and seeking advice, we implemented a generalized model that combines Fully Connected with Convolutional layers. The Convolutional layers lightened the model while learning more efficiently small-scale spatial recognition, utilizing 2D kernels and pooling techniques to effectively capture local patterns and features in the data. Meanwhile, the FC layers are employed to perform more general learning by integrating and interpreting the features extracted by the Convolutional layers. This hybrid approach allows the model to leverage the strengths of both layer types, enhancing its ability to recognize intricate details and form a comprehensive understanding of the data.

3.2 Accuracy oriented programming

Once we got understood the general architecture of our model, we keep experimenting with different configurations to achieve more accuracy in less train run time.

3.2.1 Architecture fine tuning

And so, initially we sought for a complex model, thinking that complexity may imply high prediction accuracy.

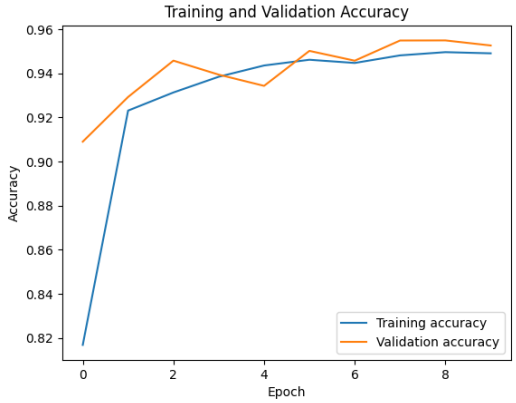
```python
class CNN(nn.Module):
    def __init__(self, channels, classes):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(channels,  out_channels: 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d( in_channels: 32,  out_channels: 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d( in_channels: 64,  out_channels: 128, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(128 * 3 * 3,  out_features: 128)
        self.fc2 = nn.Linear( in_features: 128,  out_features: 64)
        self.fc3 = nn.Linear( in_features: 64, classes)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = F.relu(self.conv2(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = F.relu(self.conv3(x))
        x = F.avg_pool2d(x,  kernel_size: 2)
        x = torch.flatten(x, 1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Figure 5: Our first integrated model architecture.

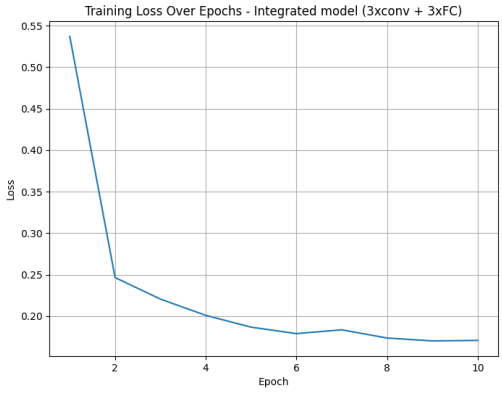
And we found the best results until now:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 28, 28]             320
            Conv2d-2          [-1, 64, 14, 14]          18,496
            Conv2d-3           [-1, 128, 7, 7]          73,856
            Linear-4                 [-1, 128]         147,584
            Linear-5                  [-1, 64]           8,256
            Linear-6                  [-1, 10]             650
================================================================
Total params: 249,162
Trainable params: 249,162
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.34
Params size (MB): 0.95
Estimated Total Size (MB): 1.29
----------------------------------------------------------------
```

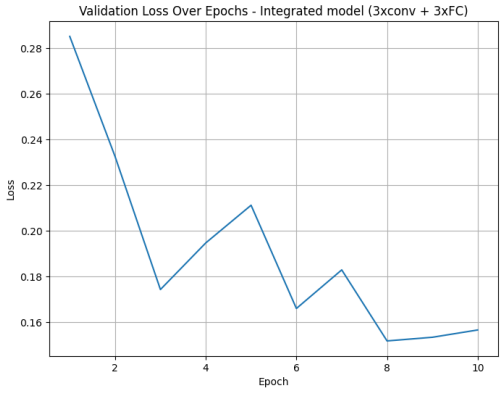
(a) Model summary, with 250,000 parameters.



(b) Model's accuracy. We see a good improvement.



(c) Train loss.



(d) Validation loss.

Figure 6: Data and performance of the integrated model of 3 convolutional and 3 fully connected layers.

```
Test Loss: 1.6341, Test Accuracy: 0.5486, Unknown Ratio: 0.5031, Time elapsed in test: 00:01:12
```

Figure 7: First integrated model test results.

With around 95% train accuracy, and 55% test accuracy, we felt we were on the right track but not ready to define the architecture.

We felt that the main problem of our model was the premise we started with, its complexity.

We thought that this model was too complex: it performed overfitting on the data and took much run time to get a *only good* accuracy. We then asked ourselves if we can compose a simpler model that performs equally good or even better...

```python
# Evaluate the OSR model
# Loading the saved OSR model
th = 0.7
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
osr_model_loaded = OSRCNN(th).to(device)
osr_model_loaded.load_state_dict(torch.load('osr_3conv_model_epoch_10.pth', map_location=device))
osr_model_loaded.eval()
acc_mnist, acc_ood, acc_total = eval_model(osr_model_loaded, combined_test_loader, device)
print(f'MNIST Accuracy: {acc_mnist*100:.2f}%')
print(f'OOD Accuracy: {acc_ood*100:.2f}%')
print(f'Total Accuracy: {acc_total*100:.2f}%')

# So after ±50 runs this model approaches a 96% of accuracy, with another model we may get more!!
# There won't be no time limit for training, just for test
# Ask Ron how many pics will be on test
```

```
MNIST Accuracy: 98.76%
OOD Accuracy: 45.62%
Total Accuracy: 72.19%
```

Figure 8: New integrated model architecture and results

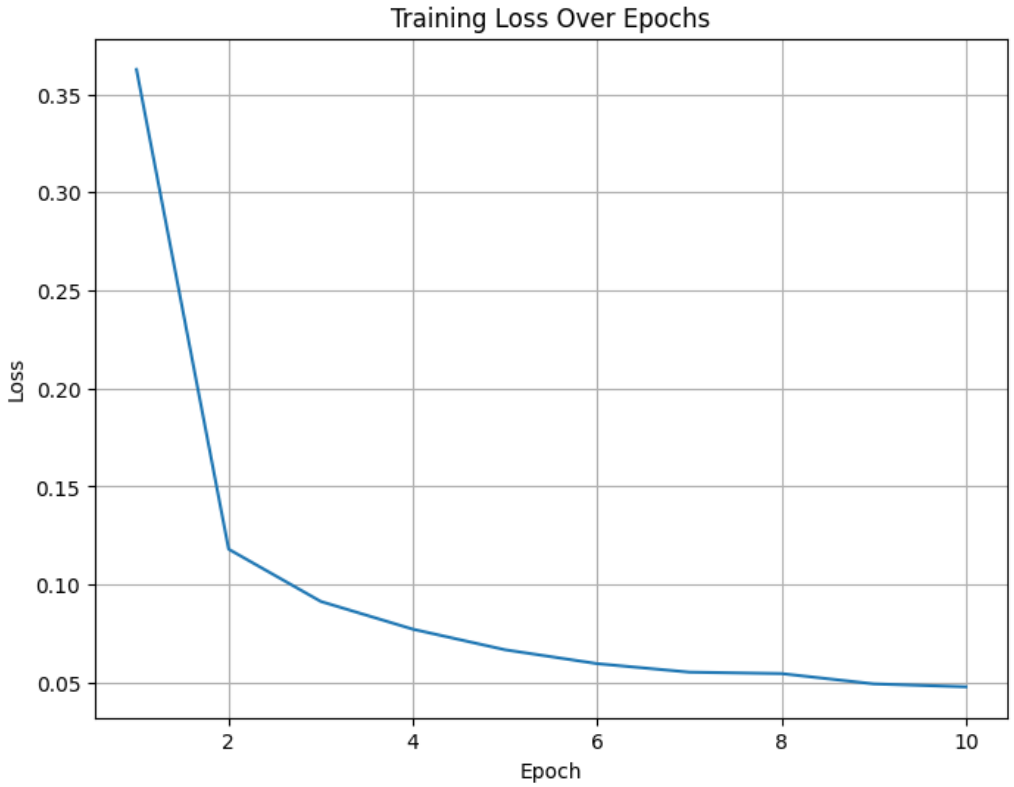


Figure 9: Its train loss.

We saw in the model of 2 convolutional layers and 2 dense layers that the MNIST accuracy improved as well as the total test accuracy. Also, we saw a better loss convergence, in a smoother way. So we opted for a

smaller yet much more precise mode, hoping that with hyper-parameters fine tuning we could get an excellent overall performance.

3.2.2 Hyper-Parameters implementation & fine tuning

As said in the theoretical research section, we found a series of techniques and features of convolutional and FC layers that helped us design a better model overall.

- We decided to implement Max-Pooling in the convolutional layers. Because it helps the model define the limits of the image more precisely, thus sending clearer pattern recognition to the following dense layers.
- We implemented Adam optimizer in order to converge more effectively to the global minimum of the loss function, as explained in the theoretical research section.
- We used the SoftMax function so the resulting output tensor is normalized into a probability distribution, and so we can perform decisions over specific data (and define hyper-parameters) in a general, overall way.
- The learning rate of the model was started at 0.001. Given that our model performed quickly overall, we preferred to perform a slower but more qualitative learning, having the optimizer as a "backup", allowed to change the learning rate accordingly with the model performance. As we saw too in [4], Adam is highly effective with small learning rates.

4 Final results

After this whole process, code corrections, debates between the group and more, the final version of our model is the following:

```
[ ]  # Model Class
     class OSRCNN(nn.Module):
         def __init__(self, th):
             super(OSRCNN, self).__init__()
             self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
             self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
             self.fc1 = nn.Linear(64 * 7 * 7, 128)
             self.fc2 = nn.Linear(128, 11)  # 10 classes + 1 unknown
             self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
             self.relu = nn.ReLU()
             self.softmax = nn.Softmax(dim=1)
             self.th = th
             self.valMode = False

         def forward(self, x):
             x = self.pool(self.relu(self.conv1(x)))
             x = self.pool(self.relu(self.conv2(x)))
             x = x.view(-1, 64 * 7 * 7)
             x = self.relu(self.fc1(x))
             x = self.fc2(x)
             if not self.training and not self.valMode:
                 with torch.no_grad():
                     x = self.softmax(x)
                     probas, y_pred = torch.max(x, 1)
                     y_pred[probas < self.th] = 10
                     return y_pred
             return x

         def set_validation(self, val_mode):
             self.valMode = val_mode
```

Figure 10: The final architecture our model took.

And having the final version of our model, we experimented with different hyper-parameters, getting the following results:

Threshold	Number of epochs	MNIST accuracy	OOD accuracy	Total accuracy
0.7	10	∼99%	∼46 %	∼72%
0.75	10	∼98%	∼59%	∼75%
0.8	10	∼98%	∼64%	∼78%
0.85	10	∼98%	∼70%	∼81%
0.9	10	∼97%	∼78%	∼86%
0.95	10	∼97%	∼84%	∼91%
0.95	20	∼97%	∼88%	∼91%
0.95	30	∼97%	∼91%	∼93%
0.99	30	∼95%	∼96%	∼96%

After getting better results, it became evident the importance on training and high accuracy of our model on the MNIST dataset for improving the test accuracy on the open set [7].

As we achieved a high accuracy when training and testing on the MNIST dataset, we understood that our model predicts correctly the vast majority of time. This enabled us to set a threshold of 0.95, thus understanding that if the model is not sure about its prediction, then the most plausible scenario is that it encountered an unknown data. Setting a really high threshold allowed our model to achieve more than 90% of accuracy on the OSR test. Following the same logic, we tried too a threshold of 0.99 and got the best results of the whole process.

And so, the chosen hyper-parameters are:

⇒ Threshold = 0.99

⇒ Learning rate = 0.001

⇒ Number of epochs at train = 30

⇒ Optimizer = Adam

⇒ Criterion = CrossEntropyLoss

That performed excellently!

5 Limitations

Even though we are satisfied with our model performance, it is overall a basic and simple model. It is clear that it is able to perform in the realm of image processing only, but moreover, it is limited to 28x28 pixel grayscale images. Testing our model over color images of different things that are not digits may result in poor performance, as our model is not prepared for diversified datasets.

5.1 Complex datasets testing

In the same way, we wanted to test our model on datasets that are similar to MNIST, such as Cifar, Fashion MNIST, KMNIST (handwritten korean alphabet letters), and EMNIST (handwritten letters).

And even thought we expected our model to perform poorly, it worked surprisingly fine:

```
[12] import torchvision
     from torchvision import transforms

     # Define transformations
     transform = transforms.Compose([
         transforms.Grayscale(num_output_channels=1),
         transforms.Resize((28, 28)),
         transforms.ToTensor(),
         transforms.Normalize((0.5,), (0.5,))
     ])

     # Load KMNIST dataset
     emnist_dataset = torchvision.datasets.EMNIST(root='./data', split='byclass', train=False, download=True, transform=transform)
     combined_test_loader = DataLoader(CombinedDataset(mnist_test, emnist_dataset), batch_size=batch_size, shuffle=True)

     # Evaluate the OSR model
     # Loading the saved OSR model
     th = 0.99
     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     osr_model_loaded = OSRCNN(th).to(device)
     osr_model_loaded.load_state_dict(torch.load('osr_model_epoch_30.pth', map_location=device))
     osr_model_loaded.eval()
     acc_mnist, acc_ood, acc_total = eval_model(osr_model_loaded, combined_test_loader, device)
     print(f'MNIST Accuracy: {acc_mnist*100:.2f}%')
     print(f'OOD Accuracy: {acc_ood*100:.2f}%')
     print(f'Total Accuracy: {acc_total*100:.2f}%')

     # So after ±50 runs this model approaches a 96% of accuracy, with another model we may get more!!
     # There won't be no time limit for training, just for test
     # Ask Ron how many pics will be on test

     MNIST Accuracy: 94.96%
     OOD Accuracy: 70.97%
     Total Accuracy: 72.87%
```

Figure 11: Our model's performance on the EMNIST dataset.

```
     import torchvision
     from torchvision import transforms

     # Define transformations
     transform = transforms.Compose([
         transforms.Grayscale(num_output_channels=1),
         transforms.Resize((28, 28)),
         transforms.ToTensor(),
         transforms.Normalize((0.5,), (0.5,))
     ])

     # Load KMNIST dataset
     kmnist_dataset = torchvision.datasets.KMNIST(root='./data', train=False, download=True, transform=transform)
     combined_test_loader = DataLoader(CombinedDataset(mnist_test, kmnist_dataset), batch_size=batch_size, shuffle=True)

     # Evaluate the OSR model
     # Loading the saved OSR model
     th = 0.99
     device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
     osr_model_loaded = OSRCNN(th).to(device)
     osr_model_loaded.load_state_dict(torch.load('osr_model_epoch_30.pth', map_location=device))
     osr_model_loaded.eval()
     acc_mnist, acc_ood, acc_total = eval_model(osr_model_loaded, combined_test_loader, device)
     print(f'MNIST Accuracy: {acc_mnist*100:.2f}%')
     print(f'OOD Accuracy: {acc_ood*100:.2f}%')
     print(f'Total Accuracy: {acc_total*100:.2f}%')

     # So after ±50 runs this model approaches a 96% of accuracy, with another model we may get more!!
     # There won't be no time limit for training, just for test
     # Ask Ron how many pics will be on test

     MNIST Accuracy: 94.96%
     OOD Accuracy: 83.24%
     Total Accuracy: 89.10%
```

Figure 12: Our model's performance on the KMNIST dataset.

Having 90% total accuracy on KMNIST, 70% on EMNIST, we can infere that with more training and fine tuning, our model may be able to distinguish better between the datasets.

6 Conclusions

To conclude, our open set model demonstrated good performance and achieved commendable accuracy in its tests. However, it is important to note that this model remains relatively small and limited in scope. While it has shown effectiveness, its performance tends to be only adequate when applied to more complex datasets. This indicates that there is room for further refinement and enhancement to improve its robustness and adaptability in handling more diverse and challenging data scenarios.

7 Bibliography

All the following are hyperlinks to the internet pages that helped us in our research and understanding of the different topics:

1. Famous DL models: CNN LeNet; CNN AlexNet; MLP.
2. CNN examples: CNN explainer; Keras CNN autoencoder example; Kaggle CNN tutorial; GeeksforGeeks intro to CNNs; Medium CNN example.
3. Datasets:
   - Pytorch docs. on Datasets.
   - Training, validation and test Datasets from Wikipeida.
   - Pytorch docs. of Fashion MNIST and Wikipedia page.
   - Pytorch docs. of Cifar and Wikipedia page.
   - Pytorch forum on Concdataset() method.
4. Optimization article of GeeksforGeeks; and Adam's pytorch docs.
5. Pytorch docs. on CrossEntropyLoss
6. CS231n: Deep Learning for Computer Vision course of Stanford University, and their article in CNNs.
7. Activation functions from GeeksforGeeks.
8. Pytorch docs. on AdaptiveAvgPool2d and MaxPool2d.
9. Wikipedia's article on Kernel.
10. SciKit docs. on Confusion Matrix.
11. torch.flatten() method explanation on StackExchange.
12. ChatGPT architecture recommendations.

References

[1] Yann LeCun; Leon Bottou; Yoshua Bengio; Patrick Haffner. *"Gradient-based learning applied to document recognition"*. Proceedings of the IEEE, 1998.

[2] Alex Krizhevsky; Ilya Sutskever; Geoffrey E. Hinton. *"ImageNet Classification with Deep Convolutional Neural Networks"*. Communications of the ACM. 2017.

[3] Walter J. Scheirer; Anderson Rocha; Archana Sapkota; Terrance E. Boult. *"Towards Open Set Recognition"*. Institute of Electrical and Electronics Engineers. "*Transactions on pattern analysis and machine intelligence*". 2013.

[4] Diederik P. Kingma; Jimmy Lei Ba. *"Adam: A method for stochastic optimization"*. OpenAI, University of Amsterdam & University of Toronto. 2015.

[5] Abhijit Bendale; Terrance E. Boult. *"Towards Open Set Deep Networks"*. University of Colorado at Colorado Springs. 2015.

[6] Dan C. Cireşan; Ueli Meier; Jonathan Masci; Luca M. Gambardella; Jürgen Schmidhuber. *"Flexible, High Performance Convolutional Neural Networks for Image Classification"*. IDSIA, USI and SUPSI, Switzerland. 2011.

[7] Sagar Vaze; Kai Han; Andrea Vedaldi; Andrew Zisserman. *"Open-Set Recognition: A good closed-set classifier is all you need?"*. University of Oxford & The University of Hong Kong. 2022.

Thank you!