

BEN-GURION UNIVERSITY OF THE NEGEV

APPLIED DEEP LEARNING - FALL 2025

---

## Project Summary

---

*Authors:*

Eilon Yaffe   Redacted   Redacted

Submission date: February 28, 2025

## 0 Previous Attempts and Insights Gained

In this section, we will detail our key attempts and the reasoning behind each attempt to reach a supposedly optimal model for the OSR problem. We had an extensive trial and error process, therefore we will focus only on the most noteworthy efforts.

In the following attempts, unless specified otherwise, we used a Combined Dataset for evaluation, consisting of the MNIST test set along with OOD data from CIFAR-10 and Fashion-MNIST, to assess the model's OOD detection capabilities.

### 0.1 First Attempt

After extensive reading and evaluating various approaches, we unanimously agreed to integrate an autoencoder (AE) into our model. We remembered that we previously implemented an autoencoder that reconstructs noisy MNIST dataset images with an accuracy of approximately 85% in HW2. We decided to leverage this existing autoencoder for our open set recognition (OSR) model.

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.code_size = 150

        self.encoder = nn.Sequential(
            nn.Unflatten(1, (1, 28, 28)),
            nn.Conv2d(in_channels=1, out_channels=7, kernel_size=5, stride=1, padding=2),
            nn.SiLU(),
            nn.Conv2d(in_channels=7, out_channels=14, kernel_size=7, stride=2, padding=3),
            nn.SiLU(),
            nn.Conv2d(in_channels=14, out_channels=14, kernel_size=5, stride=1, padding=2),
            nn.SiLU(),
            nn.Flatten(),
            nn.Linear(2744, 512),
            nn.SiLU(),
            nn.Linear(512, self.code_size)
        )

        self.decoder = nn.Sequential(
            nn.Linear(self.code_size, 512),
            nn.SiLU(),
            nn.Linear(512, 2744),
            nn.Unflatten(1, (14, 14, 14)),
            nn.ConvTranspose2d(in_channels=14, out_channels=7, kernel_size=3, stride=2, padding=1, output_padding=1),
            nn.SiLU(),
            nn.ConvTranspose2d(in_channels=7, out_channels=1, kernel_size=3, stride=1, padding=1),
            nn.Tanh(),
            nn.Flatten()
        )

    def forward(self, x):
        encoded_vec = self.encoder(x)
        recon_vec = self.decoder(encoded_vec)
        return recon_vec, encoded_vec

class ML_autoencoder_OS(autoencoder):
    def __init__(self):
        super(ML_autoencoder_OS, self).__init__()

        self.n_classes = 11 # 10 MNIST classes + 1 for "Unknown"
        self.clf = nn.Sequential(
            nn.Linear(self.code_size, self.n_classes),
            nn.LogSoftmax(dim=1),
        )

    def forward(self, x):
        encoded_vec = self.encoder(x)
        recon_vec = self.decoder(encoded_vec)
        preds = self.clf(encoded_vec)
        return recon_vec, preds, encoded_vec
```

Figure 1: First attempt model

As can be observed in figure 1, this model consists of an encoder that compresses input images into a latent space and a decoder that reconstructs the original images from this compressed representation. In addition to reconstruction, the model includes a classifier trained to identify MNIST samples. Our idea was to use the reconstruction error for the OSR problem. Meaning, if the model sees an image that is very unlike the MNIST Sample it was trained on, it will struggle reconstructing it, re-

sulting in a higher reconstruction error, thus enabling us to classify them as out-of-distribution (OOD).

Regarding the classification in our model, we added an extra neuron for the unknown class, which increased the number of classes predicted by the SoftMax layer to 11 (10 known classes and 1 unknown class), effectively resizing the previous AE we had to accommodate this OSR problem.

The model's training procedure employed a dual-loss approach, incorporating both a reconstruction loss and a classification loss. This strategy aimed to improve the model's ability to reconstruct images accurately while simultaneously achieving high classification accuracy on in-distribution (ID) data. For optimization, the Adam optimizer was chosen due to its efficiency and adaptive learning rate capabilities (Refer to figure below for the complete training function code).

```
def train_OSR_model(margin=1.0, num_epochs=300):
    criterion = nn.MSELoss()
    clf_criterion = nn.NLLLoss()
    model = ML_autoencoder_OSR().to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=1e-5)

    # Lists to hold loss across epochs
    loss_train_ae = []
    loss_train_clf = []

    for epoch in tqdm(range(num_epochs)):
        loss_epoch = 0
        loss_ae_epoch = 0
        loss_clf_epoch = 0

        for data in trainloader:
            # Batch loss for updating the model
            loss = 0
            optimizer.zero_grad()
            img, labels = data
            img = img.view(img.size(0), -1).to(device) #resize the image into a vector
            labels = labels.to(device)

            # Forward pass
            recon, preds, embeddings = model(img)

            # Reconstruction loss
            loss_ae_batch = criterion(recon, img)
            loss += loss_ae_batch
            loss_ae_epoch += loss_ae_batch.item()

            # Classification loss
            loss_clf_batch = clf_criterion(preds, labels)
            loss += loss_clf_batch
            loss_clf_epoch += loss_clf_batch.item()

            # Backward pass and optimization
            loss.backward()
            optimizer.step()
            loss_epoch += loss.item()

        # Record losses
        loss_train_ae.append(loss_ae_epoch / len(trainloader))
        loss_train_clf.append(loss_clf_epoch / len(trainloader))

    # Evaluation mode
    model.eval()

    # Evaluation mode
    model.eval()

    # Plot losses
    plt.plot(loss_train_ae, label="Reconstruction loss")
    plt.plot(loss_train_clf, label="Classification loss")

    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.legend()
    plt.grid()
    plt.title("AE Training Loss")
    plt.show()
    return model
```

Figure 2: First attempt - training function

With *self.code\_size* (embedded space) = 150, batch size = 128 and epoch count = 300:

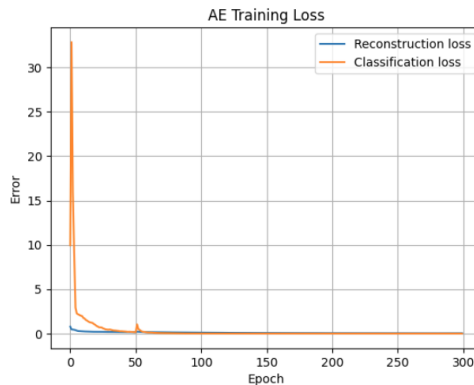


Figure 3: First attempt model - training loss

As can be observed in figure 3, the reconstruction loss and the classification loss generally decrease over time, indicating that the model is learning to reconstruct images and classify them better. In addition, both losses are relatively low (both seem to converge to zero across epochs), which implies the autoencoder part of the model is effectively capturing the essential features of the MNIST images for reconstruction and the model itself is learning to correctly classify the MNIST digits.

Using a threshold of 0.25, we get the next evaluation results:

```
Accuracy on MNIST in OSR: 68.02%
Accuracy on OOD in OSR: 97.58%
Total Accuracy in OSR: 87.72%
Accuracy on MNIST in MNIST test: 88.94%
```

Figure 4: First attempt - results (1)

We aimed to generalize more and mitigate potential overfitting by making the embedding space smaller. Hence, by adjusting the embedded space to 32 and the batch size to 256, we observe that the training loss remains relatively similar to what is depicted in figure 3 - both the reconstruction and classification loss converge to zero. These adjustments enable us to achieve higher evaluation results.

Accuracy on MNIST in OSR: 82.46%	Accuracy on MNIST in OSR: 88.10%
Accuracy on OOD in OSR: 96.45%	Accuracy on OOD in OSR: 91.19%
Total Accuracy in OSR: 91.79%	Total Accuracy in OSR: 90.16%
Accuracy on MNIST in MNIST test: 89.56%	Accuracy on MNIST in MNIST test: 89.56%
(a) threshold = 0.15	(b) threshold = 0.2

Figure 5: First attempt - results (2)

Evidently this has improved the results on the open set.

### Key Insights Summary:

- Model advantages:
  - The model exhibits high accuracy in classifying MNIST digits, demonstrating robust performance for known data.
  - The model shows adaptability to hyperparameters adjustments, achieving improved evaluation results with different hyperparameters values.
- Model disadvantages:
  - The model’s performance is sensitive to reconstruction error threshold changes, needing careful fine-tuning to balance OOD detection and ID classification.
  - The model has a potential overfitting risk, indicating the need for regularization techniques or smaller embedding dimensions.
- Insights:
  - A significant gap between the reconstruction error for OOD samples and the classification error for in-distribution samples is essential. High reconstruction error for OOD samples ensures reliable identification, while low classification error for in-distribution samples confirms accurate recognition. Therefore, to achieve a better performance, we should aim to

maximize the gap between the reconstruction error for OOD samples and the classification error for ID samples.

- The threshold trade-off: desirably, samples with reconstruction error below the threshold are ID, while those exceeding it are OOD. A low threshold increases false positives by misclassifying ID samples as OOD, whereas a high threshold raises false negatives by misclassifying OOD samples as ID. Thus, we should select an appropriate threshold as it is crucial for the overall performance of the OSR model.

## 0.2 Second Attempt

In this attempt, we implemented one key change - we incorporated a contrastive loss into the model's training process.

Based on our research, Contrastive loss is a loss function that is used to learn an embedding space where similar data points are closer together and dissimilar points are farther apart. It is achieved by minimizing the distance between embeddings of similar data points (positive pairs) and maximizing the distance between embeddings of dissimilar data points (negative pairs).

Hence, we chose to incorporate contrastive loss into our OSR model to improve its ability to distinguish between known and unknown classes. As per our understanding, this contrastive loss could help to create a more structured embedding space, where instances of the same class are closer together, and those from different classes are farther apart. We hoped this will enhance the model's capability to recognize OOD samples.

To incorporate contrastive loss into our code, we followed the following steps:

1. We implemented a function that computes the contrastive loss based on the embeddings generated by the autoencoder component of the model and their corresponding labels. This function works by first calculating the pairwise Euclidean distances between all embeddings in a batch. It then identifies positive pairs, which are data points from the same class, and negative pairs, which belong to different classes. The function computes two loss components: a positive loss that minimizes the distance between embeddings of positive pairs, and a negative loss that maximizes the distance between embeddings of negative pairs, but only if the distance is below a specified margin to prevent excessive separation. The final contrastive loss is obtained by combining these two components (code provided in figure 6).
2. We modified the **train\_OSR\_model** function from figure 2. We integrated the contrastive loss by calculating it within the training loop using the **compute\_contrastive\_loss** function. This loss is then combined with the reconstruction and classification losses to form the total loss. In addition, we created a hyper-parameter named *contrastive\_weight* which controls the relative importance of contrastive loss compared to the other losses. Finally, the total loss, including contrastive loss, is used for updating the model's parameters through back-propagation (code provided in figure 7).

```
def compute_contrastive_loss(embeddings, labels, margin_same=margin_same, margin_diff=margin_diff):
    """
    Compute contrastive loss for a batch of embeddings
    Parameters:
        embeddings: tensor of shape (batch_size, embedding_dim)
        labels: tensor of shape (batch_size,)
        margin_same: maximum distance for same class
        margin_diff: minimum distance for different class
    """
    batch_size = embeddings.size(0)
    if batch_size <= 1:
        return torch.tensor(0.0).to(embeddings.device)

    #compute pairwise distances between embeddings
    distances = torch.cdist(embeddings, embeddings, p=2) # Euclidean distance

    #mask for positive pairs and negative pairs
    labels_matrix = labels.unsqueeze(0) == labels.unsqueeze(1)
    positive_pairs = labels_matrix.logical_xor(torch.eye(batch_size, device=embeddings.device).bool())
    negative_pairs = ~labels_matrix

    #loss for positive pairs (push together)
    positive_loss = (distances * positive_pairs.float()).sum() / positive_pairs.sum().clamp(min=1)

    #loss for negative pairs (push apart)
    negative_loss = torch.clamp(margin_diff - distances, min=0) * negative_pairs.float()
    negative_loss = negative_loss.sum() / negative_pairs.sum().clamp(min=1)

    return positive_loss + negative_loss
```

Figure 6: Contrastive loss calculating function

```
def train_OSR_model(margin, num_epochs, contrastive_weight, step_size, decay):
    criterion = nn.MSELoss()
    clf_criterion = nn.NLLLoss()

    model = ML_autoencoder_OSR().to(device)

    optimizer = torch.optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-5)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=decay)

    # Lists to hold loss across epochs
    loss_train_ae = []
    loss_train_clf = []
    loss_train_contrastive = []

    for epoch in tqdm(range(num_epochs)):
        loss_epoch = 0
        loss_ae_epoch = 0
        loss_clf_epoch = 0
        loss_contrastive_epoch = 0

        for data in trainloader:
            # Batch loss for updating the model
            loss = 0
            optimizer.zero_grad()
            img, labels = data
            img = img.view(img.size(0), -1).to(device) #resize the image into a vector
            labels = labels.to(device)

            # Forward pass
            recon, preds, embeddings = model(img)

            # Reconstruction loss
            loss_ae_batch = criterion(recon, img)
            loss += loss_ae_batch
            loss_ae_epoch += loss_ae_batch.item()

            # Classification loss
            loss_clf_batch = clf_criterion(preds, labels)
            loss += loss_clf_batch
            loss_clf_epoch += loss_clf_batch.item()

            # Contrastive loss
            loss_contrastive_batch = compute_contrastive_loss(embeddings, labels, margin_diff=margin)
            loss += contrastive_weight * loss_contrastive_batch
            loss_contrastive_epoch += loss_contrastive_batch.item()

        # Backward pass and optimization
        loss.backward()
        optimizer.step()
        loss_epoch += loss.item()

    # Record losses
    loss_train_ae.append(loss_ae_epoch / len(trainloader))
    loss_train_clf.append(loss_clf_epoch / len(trainloader))
    loss_train_contrastive.append(loss_contrastive_epoch / len(trainloader))

    scheduler.step()

    # Evaluation mode
    model.eval()

    # Plot losses
    plt.plot(loss_train_ae, label="Reconstruction Loss")
    plt.plot(loss_train_clf, label="Classification Loss")
    plt.plot(loss_train_contrastive, label="Contrastive Loss")

    plt.xlabel("Epoch")
    plt.ylabel("Error")
    plt.legend()
    plt.grid()
    plt.title("AE Training Losses")
    plt.show()
    return model
```

Figure 7: Second attempt - training function with contrastive loss

Before beginning the trial-and-error process with contrastive loss, we refactored our code to improve modularity, making it easier to compare different features and configurations. This is facilitated by what we refer to as the 'Quick Tuning Section', which will be further explained in future attempts.

```
'''Model Hyperparameters'''
embed_space = 64 #size of the embedded space, aka the output of the final encoder layer

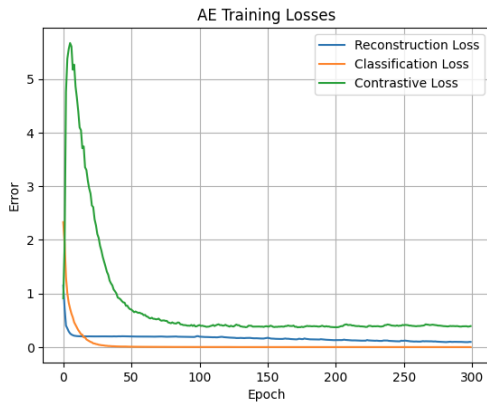
'''Training Hyperparameters'''
margin= 1.0
contrastive_weight = 0.1 # contrastive loss weight within the combined loss in the training function. can zero this to avoid using the contrastive loss in training.
margin_same = 0.01 # distance value for similarly labelled objects in contrastive loss computation
margin_diff = 0.5 # distance value for differently labelled objects in contrastive loss computation
batch_size = 256
num_epochs = 300
lr = 0.001
step_size = 10
decay = 0.5

'''Evaluation Hyperparameters'''
threshold = 0.15
```

Figure 8: Quick Tuning Section

We ran the training function of our model with the hyper-parameters configurations specified in the figure above. We started with rather low values for the contrastive loss parameters and same epoch count / batch size to see if this provides a noticeable improvement without harming our classification ability. We attempted to optimize these hyper-parameters on later attempts.

The results:



Accuracy on MNIST in OSR: 72.62%  
Accuracy on OOD in OSR: 98.47%  
Total Accuracy in OSR: 89.85%  
Accuracy on MNIST in MNIST test: 96.04%

Figure 9: Second attempt - First results

In these results, the contrastive loss does seem to substantially improve the OOD detection, as evident by higher accuracy on OOD data achieved during evaluation. However, this improvement in OOD detection reduced the accuracy on the original MNIST test set, which is a typical trade-off when the model prioritizes creating a feature space that distinguishes between ID and OOD samples.

Also, we would expect to see the contrastive loss gradually decrease over time as the model learns to better differentiate between ID and OOD samples and yet, this trend wasn't shown during the model's evaluation.

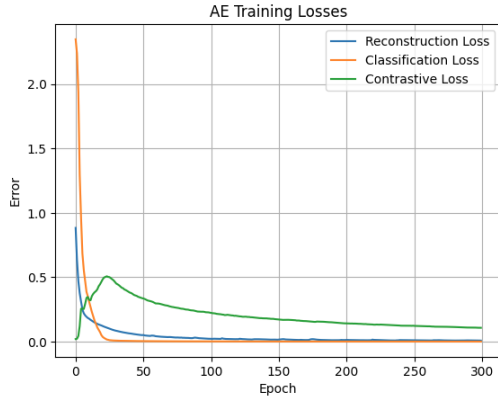
Thus, let us divide the contrastive loss for each batch by the size of the embedded space divided by 2 in an attempt to scale the loss down and to refine the model's hyper-parameters:

```
'''Model Hyperparameters'''
embed_space = 96 #size of the embedded space, aka the output of the final encoder layer

'''Training Hyperparameters'''
margin= 1.0
contrastive_weight = 0.05 # contrastive loss weight within the combined loss in the training function. can zero this to avoid using the contrastive loss in training.
margin_same = 0.01 # distance value for similarly labelled objects in contrastive loss computation
margin_diff = 1.0 # distance value for differently labelled objects in contrastive loss computation
batch_size = 256
num_epochs = 300
lr = 0.001
step_size = 10
decay = 0.5

'''Evaluation Hyperparameters'''
threshold = 0.15
```

Figure 10: Hyperparameters update



Accuracy on MNIST in OSR: 93.74%  
 Accuracy on OOD in OSR: 95.89%  
 Total Accuracy in OSR: 95.18%  
 Accuracy on MNIST in MNIST test: 95.55%

Figure 11: Second attempt - Second results

As can be observed above, the model showed significant improvements, including better classification accuracy and enhanced OOD detection, thanks to the normalized contrastive loss. This normalization allows the model to learn more discriminative embeddings by bringing similar samples closer and separating dissimilar ones, while also stabilizing gradients for faster convergence. Dividing the contrastive loss by half of the embedding dimensionality prevented it from being dominated by the embedding size, ensuring better comparison across dimensions and appropriate contribution to the overall loss.

#### Key Insights Summary:

- The choice of margin, embedded space and the contrastive weight are crucial for the behavior of the contrastive loss. Properly tuning these hyper-parameters can assist to prevent overfitting and improve performance. Hence, experimenting with different values for these parameters is essential in order to find the optimal configuration that maximizes the model's ability to distinguish between ID and OOD samples while maintaining generalization to unseen data.
- The incorporation of contrastive loss in the training process has overall improved the model compared to previous attempt. However, we believe it would be beneficial to postpone the use of contrastive loss for now and revisit it later, once we have significantly improved accuracy on the MNIST. At that point, contrastive loss will likely help the model to better distinguish MNIST samples from OOD ones.



### 0.3 Third Attempt

We started this attempt by redesigning the model's structure to enhance scalability and make modifications more convenient, depicted in the figure below.

```
class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.code_size = embed_space

        self.encoder_init = nn.Sequential(
            nn.Unflatten(1, (1, 28, 28))
        )
        self.encoder_layers = nn.ModuleList([
            nn.Sequential(
                nn.Conv2d(in_channels=1, out_channels=7, kernel_size=5, stride=1, padding=2),
                nn.SiLU(),
            ),
            nn.Sequential(
                nn.Conv2d(in_channels=7, out_channels=14, kernel_size=7, stride=2, padding=3),
                nn.SiLU(),
            ),
            nn.Sequential(
                nn.Conv2d(in_channels=14, out_channels=14, kernel_size=5, stride=1, padding=2),
                nn.SiLU(),
            ),
            nn.Sequential(
                nn.Flatten(),
                nn.Linear(2744, 512),
                nn.SiLU(),
                nn.Linear(512, self.code_size),
            )
        ])

        self.decoder_init = nn.Sequential(
            nn.Linear(self.code_size, 512),
            nn.SiLU(),
            nn.Linear(512, 2744),
            nn.Unflatten(1, (14, 14, 14)),
        )

        self.decoder_layers = nn.ModuleList([
            nn.Sequential(
                nn.ConvTranspose2d(in_channels=14, out_channels=7, kernel_size=3, stride=2, padding=1, output_padding=1),
                nn.SiLU(),
            ),
            nn.Sequential(
                nn.ConvTranspose2d(in_channels=7, out_channels=1, kernel_size=3, stride=1, padding=1),
                nn.Tanh(),
            ),
            nn.Sequential(
                nn.Flatten(),
            )
        ])

    def encode(self, x):
        encoded_features = []
        input = self.encoder_init(x)

        # forward the input through each layer and save the output of each layer in a separate array
        for i in range(len(self.encoder_layers)):
            input = self.encoder_layers[i](input)
            encoded_features.append(input)
        return encoded_features

    def decode(self, encoded_features):
        x = self.decoder_init(encoded_features[3]) #encoded_features[3] is the final encoded vector
        x = self.decoder_layers[0](x)
        x = self.decoder_layers[1](x)
        recon = self.decoder_layers[2](x) #finished reconstruction
        return recon

    def forward(self, x): #unnecessary - only for organized code
        encoded_features = self.encode(x)
        encoded_vector = encoded_features[3]
        recon = self.decode(encoded_features)
        return recon, encoded_vector

class ML_autoencoder_OSR(autoencoder):
    def __init__(self):
        super(ML_autoencoder_OSR, self).__init__()

        self.n_classes = 11 # 10 MNIST classes + 1 for "Unknown"

        self.clf = nn.Sequential(
            nn.Linear(self.code_size, self.n_classes),
            nn.LogSoftmax(dim=1),
        )

    def forward(self, x):
        encoded_features = self.encode(x)
        encoded_vector = encoded_features[3]
        recon = self.decode(encoded_features)
        preds = self.clf(encoded_vector)

        return recon, preds, encoded_vector
```

Figure 12: Previous model updated

We then proceeded by making two substantial changes:

1. Implementation of skip connections in both the AE and the classification process.
2. Addition of a classifier that does not use the AE's encoded vector as input.

These changes required the addition of new binary variables (flags), which we used to compare our model's performance with and without these additions, as can be seen in the figure below:

```
'''Model Hyperparameters'''
embed_space = 64 #size of the embedded space, aka the output of the final encoder layer
skip_recon = True #flag variable for enabling skip connections for reconstruction
skip_classification = False #flag variable for enabling skip connections for classification
dim_reduction_out = 14 # number of output channels of the final convolutional layer
use_simpleCNN = True # uses the simpleCNN structure instead of the encoded vector-based classifier layers.

'''Data Preparation Hyperparameters'''
data_augmentation = False #flag variable for enabling data augmentation

'''Training Hyperparameters'''
contrastive_weight = 0.1 # default contrastive weight for the contrastive loss weight within the combined loss in the training function.
margin_same = 0.01 # default value for the min. distance value for similarly labelled objects in contrastive loss computation
margin_diff = 0.75 # default value for the min. distance value for differently labelled objects in contrastive loss computation
batch_size = 128
num_epochs = 100
lr = 0.004
step_size = 10
decay = 0.5

'''Evaluation Hyperparameters'''
threshold = 0.015
```

Figure 13: Quick Tuning Section - updated

In our previous attempts, we could not observe a clear demarcation between the OOD and ID reconstructions, as there was always an apparent trade-off between the accuracy on ID and OOD. We sought to improve the "margin" of the threshold, and figured improving the reconstruction capability of our autoencoder would be a good way to start.

An obvious way to improve the autoencoder's reconstruction capability without making significant alterations to the layers is to implement skip connections.

We decided to implement the skip connections by passing the 2nd convolutional layer of the encoder along to the 1st decoder layer, and the 1st convolutional layer of the encoder to the 2nd decoder layer. This required some restructuring of the model, mostly having the encode function return an array of the encoded features captured in each layer, and making appropriate changes in the decoder layers and the decode function.

```
def decode(self, encoded_features):
    x = self.decoder_init(encoded_features[3]) #encoded_features[3] is the final encoded vector
    if skip_recon: #forward the input through decoder layers with skip connections
        x = torch.cat([x, encoded_features[1]], dim=1)
        x = self.decoder_layers[0](x)
        x = torch.cat([x, encoded_features[0]], dim=1)
        x = self.decoder_layers[1](x)
    else:
        dummy_zero = torch.zeros_like(encoded_features[0]) #dummy inputs to match the size of the input channels
        dummy_one = torch.zeros_like(encoded_features[1])
        x = torch.cat([x, dummy_one], dim=1)
        x = self.decoder_layers[0](x)
        x = torch.cat([x, dummy_zero], dim=1)
        x = self.decoder_layers[1](x)

    recon = self.decoder_layers[2](x) #finished reconstruction, as a 2D vector of (batch_size, 28 * 28)

    return recon
```

Figure 14: Decode function with skip connections

by enabling skip connections, we effectively pass more information over to the decoder, which would allow it to capture more features and produce reconstructions that are closer to the original. This,

however, also improves the model’s ability to accurately reconstruct OOD images, which is detrimental to our goal. to combat this, we decided to make the embedded space substantially smaller (Most trial runs with skip connections used an embedded space size of 8 or 16).

The second skip connection implementation happens within the OSR model itself, designed to add more features from other layers of the encoder into the final classifier layer.

```
def forward(self, x):
    encoded_features = self.encode(x)
    encoded_vector = encoded_features[3] #grabbing the final encoded_vector, for use in classification later
    preds = encoded_vector

    recon = self.decode(encoded_features)

    if use_simpleCNN:
        preds = self.simpleCNN(self.encoder_init(x))
    else:
        if skip_classification: #classification section via skip connection
            pooled_features = [ #first we use average pooling to fit conv1 and conv2 in the same dimensions as conv3, and stick them in an array
                F.adaptive_avg_pool2d(encoded_features[0], self.concat_dims),
                F.adaptive_avg_pool2d(encoded_features[1], self.concat_dims),
                encoded_features[2], #conv3 is used as is
            ]
            encoded_spatial = encoded_vector.view(encoded_vector.size(0), -1, 1, 1) #this converts our encoded vector to a multi-dimensional tensor of four dimensions, where -1 tells
            #the function to automatically calculate the size, as it varies based on our code size hyperparameter.
            encoded_spatial = encoded_spatial.expand(-1, -1, self.concat_dims[0], self.concat_dims[1]) #this compresses (or expands) the encoded vector so that it fits our
            #requirements for the third and fourth dimensions (to align with conv1, 2, and 3)
            concat_vector = torch.cat([encoded_spatial, pooled_features[0], pooled_features[1], pooled_features[2]], dim=1)
            preds = self.channel_reduction(concat_vector)

        preds = self.clf(preds)

    return recon, preds, encoded_vector
```

Figure 15: OSR forward function with skip connections

However, since the classifier only has one FC layer, we added another sequence of layers, that serves as a ‘dimensionality reduction’ sequence that compresses all the learned features into a single channel. This sequence is comprised of a convolutional layer that has an output channel count determined by the hyperparameter **dim\_reduction\_out** (its tuning in figure 13), followed by an activation function, a tensor flattening and finally a FC layer to fit the input size of the original classifier.

Accuracy on MNIST in OSR: 94.92%  
 Accuracy on OOD in OSR: 99.98%  
 Total Accuracy in OSR: 98.29%

Figure 16: Results only with skip connections in AE

We will also note that the necessary threshold dropped significantly, down to 0.005 for the results shown in the figure above.

Accuracy on MNIST in OSR: 93.83%  
 Accuracy on OOD in OSR: 99.93%  
 Total Accuracy in OSR: 97.90%

Figure 17: Results with both skip connection implementations

There were no noticeable improvements in accuracy with the skip connections in the classifier portion of the model, as can be observed in Figure 17, and we had experienced a sharp drop in classification accuracy once the embedded space size exceeded a certain size (96).

This convinced us to drop the idea of utilizing skip connections for the classifier.

However, we still needed a way to improve our MNIST classification accuracy, so we decided to implement a new classifier which is based on CNN, that takes the original input rather than the encoded vector.

```

self.simpleCNN = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.MaxPool2d(2),
    nn.Flatten(),
    nn.Linear(64 * 7 * 7, 128),
    nn.BatchNorm1d(128),
    nn.Linear(128, self.n_classes),
    nn.LogSoftmax(dim=1),
)

```

Accuracy on MNIST in OSR: 95.93%  
 Accuracy on OOD in OSR: 99.89%  
 Total Accuracy in OSR: 98.57%  
 Accuracy on MNIST in MNIST test: 96.62%

Figure 18: SimpleCNN classifier structure and results

Evidently, this produced a not-insignificant improvement in the MNIST classification category (a 2% rise in accuracy).

### Key Insights Summary:

- Despite of our initial hypothesis, adding skip connections to our autoencoder didn't decrease the gap between the reconstruction losses of ID and OOD samples. We are not exactly sure why that is the case; One conjecture is that with the earlier encoder layers contributing directly to the decoder, the optimizer had skewed the weights in these layers in favor of retaining MNIST specific features, which helped MNIST images achieve substantially lower reconstruction losses compared to OOD ones.
- Not utilizing the encoded vector for the classification and using a CNN classifier instead substantially improved our MNIST classification accuracy, and led us to the conclusion that we might be better served completely separating the two, which would make us much more flexible when it comes to adjustments in each part (and also led us to better understanding the idea of a baseline model).

## 0.4 Fourth Attempt

We will go over three substantial changes made in this attempt:

1. Addition of a dynamically computed threshold (Henceforth called the adaptive threshold).
2. A new, more robust CNN classifier model.
3. New data augmentations for the train set.

Let's begin with the adaptive threshold - we realized that manually adjusting our threshold with every new iteration results is not very efficient, and that whatever manual threshold we settle on might be a poor fit for the test dataset.

As such, we decided to take a dynamically calculated threshold that relies on the training data. The way the threshold is calculated is as follows: We first assume that the reconstruction losses provided by the last epoch are **approximately** I.I.D, and calculate the threshold as:  $\mu + \alpha * \sigma$ , where  $\mu$  is the standard mean of the reconstruction losses,  $\sigma$  is the standard deviation, and  $\alpha$  is a hyper-parameter. This lead us to consider utilizing the empirical rule (Also known as the  $3\sigma$  rule), which states that approximately 99.7% of the values within a normal distribution lie within three standard deviations of the mean.

Naturally, our reconstruction losses are not truly I.I.D, so it shouldn't (And it doesn't) capture 99.7% (Or rather, 99.85%, given that the outliers below the mean are already accounted for) of the ID samples. However, repeated trials did show that that this idea has merit, and this method both managed

to capture a sufficient number of samples with a  $\alpha$  value of 3, and served as a guideline for further manual tuning of the threshold when needed.

```
# Evaluation mode
recon_err = torch.tensor(last_epoch_recon_errors).clone().detach().cpu()
mean = torch.mean(recon_err) #calculating the mean, item() converts the scalar tensor to a float
std = torch.std(recon_err)
model.threshold = mean + z_scalar * std #using the 3 standard deviation trick, which encompasses 99.7% of MNIST errors according to the normal distribution.
```

Figure 19: Calculating threshold in the training function

The figure above shows how we calculated the threshold - we took the reconstruction losses from the last epoch, then used a hyper-parameter as a scalar for the deviation. we usually tested with a value of either 3 or 2.576 (A lower scalar value helps to identify more OOD samples correctly, while missing a few more ID samples).

FashionMNIST and CIFAR10 as OOD:	KMNIST as OOD:
Accuracy on MNIST: 97.49%	Accuracy on MNIST: 97.49%
Accuracy on OOD: 97.40%	Accuracy on OOD: 78.81%
Total Accuracy: 97.43%	Total Accuracy: 88.15%
EMNIST as OOD:	SVHN as OOD:
Accuracy on MNIST: 97.49%	Accuracy on MNIST: 97.49%
Accuracy on OOD: 0.77%	Accuracy on OOD: 99.99%
Total Accuracy: 32.18%	Total Accuracy: 99.30%

Figure 20: Results with the adaptive threshold

As can be seen in figure 20, we've added a few more datasets as OOD to verify that we're not custom-fitting our model to work well only on FashionMNIST and CIFAR10 as OOD. It can be seen that the adaptive threshold works decently well as a threshold for FashionMNIST / CIFAR10, which is the only thing we can properly use as a benchmark at this stage.

The results on EMNIST and KMNIST suggest that our autoencoder reconstructs the OOD samples well enough to the point where it's very hard to distinguish between these samples and the MNIST digit samples based on the loss.

We will also note that our accuracy on MNIST classification improved due to the changed classifier model, which we will explain later on in this section.

One benefit of separating our reconstruction and classifier models is that we can approach each task individually, which means that we can simply search for proven and effective open source models online to use.

For our classifier, we used a variation of the model shown [here](#).

```

self.simpleCNN = nn.Sequential(
    nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm2d(32),
    nn.Conv2d(in_channels=32, out_channels=32, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm2d(32),
    nn.MaxPool2d(2),
    nn.Dropout(0.25),
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm2d(64),
    nn.Dropout(0.25),
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
    nn.ReLU(),
    nn.BatchNorm2d(128),
    nn.MaxPool2d(2),
    nn.Dropout(0.25),
    nn.Flatten(),
    nn.Linear(128 * 7 * 7, 512),
    nn.ReLU(),
    nn.BatchNorm1d(512),
    nn.Dropout(0.5),
    nn.Linear(512, 128),
    nn.BatchNorm1d(128),
    nn.Dropout(0.5),
    nn.Linear(128, self.n_classes),
    nn.LogSoftmax(dim=1),
)

```

Figure 21: New CNN classifier model

```

transform_augmented = transforms.Compose([
    transforms.RandomRotation(degrees=8),
    transforms.RandomAffine(degrees=0, shear=0.3),
    transforms.RandomResizedCrop(size=28, scale=(1-0.08, 1+0.08)),
    transforms.RandomAffine(degrees=0, translate=(0.08, 0.08)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

```

Figure 22: New data augmentation for training dataset

This new data augmentation had actually improved our classification, to the point where we've gotten results of up to 98.8% accuracy in MNIST-only tests with a pseudo-infinite threshold.

However, training the autoencoder on the augmented train set harmed performance and made the demarcation between ID and OOD reconstruction losses a lot more blurry.

This posed a problem that we later decided to solve by exclusively feeding augmented input to the classifier, and not the autoencoder (As will be seen in the next attempt).

We have also tried to add noise to the input being fed to the autoencoder, as it's a variation on the model we've used to denoise images back in task 2 (The assumption being that it will be able to denoise the MNIST images fairly well, while ruining its chances of properly reconstructing untrained OOD images), however, it didn't manage to provide any meaningful insights / improvements over the previous attempts.

Perhaps it would've changed if we'd gone with a far more aggressive noise addition, but we opted to scrap this approach.

### Key Insights Summary:

- Our current autoencoder with skip connections performs a bit too well when it comes to reconstructing untrained images, and there's a need for a simpler, "dumber" model.
- Aggressive augmentations on the input will blur the line between ID and OOD images for the autoencoder, thus they should be avoided.
- The adaptive threshold works well for finding an initial threshold that encompasses most of the ID images.

### 0.5 Fifth Attempt

The fifth attempt was mostly about implementing the insights we've gathered in the previous attempt. As such, the changes made were:

1. Replacement of the skip connection denoising AE with a simpler AE, with no skip connections.
2. Separating the input during the model forward pass, and only feeding unaugmented data to the autoencoder to minimize generalization.

The new autoencoder model only utilizes two convolutional layers in the encoder, as opposed to three convolutional layers in the previous encoder, and it utilizes both batch normalization and max pooling, which in theory should prevent the model from being able to produce high-resolution reconstructions. On top of that, it utilizes ReLU activation functions instead of SiLU, as the idea of vanishing negative values seemed fitting for our goal of producing bad OOD reconstructions.



Figure 23: Reconstruction vs Original - OOD example from SVHN dataset

As can be seen in the figure above, the autoencoder has completely failed to capture and reconstruct any meaningful features from the original image, which is what we aimed for.

```

class autoencoder(nn.Module):
    def __init__(self):
        super(autoencoder, self).__init__()
        self.code_size = embed_space
        self.threshold = 0

        self.encoder_init = nn.Sequential(
            nn.Unflatten(1, (1, 28, 28))
        )
        self.encoder_layers = nn.ModuleList([
            nn.Sequential(
                nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1),
                nn.BatchNorm2d(16),
                nn.ReLU(),
                nn.MaxPool2d(2,2),
            ),
            nn.Sequential(
                nn.Conv2d(in_channels=16, out_channels=4, kernel_size=3, padding=1),
                nn.BatchNorm2d(4),
                nn.ReLU(),
                nn.MaxPool2d(2,2),
            ),
            nn.Sequential(
                nn.Flatten(),
                nn.BatchNorm1d(4 * 7 * 7),
                nn.Linear(4 * 7 * 7, self.code_size),
                nn.LeakyReLU(0.2),
            )
        ])
        self.decoder_layers = nn.ModuleList([
            nn.Sequential(
                nn.Linear(self.code_size, 4 * 7 * 7),
                nn.BatchNorm1d(4 * 7 * 7),
                nn.Unflatten(1, (4, 7, 7)),
            ),
            nn.Sequential(
                nn.ConvTranspose2d(in_channels=4, out_channels=16, kernel_size=2, stride=2),
                nn.BatchNorm2d(16),
                nn.ReLU(),
            ),
            nn.Sequential(
                nn.ConvTranspose2d(in_channels=16, out_channels=1, kernel_size=2, stride=2),
                nn.Tanh(),
                nn.Flatten(),
            )
        ])
    )

```

Figure 24: New Autoencoder model

However, even with the current Autoencoder model, using the augmented trainset exclusively couldn't produce the results we wanted, so we were forced to train the autoencoder on the original input (Which was still transformed according to instructions), which is why we've made alterations to the OSR model's forward function, specifically in how it forwards input to each model.

```

def forward(self, x):
    encoded_features = self.encode(x)
    encoded_vector = encoded_features[-1]
    recon = self.decode(encoded_features)
    recon_loss = torch.mean((recon - x) ** 2, dim=1) #contains a 2D tensor of (batch size, 1)

    aug_input = self.encoder_init(x) #simply unflattens the samples
    if data_augmentation and augment_clf_only:
        aug_input = self.augmentation(aug_input)

    if use_simpleCNN:
        preds = self.simpleCNN(aug_input)
    else:
        preds = self.clf(encoded_vector)

    if not self.training: #OOD determination in evaluation only
        is_ood = recon_loss > self.threshold #boolean mask for all indices within the batch that exceed the rejection threshold
        for i in range(len(is_ood)):
            if is_ood[i]: #if true, then this index of recon_loss satisfied the rejection threshold inequation
                preds[i, :10] = 0
                preds[i, 10] = 1 #sets the unknown class value to be higher than the rest
        return preds, recon
    return preds, recon, encoded_vector

```

Figure 25: The adjusted forward function of the OSR model



As can be seen in the figure above, we implemented this through yet another hyper-parameter called `augment_clf_only`, which controls whether or not we pass augmented data only to the classifier (And do so within the model), or, if false, proceed with augmenting the entire train set during the data preprocessing stage.

We have also copied the transformations into an attribute of the model, referred to in the code as `self.augmentation`.

<b>FashionMNIST and CIFAR10 as OOD:</b>	<b>KMNIST as OOD:</b>
Accuracy on MNIST: 94.18%	Accuracy on MNIST: 94.37%
Accuracy on OOD: 98.19%	Accuracy on OOD: 95.09%
Total Accuracy: 96.86%	Total Accuracy: 94.73%
<b>EMNIST as OOD:</b>	<b>SVHN as OOD:</b>
Accuracy on MNIST: 94.36%	Accuracy on MNIST: 94.14%
Accuracy on OOD: 60.51%	Accuracy on OOD: 100.00%
Total Accuracy: 71.50%	Total Accuracy: 98.37%

Figure 26: Final results of attempt 5

**To summarize,** This attempt had substantial improvements in OOD recognition compared to the previous attempts, at the cost of lowered MNIST classification accuracy. This likely stems from a lower threshold, which suggests that the variance of the reconstruction losses over ID samples is bigger than before (That is to say, it's quite bad at reconstructing certain outlier images).

As the model currently stands, the test set samples are also going through the same augmentation that the train set samples go through.

This presented an issue, and upon disabling that augmentation for the test set, we saw that our classification accuracy dropped to roughly 60%. **This issue has been fixed in the final model version.**

# 1 Final Model

## 1.1 Methodology and Rationale

### Overview of the rationale

Our initial plan was to approach the Open Set Recognition problem with the tools and methods we've learned throughout the course.

After researching common approaches to handling the task, and an extensive trial and error, we found an approach that in a nutshell utilizes a rejection threshold based on reconstruction losses.

By training an image reconstruction model to reconstruct MNIST samples, we managed to achieve relatively low reconstruction losses on In-Distribution (ID) samples on average and a relatively high reconstruction losses on Out-Of-Distribution (OOD) samples on average, and use that gap in the respective loss averages as a separator (or a threshold), that determines whether a given input should be labeled as a Known or Unknown class.

With that idea in mind, we strove to find a model architecture that could achieve two distinct goals:

- 1) ID image reconstruction with a small reconstruction loss.
- 2) OOD image reconstruction with a high reconstruction loss.

This outcome would allow us to find a threshold that neatly separates the two classes, and is thus ideal. Thankfully, we have learned about one such architecture that seemingly fit our needs perfectly - Autoencoders.

Autoencoders can compress their learned features into a small embedded vector, meaning that by making the embedded vector small enough, it would struggle to reconstruct anything it had not explicitly been trained on.

This should, in theory, produce high reconstruction losses for OOD images. On top of that, we have already seen in class that simple autoencoders can provide good reconstructions of MNIST images even with a small bottleneck (Size of the embedded vector).

As the autoencoder architecture seemed capable of satisfying both requirements, we decided to proceed with an autoencoder model as our 'image reconstructor' and a CNN model as our 'classifier', with the OSR model working in the following manner - it first sends the input through the reconstructor and the classifier, and should the reconstruction loss go past the threshold, it will receive the prediction of the Unknown class. Otherwise, it will return the same label as the classifier.

### Review of the methodology

We will go over each part of the model, the data preprocessing and the training alterations we made:

1. OSR Model - the OSR model is an integrated model that utilizes two separate models as attributes - a convolutional autoencoder and a convolutional classifier. It also keeps a threshold as an attribute, that is computed at the end of the training process and utilized during evaluation. The OSR model behaves differently depending on whether it is in training or evaluation mode:

During training, the OSR model forwards the input through the autoencoder, and receives a flattened tensor of the reconstructions. After that, it forwards the original input through the classifier, and returns both the predictions and the reconstructions, so that the training function's optimizer may improve both losses.

During evaluation, the OSR model does largely the same thing as the training process, with an added section where it calculates the reconstruction losses itself, and for each sample within the batch that has a loss that exceeds the threshold, it alters the classifier's predicted label so that it will be recognized as an Unknown class.

2. Autoencoder Model - The autoencoder model is a fairly basic symmetrical convolutional autoencoder, with two convolutional layers followed by a linear layer in the encoder, and a linear layer followed by two convolutional layers in the decoder.

The model utilizes ReLU activation functions specifically as these produce the highest reconstruction losses on untrained, OOD images by setting entire sections of pixels to zero due to poorly fitting weights (examples shown in the code notebook and evaluation section of this paper), while managing to reconstruct MNIST images comparatively well. This provides a relatively clean separation between ID and OOD images.

3. Classifier Model - the classifier model is a CNN with six convolutional layers and two linear layers, designed for classifying MNIST images that produces good results - up to 98.9% accuracy (as shown in the code notebook under "Baseline Model Total Accuracy"). Due to the reliance of the classifier on augmented data for better results, and the reliance of the autoencoder on non-augmented data for a neater separation of ID and OOD samples, we implemented a method to perform the augmentation during the training process exclusively for the classifier.

This method initially splits each batch split into two parts, one of which passes through a sequence of image transformations, and another that remains as is. After that, both parts are concatenated into a single tensor and forwarded through all the classifier layers\*\*.

4. Baseline Model - the baseline model and the classifier model are synonymous in our project; The difference between the OSR model and the baseline model is that the baseline does not include the image reconstruction through the AE, nor does it include the changes done in the OSR model's forward function.
5. Data Preprocessing - The training and test sets are both resized to (1, 28, 28), transformed into greyscale (If necessary), changed into tensor form and then normalized with a mean and deviation of 0.5.
6. Training Process - the training process uses a weighted sum of three different losses, with different criteria for each, the Adam optimizer and a learning rate scheduler.  
The first loss is the simplest, which is the classification loss - it is computed with the common cross entropy loss function.

The second loss is the reconstruction loss, which is computed using MSE loss; effectively the square of the difference between the original input and the returned reconstruction.

The third loss is the contrastive loss - this loss is designed to help during classification by forcing the difference between differently labeled samples to widen and forcing the gap between similarly labeled samples to narrow. This happens by forwarding an embedded vector of the classifier\*\*\* (called embeddings in the code) to the loss computation function.

The contrastive loss is then divided by the size of the embedded vector and given a weight as a hyper-parameter.

The final loss upon which we perform backpropagation is calculated as:

classification loss + reconstruction loss + (contrastive weight \* contrastive loss).

After the final epoch, the training loss calculates the rejection threshold based on the reconstruction losses, relying on principles from the Central Limit Theorem. This process is explained in depth in attempt 4.

\*\* Fully augmenting the entire input batch for the classifier resulted in a substantial MNIST classification accuracy drop during evaluation, when the test set was unaugmented; In fact, on average, the accuracy on an unaugmented MNIST-only test set with a fully augmented input was around 60%, while the accuracy on an augmented MNIST-only test set was roughly 98.6% (Despite having roughly the same final classification loss during training).

We concluded that this is simply due to the augmentation being a bit too aggressive, and decided to mix it together with some unaltered images to provide better results on an unaugmented test set, even though it produced a lower accuracy than what we've gotten on an augmented test set.

We have done so as we do not know what kinds of transformations may be applied on future test sets, and it seemed prudent to maximize the unaugmented results.

\*\*\* In earlier attempts, such as attempt 2, the embeddings used for the contrastive loss were taken from the last encoder layer of the autoencoder, which made sense at the time given that it was used for classification.

After the complete separation of the AE and the classifier, this no longer made sense, so the embeddings vector was changed to be one of the latest layers of the classifier.

- To document the various parts of our code, in addition to standard use of comments for the more intricate parts of the code, we used the Google Style Documentation for non-elementary functions.

## 1.2 Hyper-parameters and Configuration

```
'''Model Hyperparameters'''
embedding_space = 32

'''Training Hyperparameters'''
contrastive_weight = 1
margin_diff = 1

batch_size = 200
baseline_num_epochs = 100
OSR_num_epochs = 300
lr = 0.01
step_size = 25
decay = 0.4
z_scalar = 3
```

Figure 27: Final configuration of hyper-parameters used

In-depth explanation of each hyper-parameter:

- **embedding\_space:** The embedding\_space represents the second dimension of the input tensor after passing through the encoder's final layer, with the first dimension being the batch size (It is also known as the latent space size).  
It is effectively a bottleneck on how many features can be transferred over to the decoder for reconstruction purposes.  
A smaller embedding\_space value restricts the amount of information the decoder can use, which means it will generally perform worse on untrained datasets.  
On trained datasets, a smaller bottleneck can sometimes improve results by forcing the decoder to generalize due to having fewer features to extrapolate from, but it changes on a case-by-case

basis.

In our case, the trade-off comes in the form of balancing maintenance of good ID reconstructions and bad OOD reconstructions.

The final selected value is `embedding_space = 32`, as that value is the lowest we could reliably go while still maintaining decent reconstructions of ID images over multiple training iterations.

- **contrastive\_weight:** The `contrastive_weight` is quite literal in its purpose; it is a scalar value that multiplies the contrastive loss before it's being added to the general loss.

Since the general loss is made up of a sum of three different losses, there is a need to balance them such that the gradient won't favor one particular loss too much, and that is where the weight hyper-parameter comes in.

The lower the value, the less emphasis we give to the clustering of the MNIST images during the classification process, and more emphasis we give to the generic classification and reconstruction optimization.

The final selected value is `contrastive_weight = 1`, the decision being based almost exclusively on empirical evidence gathered during prior training and evaluation attempts.

- **margin\_diff:** This hyper-parameter controls the minimum value the distance between two differently labeled embeddings can be, before they incur a penalty.

This means that with a higher value, more pairs will result in a penalty and will eventually result in a higher overall contrastive loss value.

Tuning this parameter for the best results was mostly an empirical effort.

The final selected value is `margin_diff = 1`.

- **batch\_size:** the batch size is a standard hyper-parameter in machine learning; it represents the size of each input group that passes through the model.

A smaller batch size usually improves accuracy, but at the cost of running more batches per epoch, which slows down the training and evaluation processes. Comparatively, a bigger batch size would allow for faster executions of the training and evaluation processes, but might lead to slight overfitting.

The final selected value is `batch_size = 200` - We initially planned to go with 256 for the final model, but we had spare computational overhead, and thus decided to lower our batch size to try and make some slight improvements in accuracy while maintaining our training goals.

- **baseline\_num\_epochs & OSR\_num\_epochs:** Again, these hyper-parameters are both common in machine learning and their role is self-evident; they control how many epochs we perform during training for the baseline and OSR models, respectively.

A higher epoch count usually leads to better refinement of the weights, at the cost of a higher computation cost.

In our case, the classifier model needs significantly fewer epochs to reach near-optimal accuracy, compared to our autoencoder; this is why the two epoch counts are separate.

The final selected values for these are `baseline_num_epochs = 100` and `OSR_num_epochs = 300`. Simply put, both models usually stagnated entirely by the time they reached these epoch counts.

- **lr:** Stands for Learning Rate, another one of the common-use machine learning hyper-parameters.

This is effectively a scalar for the size of the "step" the model takes in the direction opposite to the gradient per training iteration. It is dynamically adjusted during training through a scheduler. High learning rates mean that the model takes large "steps" in an appropriate direction, which in theory should reduce the number of "steps" needed to reach an optimal result, however, they also tend to get stuck on local minimum values if the function is not convex, and end up not improving at all over many iterations.

In contrast, smaller values result in having to take more "steps", but they're less likely to miss avenues of improvement.

We have seen that setting a higher initial learning rate improves our AE's accuracy, which is also why we decided to use a scheduler.

The final selected value is  $lr = 0.01$ , as this initial learning rate significantly improved our AE's accuracy instead of the slightly more common 0.001.

- **step\_size:** This hyper-parameter determines how often the scheduler adjusts the learning rate - a higher value would mean that the learning rate changes less often, and vice versa.

The final selected value for this is  $step\_size = 25$ . We set it to 25 as we considered that our initial learning rate was set perhaps a bit too high for making small jumps, and wanted to swiftly lower it down to a smaller value. It proved to be an improvement over larger step sizes like 50 or 100.

- **decay:** This hyper-parameter controls the gamma value of the scheduler; In essence, it is the rate by which we reduce the learning rate every  $step\_size$  iterations.

We set it to a rather high value in order to produce a rather smooth descent for the learning rate, which seemed to produce the best result.

The final selected value is  $decay = 0.4$  - In most recent tests, this value hovered between 0.1 and 0.8, and 0.4 proved to be best, even though values in this interval resulted in minimal changes in the total evaluation.

- **z\_scalar:** The  $z\_scalar$  controls the scale of the standard deviation when calculating the threshold during the training process.

The idea for the  $z\_scalar$  was taken straight from the  $z$  value of the  $z$  table for a standard Normal distribution.

The higher the value, the further away from the reconstruction loss mean a sample could get while still being flagged as ID (Essentially, it raises the threshold).

The final selected value is  $z\_scalar = 3$ . As explained in the fourth attempt, according to the empirical rule, a value of three should capture most of the ID samples within its range, which is why we decided to go with it. Lower values - such as 2.576 - had managed to increase our OOD accuracy slightly, but the drop in ID accuracy made the change not worth going through with.

- **threshold:** The final hyper-parameter. it allows us to set a manual threshold in case we were unsatisfied with the adaptive threshold value.

If set to 0, the model uses the adaptive threshold learned during training.

The final selected value is  $threshold = 0$ . We were largely satisfied with the adaptive threshold's results, and tuning the threshold manually provided little to no improvement.

## 1.3 Results and Figures

### 1.3.1 Training

#### Baseline model

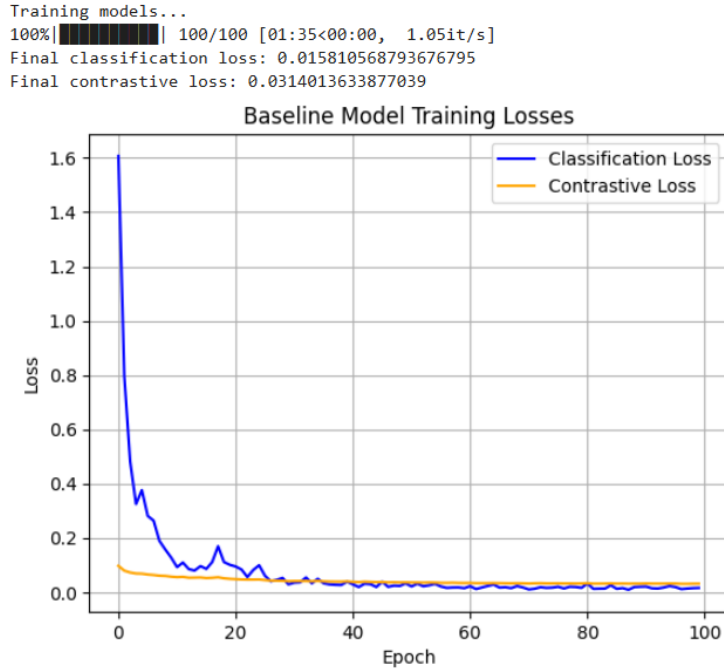


Figure 28: Final training result, baseline model

As can be seen above, the baseline training process only utilizes two losses - the standard classification loss, which behaves somewhat erratically but with a clear downwards trend, and a contrastive loss that starts with a small value and very gradually converges to roughly 0.03.

The behavior of the classification loss is not unexpected; rather, it stems from the somewhat aggressive transformations we apply to the training samples. This could be adjusted by making the augmentation / original split more in favor of original samples, but it produces slightly worse overall results.

We designed the contrastive loss to start at a small value (Achieved through dividing it by the size of the embeddings tensor) as we didn't want it to have significant impact during the early phase of the training process, but rather for it to carry more weight during the latter part of the training. In this way, it can be viewed as a form of a gentle clustering algorithm that only comes into play once the model starts to stabilize.

## OSR model

```
100%|██████████| 300/300 [04:23<00:00, 1.14it/s]
Threshold: 0.10214029252529144
Mean: 0.043423380702733994
Std: 0.01957230269908905
Final classification loss: 0.010908865137025714
Final contrastive loss: 0.028902561217546464
Final reconstruction loss: 0.04342338293790817
```

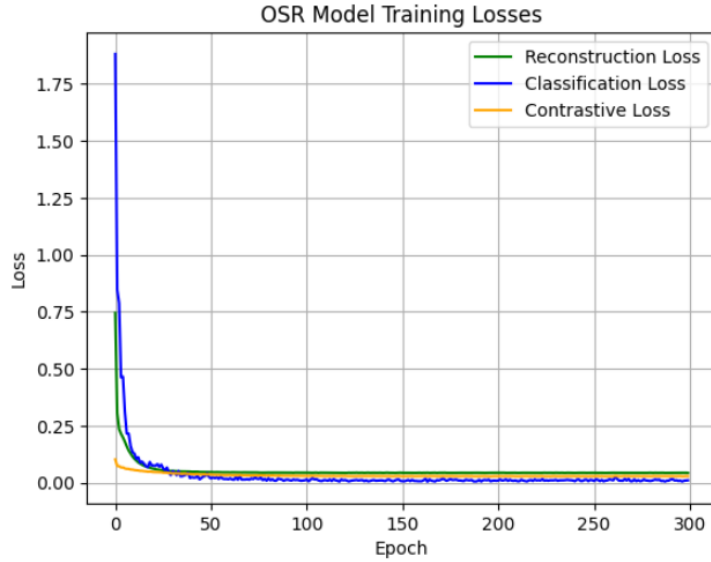


Figure 29: Final training result, OSR model

Similar to the baseline model, the OSR model also utilizes the classification and contrastive losses, and both of them behave in the same manner, though it can be seen that the classification loss stabilized further due to the greater epoch count.

Unlike the baseline model, however, the OSR training process also utilizes a reconstruction loss that adjusts the autoencoder, which is responsible for separating ID and OOD samples during evaluation.

This reconstruction loss stagnates at around 0.04 instead of converging to zero, which is actually intentional - we didn't want our autoencoder to be able to produce accurate reconstructions, as that could improve the reconstruction capability over OOD images.

We believe this stagnation is caused by the small `embed_space` value we've chosen.



### 1.3.2 Evaluation

For this section we used several auxiliary functions (a more technical documentation of these functions can be found in the provided code notebook of the project):

1. *evaluate\_and\_visualize* – Visualizes original and reconstructed images, highlighting how the OSR model reconstructs OOD samples poorly while handling MNIST samples well.
2. *plot\_confusion\_matrix* – Generates a heatmap-based confusion matrix to assess OSR model performance.
3. *evaluate\_model\_accuracy* – Computes accuracy and plots a confusion matrix for the baseline model on the MNIST test dataset.
4. *eval\_binary\_classification* – Evaluates the OSR model in a binary setting (MNIST as 0, OOD as 1) and plots a confusion matrix.
5. *extract\_embeddings\_and\_predictions* – Extracts embeddings and predictions from the OSR model for t-SNE visualization.
6. *tsne\_plot* – Applies t-SNE to embeddings, visualizing their separation in a 2D space to show clustering patterns.

### Baseline Results

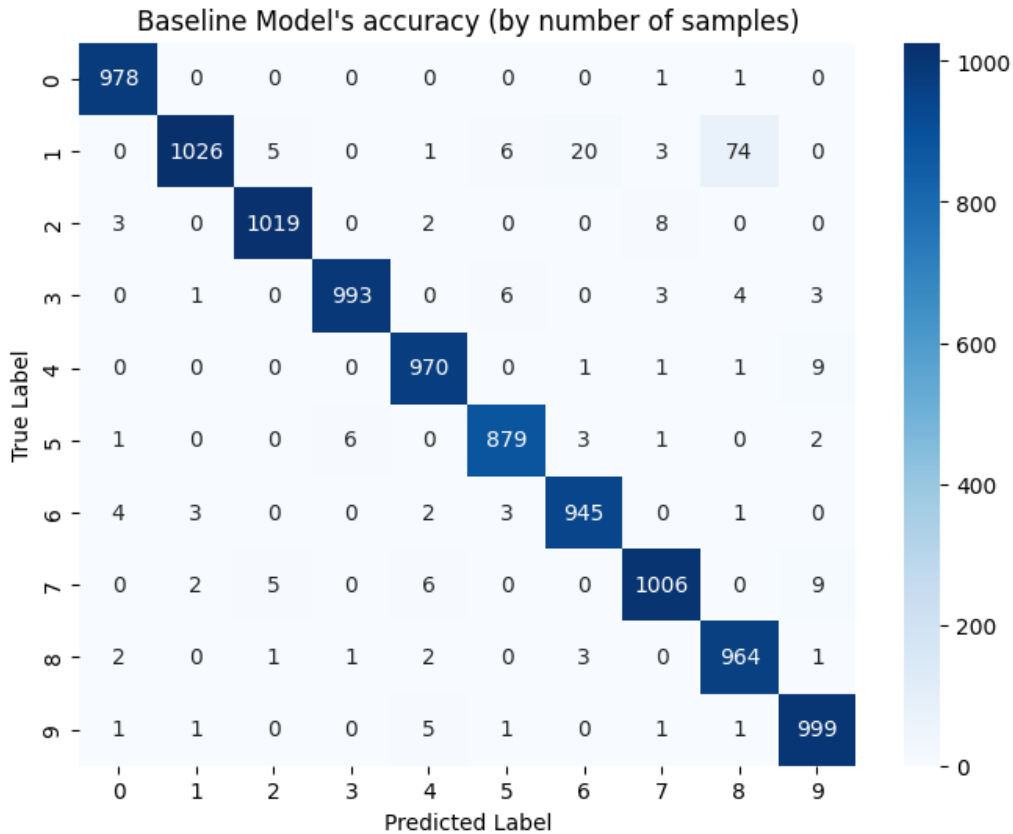


Figure 30: Baseline Results

Calling *evaluate\_model\_accuracy* on the trained baseline model and the MNIST test dataset, returned a total accuracy of: 97.79% and plotted the provided confusion matrix.

As explained under 1.1 Methodology and Rationale, the classifier, i.e. the baseline model we used is expected to have an accuracy of up to 98.7% on the MNIST test set, which is indeed the test set we used. Hence, this result is not surprising.

The confusion matrix indicates that label 1 was the most misclassified, while label 0 was classified almost perfectly. This trend is based on a single evaluation, therefore it should be interpreted cautiously.

## OOD Results

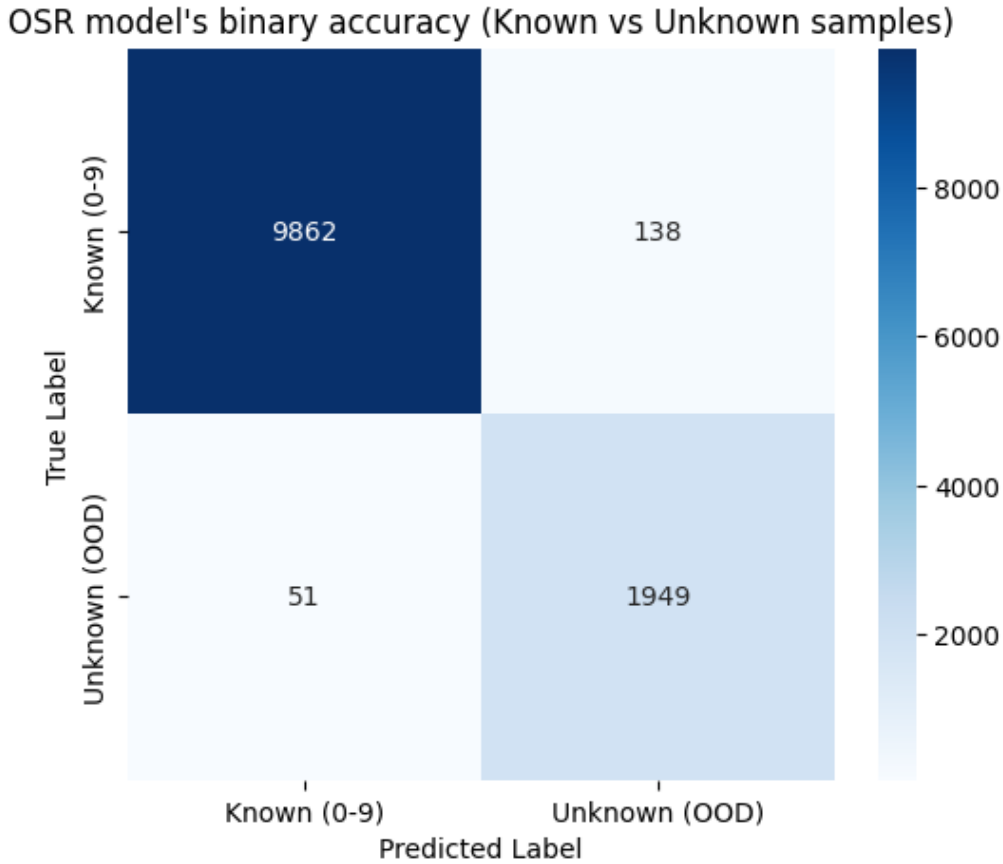


Figure 31: OOD Results

Calling *eval\_binary\_classification* on the trained OSR model and the combined dataset, returned a total "binary" accuracy of: 98.42% and plotted the provided confusion matrix.

This accuracy score represents the performance of the binary classification, in the sense of classifying samples from the combined dataset (MNIST + CIFAR10/FashionMNIST) as either from MNIST (class 0-9) or from the unknown class (class 10).

The confusion matrix demonstrates that 1.38% ( $[138 \times 100] / 10,000$ ) of the MNIST samples were misclassified as OOD, and 2.55% ( $[51 \times 100] / 2,000$ ) of the OOD samples were misclassified as MNIST, meaning that the model overall performed similarly on MNIST and OOD, when evaluated under this binary setting.

## OSR Results

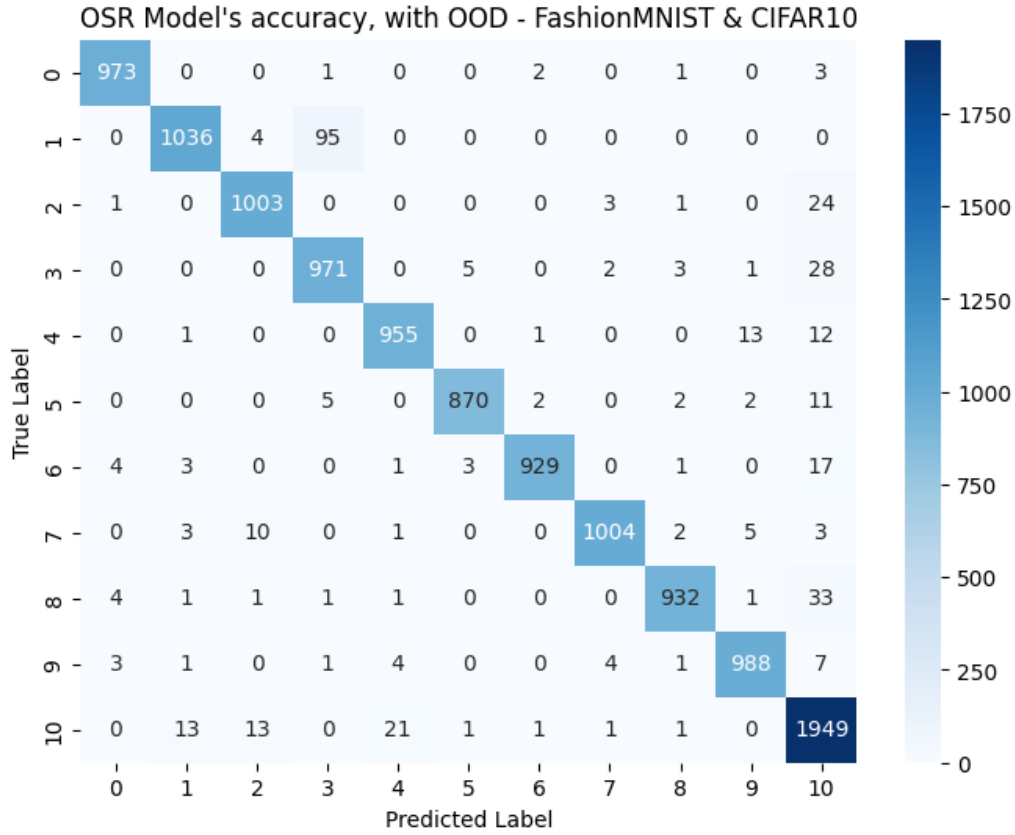


Figure 32: OSR Results

Calling `plot_confusion_matrix` on the predictions returned from the function `eval_model`, which in turn was called using the trained OSR model and the combined dataset, yielded the following accuracy scores:

Accuracy on MNIST: 96.61%

Accuracy on OOD: 97.45%

Total Accuracy: 96.75%

And the provided confusion matrix.

Besides the accuracy scores being high, the confusion matrix reveals additional information regarding the OSR model's performance: like the baseline model, the OSR model (comparatively) struggled with label 1. However, it misclassified it primarily as 3, whereas the baseline model misclassified it primarily as 8. Hence, the OSR model's struggle on samples labeled 1 could be in part due to the integration of the baseline model into the OSR model, and likely due to an inherent difficulty of reconstructing samples labeled 1.

The OSR model's MNIST accuracy (96.61%) was only slightly lower than the baseline's (97.79%), suggesting low performance trade-offs.

## Reconstruction Visualization


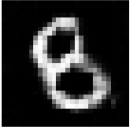



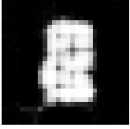



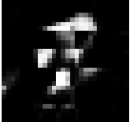


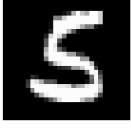
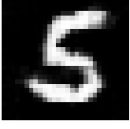

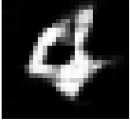

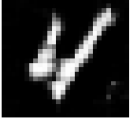
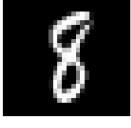
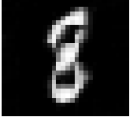
Original	Reconstruction	Prediction	True Label
		8	8
Original	Reconstruction	Prediction	True Label
		9	9
Original	Reconstruction	Prediction	True Label
		10	10
Original	Reconstruction	Prediction	True Label
		10	10
Original	Reconstruction	Prediction	True Label
		10	10
Original	Reconstruction	Prediction	True Label
		4	4
Original	Reconstruction	Prediction	True Label
		5	5
Original	Reconstruction	Prediction	True Label
		10	4
Original	Reconstruction	Prediction	True Label
		4	4
Original	Reconstruction	Prediction	True Label
		8	8

Figure 33: Reconstruction Visualization

Calling *evaluate\_and\_visualize* on the combined test dataset, and the trained OSR model, showed 10 samples from the combined dataset, as their original version (after transformation), their reconstructed version as per the OSR model, the prediction of the OSR model, and the true label of the sample.

The goal of this additional visualization is to emphasize the desired behavior of the model - if the sample isn't from the distribution it trained on (i.e. the train set which is taken from the distribution) it should perform the reconstruction poorly (as can be observed for rows 3-5), and otherwise the reconstruction should succeed (all other rows in the visualization), as with any AE model that is properly trained. This is desired since the crucial part that makes our model an OSR model is comparing the reconstruction loss to a threshold we calculate during training.

It is worth mentioning that in the third to last row, the sample from the MNIST test set was visually poorly reconstructed, and empirically mistakenly predicted to be 10 (OOD), which could also emphasize when the model is mistaken on MNIST samples.

### t-SNE Visualization

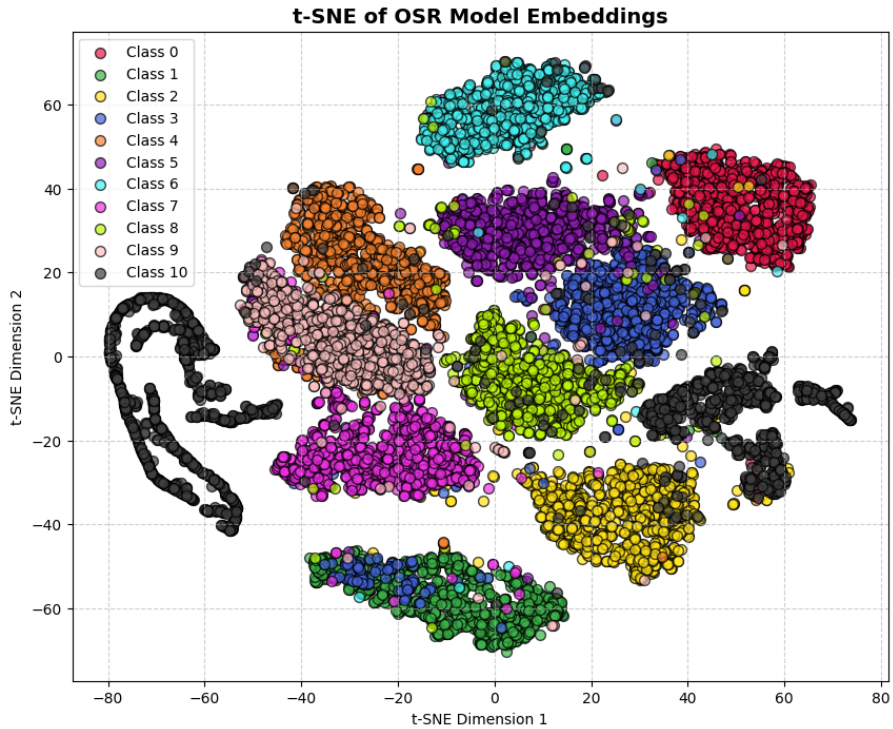


Figure 34: t-SNE Visualization

We first called *extract\_embeddings\_and\_predictions* on the OSR model and the combined test set, which extracts the embeddings and predicted labels from the trained OSR model, and then called the auxiliary function *tsne\_plot* using those embeddings and labels, to yield a scatter plot of the embeddings using t-SNE, which visualizes the model's learned embeddings in a 2D space.

It can be observed that the 10 MNIST classes formed relatively distinct clusters, while class 1 and class 3 had a more considerable overlap.

An interesting clustering was formed for class 10, as it had two clusters, which were also very far apart. This likely reflects the fact that our OOD is comprised of half CIFAR10 and half FashionMNIST, hence the OOD samples were clustered to these separate clusters.

Overall, this visualization further supports the model's ability to separate known and unknown classes effectively.

## 1.4 Limitations and Performance Analysis

Although our final model and approach seemingly effectively tackle the challenges within the OSR framework, it is essential to note that certain limitations remain.

- The classifier model, which is solely responsible for classifying between different MNIST classes, seems to be only capable of reaching up to around 98.9%. This is a hard limitation on our model’s capability for correct classification; Even if it managed to completely classify all OOD images as Unknown in a certain dataset, it would still not achieve perfect predictions over the ID samples.
- The AE model is both shallow and small, with a restrictive bottleneck in the form of a small embedded space size, which means that it fails to capture some of the more fine features of MNIST images.

This sometimes leads to reconstruction losses on certain outlier ID images that are well above the mean, causing them to be classified as Unknown. As our threshold is calculated in a manner that is completely independent of the dataset used as OOD, this issue will repeat itself regardless of the chosen OOD dataset.

- As a continuation of the above point, due to the AE model’s inability to capture fine details, should we use a dataset that is visually similar to the MNIST dataset as OOD, such as EMNIST, we will end up with similar reconstruction losses, and quite a few OOD samples will end up slipping below the threshold.

For example, it would obviously struggle to differentiate between 'I' from EMNIST and '1' from MNIST, and would likely reconstruct 'I' at about the same level of fidelity as '1'. Similarly, if it were to be tasked with something like distinguishing between clean MNIST images and noisy MNIST images, it would likely struggle as well since the captured features (Diagonal lines, loops, and so on) will be very similar between the ID and OOD samples.

In spite of these limitations, it is important to note that the model does perform fairly well on tasks with OOD datasets that are visually distinct from MNIST; A good example for this would be the SVHN dataset, where we’ve shown under the fifth attempt in this paper, that the AE completely fails at reconstructing these images, despite them containing actual numbers (And often only single digits). And overall, in our empiric observations, the model performs at a similar level with OOD datasets such as FashionMNIST and CIFAR10.