# CS462: HW2 - Features, Unsupervised Learning, Transfer Learning

Yash Chennawar | yc1376@scarletmail.rutgers.edu

November 18, 2025

## 1 Auto-Encoders

**Problem 1.**

An Auto-Encoder was trained on MNIST data in which the high dimensional input data was mapped down into a bottleneck with the Encoder, and then mapped back up to the input dimensions with the Decoder. The architecture used in the encoder was an input dimension of 784 nodes, then a hidden layer of 1000 nodes, and finally a bottleneck of $k$ nodes. The decoder started with that same bottleneck of $k$ nodes, then a hidden layer of 1000 nodes, and finally an output dimension equal to the size of the input dimension of 784 nodes. Different values of $k$ were tested, ranging from 1 to 700, and the results are plotted in Figure 1.

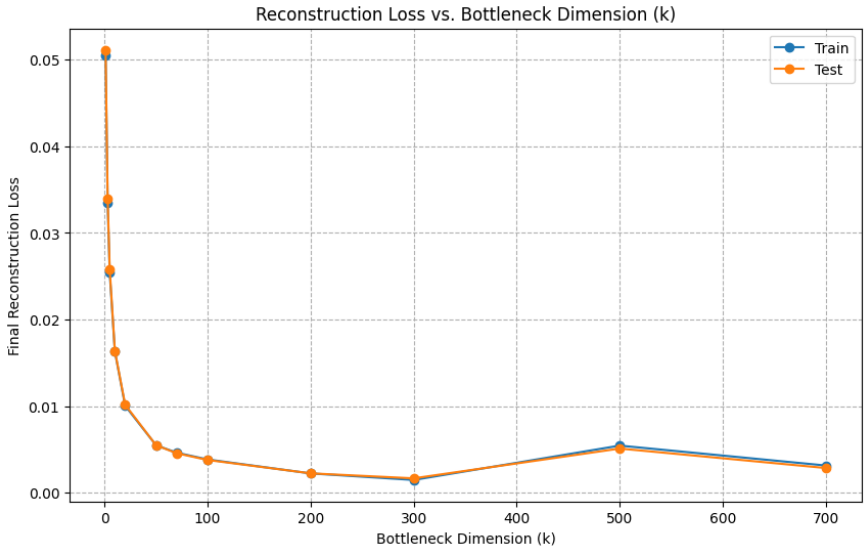



Figure 1: Reconstruction Loss vs k

This figure shows how the reconstruction loss changes as the bottleneck dimensions $k$ increases. Increasing $k = 1$ to 50 gave huge improvements. This is because for low $k$ values (1-10), the bottleneck is too narrow to compress 784 input dimensions and capture all the

intricacies of the dataset. It could not capture all the features of each digit so, the decoder could not reconstruct the image and thus, reconstruction loss was very high. Each additional latent dimension lets the model preserve more of the intricacies needed to reconstruct digits so, the loss decreases.

But then as $k$ increased from 50 to roughly 300, the loss continued to decrease but less rapidly. By $k = 300$, the encoder is able to represent the majority of MNIST variability in a few hundred latent dimensions. This suggests that the MNIST digits have an intrinsic dimensionality of roughly 300 dimensions, which is much less than the input size of 784. Even though they start off in so many dimensions, they can effectively be represented substantially fewer dimensions. This is true because the reconstruction loss is minimized here. Therefore, MNIST has roughly $k = 300$ dimensions plus some noise.

For higher $k$ values of more than 300, the reconstruction loss stops decreasing and shows signs of small increases and fluctuations. As the bottleneck dimension grows, the model will stop trying to learn a compact representation which could be generalized; it will try to fit all the dimensions from the training data into the latent space. A large-capacity auto-encoder is essentially approximating the identity function and does not generalize well for new data.

Now with this optimal $k = 300$, I trained and tested an auto-encoder with the same architecture. Figure 2 displays the training and testing dynamics, when trained for 10 epochs and learning rate 0.001. The model used the Adam optimizer and Mean Squared Error loss for the reconstruction loss. This model achieved a final train loss of 0.00151 and final test loss of 0.00146, showing that this auto-encoder architecture was able to encode the MNIST data into a latent space successfully.

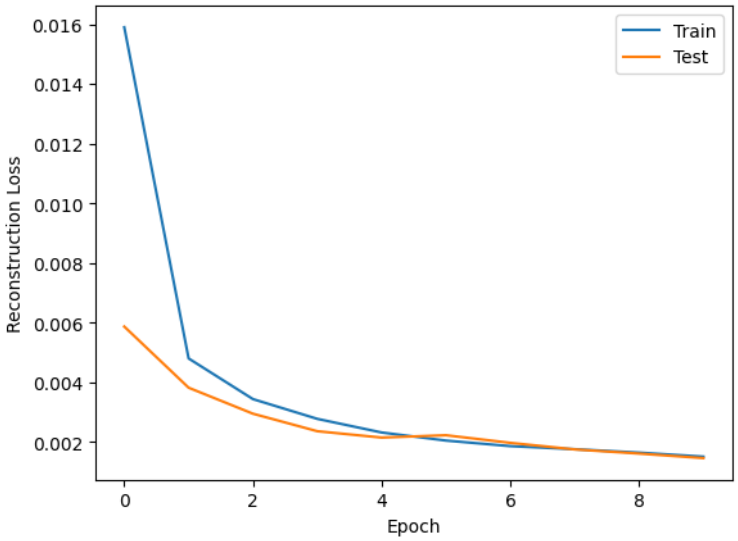



Figure 2: Loss vs Epochs for Auto-Encoder with $k = 300$

To visualize the reconstruction quality, I plotted the original and reconstructed images for the $k = 300$ auto-encoder network. This is shown in Figure 3. The images have been reconstructed very well, meaning that the network performs well.

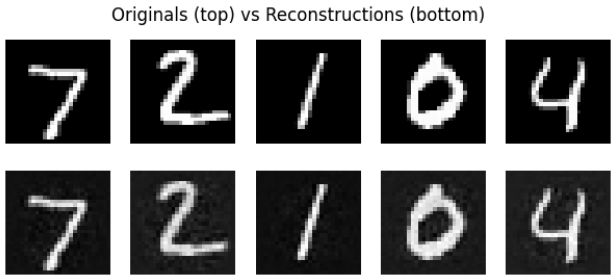



Figure 3: Reconstructions For Auto-Encoder with $k = 300$

Principle Component Analysis (PCA) was also conducted on the MNIST data. It is a linear model that shows how much of the data variance can be captured in fewer dimensions. This analysis shows that 95% of the variance can be captured with around 150 components, and 99% with around 330 components, as shown in Figure 4. PCA also suggests that the MNIST dataset exists in a lower dimensional latent space. The auto-encoder model found the lowest reconstruction loss with a few hundred bottleneck dimensions in the latent space, and this finding aligns with PCA. The auto-encoder was successfully able to nonlinearly learn a representation of MNIST, similar to what was found linearly with PCA.

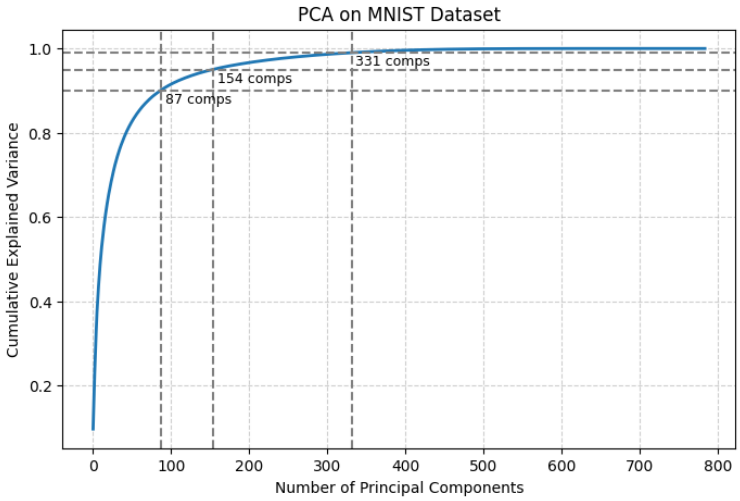



Figure 4: PCA on MNIST data

The next experiment tested whether dropout layers help an auto-encoder. A dropout layer with dropout probability $p$ was added right after the hidden layer and activation function for both the encoder and decoder. The results of the experiment are summarized in Table 1, in which $k$ and $p$ were changed to see the effect on reconstruction loss.

| **Bottleneck Size ($k$)** | **Dropout Probability ($p$)** | **Final Train Loss** | **Final Test Loss** |
|---|---|---|---|
| 300 | 0.0 | 0.00151 | 0.00146 |
| 300 | 0.1 | 0.00622 | 0.00421 |
| 300 | 0.3 | 0.01174 | 0.00760 |
| 500 | 0.1 | 0.00622 | 0.00418 |
| 500 | 0.3 | 0.01194 | 0.00785 |

Table 1: Dropout on Autoencoder Reconstruction Loss for Bottleneck Dimensions ($k$)

The lowest reconstruction loss was seen with no dropout ($k = 300, p = 0.0$). This means that the auto-encoder was able to compress and reconstruct the MNIST digits without any overfitting, as the difference between train and test loss is negligible. Adding dropout resulted in increased train and test loss for both $k = 300$ and $k = 500$. For a small dropout ($p = 0.1$), the loss rose by a factor of 4, and for a slightly larger dropout ($p = 0.3$), the loss increased significantly. This occurred for both sizes of the bottleneck dimension $k$. An example loss curve for the $k = 300$ and $p = 0.1$ configuration is shown in Figure 5, and the other 3 dropout networks follow a very similar pattern.

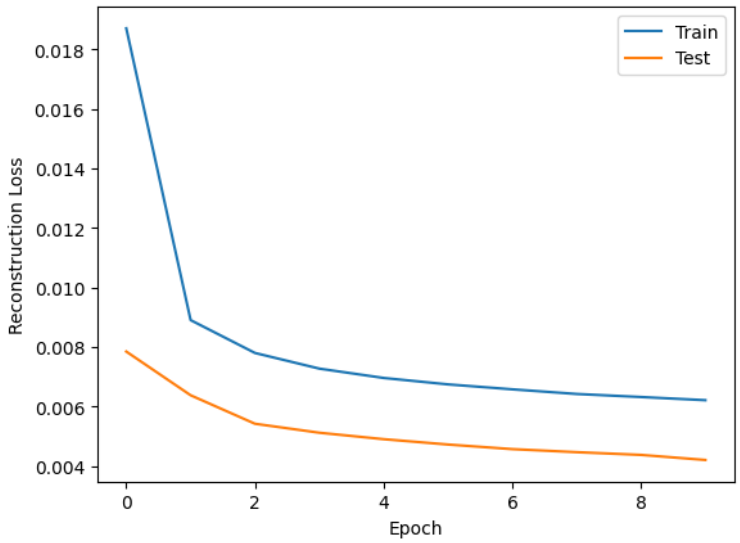



Figure 5: Reconstruction Loss for $k = 300$ and Dropout $p = 0.1$

The performance decreases with dropout regularization because it reduces the model's capacity by randomly zeroing some activations during training. Dropout often helps when the model is overfitting but the auto-encoder in this case has very minimal overfitting, regardless of the value of $k$, as seen earlier in Figure 1. Dropout is removing important representational information, thus causing higher reconstruction loss. Applying dropout on a model that does not overfit would only hurt the reconstruction. Therefore, for this particular auto-encoder architecture, dropout is not helpful.

**Problem 2.**

For the optimal $k = 300$ from Problem **1**, the trained and untrained encoders can be used for a transfer learning and classification task. $F_{\text{trained}}$ is the optimal trained encoder and $F_{\text{untrained}}$ is an encoder of same architecture but random weights. The following classification model can be made with Cross Entropy Loss and the Adam optimizer.

$$\text{Classify}(\vec{x}) = \texttt{softmax}(AF(\vec{x}) + \vec{b})$$

This classifier was then tested in 4 different cases:

1. Randomly initialized encoder, training full Classify($\vec{x}$) model.
2. Pretrained encoder from before, training only weights in the classifier's linear layer.
3. Randomly initialized encoder, training only weights in the classifier's linear layer.
4. Pretrained encoder from before, training fullClassify($\vec{x}$) model.

The results of these cases are summarized in Table 2 and Figure 6, with final train and test loss values and curves. Everything was trained for 10 epochs with a learning rate of 0.001.

| Case | Encoder Type | Trainable Layers | Train Loss | Test Loss |
|---|---|---|---|---|
| 1 | Random (Untrained) | Full (Encoder + Classifier) | 0.0178 | 0.1180 |
| 2 | Pretrained (Frozen) | Classifier Only | 0.2892 | 0.2806 |
| 3 | Random (Frozen) | Classifier Only | 0.4140 | 0.3793 |
| 4 | Pretrained (Fine-tuned) | Full (Encoder + Classifier) | 0.2893 | 0.2805 |

Table 2: Comparison of Encoder Initialization and Training Strategy for Classification

Case 1, with a randomly initialized encoder and full training on the encoder and classifier, performed the best with a train loss of 0.018 and test loss of 0.118. This is basically a feedforward neural network learning everything from scratch as all the weights start random and every layer is being trained. The gap between train and test loss is relatively high so there was slight overfitting, which could be handled with regularization. There is a high training cost since the whole network needs to be trained.

Case 2, with a pretrained encoder and training only on the classifier layer, performed with train loss of 0.289 and test loss of 0.281. This performed better than a random and frozen encoder, which shows that the pretrained encoder contained important features of MNIST digits, which were good enough to be used without fine-tuning for classification. The train and test losses are also almost identical so, there is no overfitting. There is a very low training cost since most of the model was already trained. This means that learning was successfully transferred.

Case 3, with a randomly initialized encoder and training only on the classifier layer, performed the worst with a train loss of 0.414 and test loss of 0.379. The encoder is random and frozen so, the classifier worked with useless features, essentially random noise. There need


(a) Case 1: Random encoder, full training


(b) Case 2: Pretrained encoder, frozen


(c) Case 3: Random encoder, frozen


(d) Case 4: Pretrained encoder, full training

Figure 6: Training and testing loss curves for all four classification cases.

to be meaningful representations for such a model to perform well so this case performed poorly, as expected.

Case 4, with a pretrained encoder and full training on the encoder and classifier, performed with train loss of 0.289 and test loss of 0.281. This is interesting because the results are almost equal to those of Case 2. This means that the pretrained encoder was already almost optimal so the extra training (fine-tuning) had minimal impact on it. It could have possibly improved with more epochs for there to be more impact from the fine-tuning. Again like Case 2, there was no overfitting but there was a high training cost, like Case 1.

# 2 Transfer Learning

**Problem 3.**

Now we have to train a network for the CIFAR-10 classification task. The following architecture was used. It used an Adam optimizer with 0.001 learning and Cross Entropy Loss, for 22 epochs, with some weight decay.

1. **Conv Block 1**: Conv($3 \to 32$) $\to$ ReLU $\to$ MaxPool $\to$ Dropout($p = 0.25$)
2. **Conv Block 2**: Conv($32 \to 64$) $\to$ ReLU $\to$ MaxPool $\to$ Dropout($p = 0.25$)
3. **Conv Block 3**: Conv($64 \to 128$) $\to$ ReLU $\to$ MaxPool $\to$ Dropout($p = 0.25$)
4. **Flatten**: Input $= 4 \times 4 \times 128 = 2048$
5. **Fully Connected (Hidden)**: Linear($2048 \to 256$) $\to$ ReLU $\to$ Dropout($p = 0.25$)
6. **Output Layer**: Linear($256 \to 10$)

Table 3 and Figure 7 below show the loss and accuracy metrics, along with loss curves.

| **Metric** | **Value** |
|---|---|
| Training Loss | 0.7040 |
| Testing Loss | 0.7010 |
| Test Accuracy | 75.56% |

Table 3: Performance of CIFAR-10 Classification Model

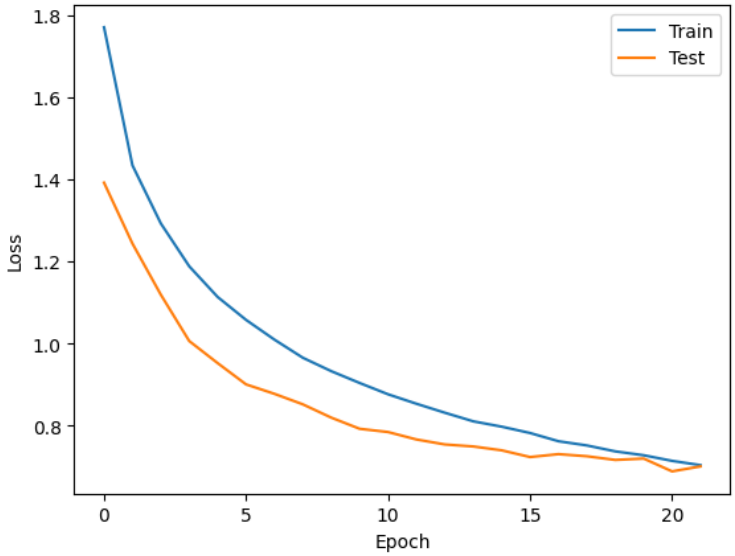



Figure 7: Loss Curves for CIFAR-10 Classification Model

Both of these figures show the performance of the CNN model on CIFAR-10 data. With 75.56% accuracy, the model had moderate performance. But seeing that the train and test loss curves have converged, the model does not overfit and generalizes well. After trying different architectures and regularization techniques, this was the best configuration found.

The loss for this, and most of the other CIFAR-10 tasks did not reduce more because of the complexity of the task compared to MNIST. This dataset has much more complex images so this architecture may be too simple.

**Problem 4.**

Now for the dataset where some CIFAR data points are rotated by 90°, a very similar architecture from before was used. All the convolutional and hidden layers remained the same, except for the output layer having 2 nodes. This is essentially a binary classification task choosing between images that are rotated or not. Table 4 and Figure 8 represent the training dynamics.

| Metric | Value |
|---|---|
| Training Loss | 0.3134 |
| Testing Loss | 0.2914 |
| Test Accuracy | 87.14% |

Table 4: Performance of Rotated CIFAR-10 Classification Model

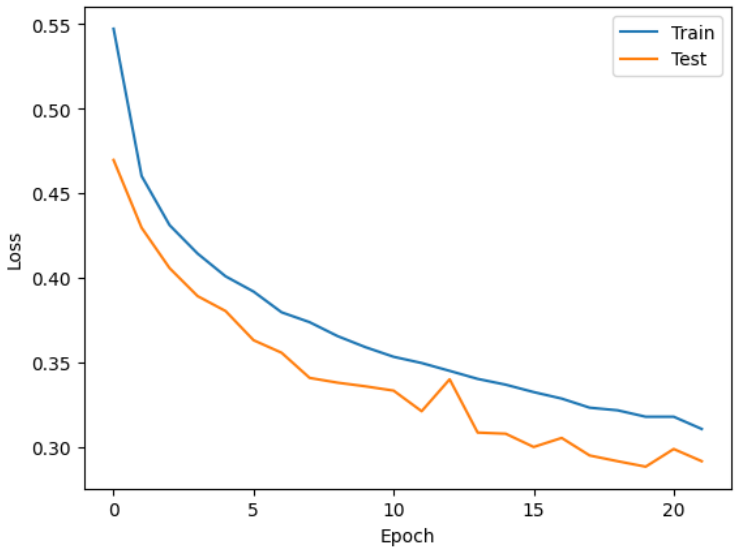



Figure 8: Loss Curves for Rotated CIFAR-10 Classification Model

The pretrained model from the last problem was able to achieve 86.97% accuracy. There is a very small difference between train and test loss so, there is minimal overfitting. This high accuracy suggests that in the last task, the model's convolutional layers learned features regarding orientation. This kind of task involves edge detection and spatial arrangement which are encoded into the convolutional layers. This accuracy may be higher than the 10-class classification task from before because it is a much simpler task.

**Problem 5.**

Now to test transfer learning, I initialized a model of same architecture as problem 3, and pretrained it using weights from the rotation classification model in problem 4. The first $k$ convolutional blocks (each with a Conv2d, ReLU, MaxPool, and Dropout) were transferred from the pretrained model. Figure 9 shows the results of choosing the first 1, 2, or 3 convolutional blocks from the pretrained network and using them for the 10-class classification task. I also compared the results for the first $k$ convolutional blocks while freezing them or leaving them non-frozen and able to train.

The rest of the network remained the same, with same hyperparameters and training parameters as before. The hidden and output layers were re-initialized and trained. This kept the experiment consistent while changing $k$ for a fair comparison.

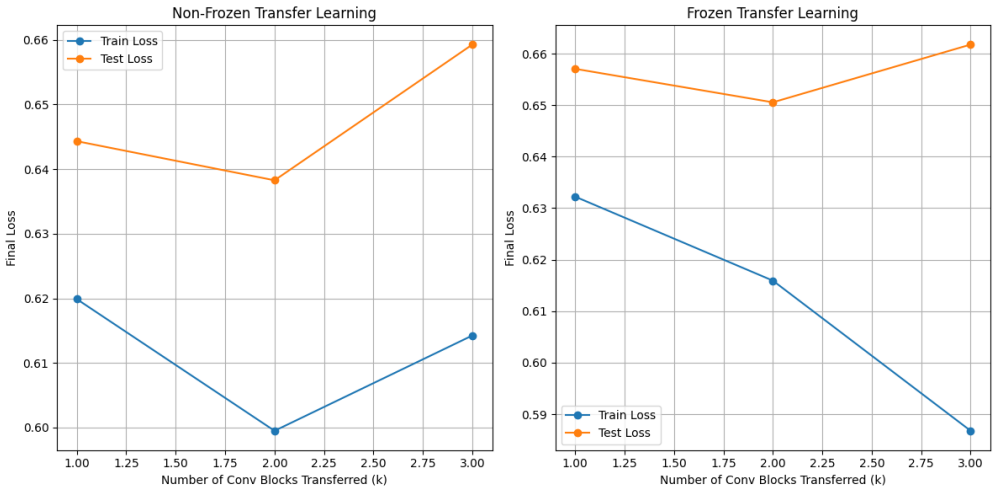



Figure 9: Transfer Learning Loss by $k$ Blocks

As seen in Figure 9, the lowest train and test loss were found with $k = 2$, with the non-frozen layers. This meant allowing the first two convolutional blocks to train their pretrained weights. So, the rest of the layers were fine-tuned to achieve the best loss, based on the pretrained layers. The model was able to both generalize and adapt to the new task. This configuration performed better than the fully frozen and fully trained networks.

Table 5 and Figure 10 both summarize the results and training dynamics for $k = 2$ non-frozen layers transferred. Compared to the baseline model from Problem 3, summarized in Table 3, this transfer learning model has much better loss and a slight increase in accuracy. This implies that the pretrained rotation model learned some low-level features like edges and textures, which was effectively generalized to the CIFAR-10 classification task.

Since a partial weight transfer had the best results, it suggests that early convolutional layers learn general features, while later layers learn very specific features. Freezing the transferred layers increased loss as there was less adaptation to the new task. Therefore, fine-tuning let the network adjust its weights to work well with pretrained weights.

| Metric | Value |
|---|---|
| Training Loss | 0.5927 |
| Testing Loss | 0.6650 |
| Test Accuracy | 77.15% |

Table 5: Performance of CIFAR-10 Transfer Network with Non-Frozen $k = 2$

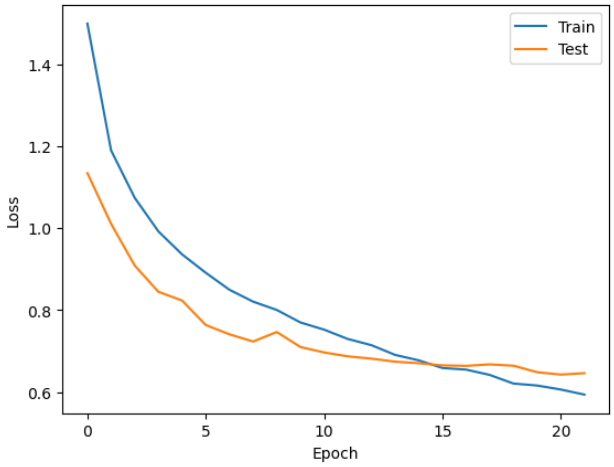



Figure 10: Loss Curves for Transfer Network with Non-Frozen $k = 2$

**Problem 6.**

The contrastive learning task required a slightly different approach than the other networks. I first initialized an embedding network using the same structure as the previous convolutional neural network, with convolutional blocks and fully connected layers. The main difference was that the output layer had 2 nodes, since it is mapping CIFAR-10 image data points into 2-dimensions.

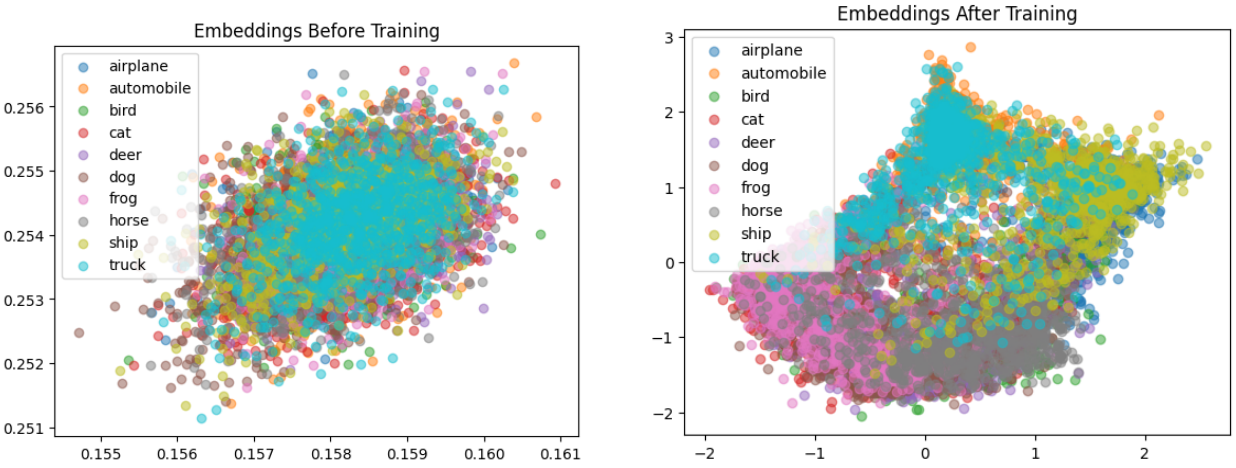
Training depended on the formula given in the assignment. The custom contrastive loss function is designed to pull embeddings of similar data points closer together and push dissimilar data points further, up to a margin $M$. The loss function works by taking the sum of the loss of similar pairs of items and the sum of dissimilar pairs of items.

$$\begin{aligned}
\text{SimilarLoss}(\vec{x}, \vec{x'}) &= S(\vec{x}, \vec{x'}) \cdot ||E(\vec{x}) - E(\vec{x'})||^2 \\
\text{DissimilarLoss}(\vec{x}, \vec{x'}) &= (1 - S(\vec{x}, \vec{x'})) \cdot \max(M - ||E(\vec{x}) - E(\vec{x'})||^2, 0) \\
\text{ContrastiveLoss}(\vec{x}, \vec{x'}) &= \text{SimilarLoss}(\vec{x}, \vec{x'}) + \text{DissimilarLoss}(\vec{x}, \vec{x'})
\end{aligned}$$

The train function creates pairs of images by comparing a batch with itself but permuted randomly. It creates a similarity label $S(\vec{x}, \vec{x'})$: 0 if they are of different classes and 1 if they are the same. This is then computed with the Euclidean distance between the embeddings of the data point pairs. The DissimilarLoss() function penalizes the embedding if the distance is

less than the margin $M$ specified. This forces dissimilar items further apart but not infinitely apart.

Figure 11a shows the initial embedding on an untrained network. All the data points are overlapping and in one big cluster, randomly arranged. This is expected because of the random weights in the embedding network.




(a) Embedding before training

(b) Embedding after training

Figure 11: Comparison of embeddings before and after training.

The embedding network being used has 2 convolutional blocks of 32 channels and 10 channels, each with a ReLU, MaxPool and Dropout of 0.1 probability. The output of this then goes to fully connected layers of sizes 1000, 500, 100, and 10, with 2 output units. This was trained for 20 epochs with a learning rate of 0.001 and a margin $M = 2$. This architecture and hyperparameter configuration was achieved with many different tries to find the best accuracy. Its training dynamics are summarized in Table 6 and Figure 12.

| **Metric** | **Value** |
|---|---|
| Training Loss | 0.5373 |
| Testing Loss | 0.5145 |

Table 6: Performance of Embedding Network

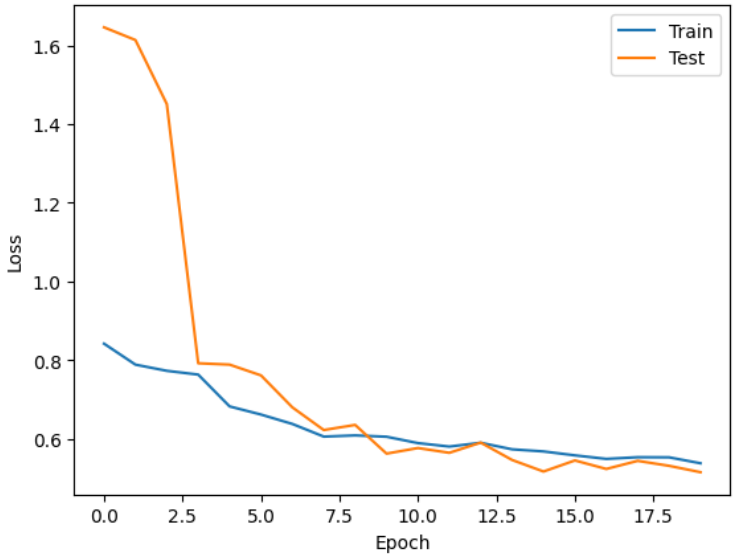



Figure 12: Embedding Network Train and Test Loss

Both train and test loss achieved low values close to 0.5. This shows that there was no overfitting, which in this context would mean minimizing contrastive loss for the specific random pairs trained on but not any unseen random pairs. Figure 11b shows the embeddings after the network trains. The different classes are shown to be separated throughout the latent space, in separate clusters. The points are spread across a much larger 2D space.

The embedding model also put similar items in similar places. For example, the truck and automobile clusters have some overlap and are very close together. Airplane and ship clusters appear close to these as well. On the other hand, all the animal clusters appear on the opposite side of the graph, and they all are relatively close together. The frog and horse clusters are separated, with cat, bird, and dog clusters in between them. The four-legged animals have many similar features so they may be visually ambiguous and appear close by, while the margin $M = 2$ forced the vehicle images further away.