Deep Learning 2025
Assignment 2

Simon Hvidtfeldt — zgh166

October 29, 2026

1 Convolutional neural networks

1.1 Task 1:

I assume that we are only supposed to comment on the main program in `feedForwardMNIST.ipynb`, which starts from cell 6.

Cell 6 This cell defines a preprocessing pipeline `transform()` which is applied to the raw image data. It converts the image to a tensor `transforms.ToTensor()`, and then it normalizes the pixel values using 0.5 as mean and 0.5 as standard deviation `transforms.Normalize((0.5,), (0.5,))`.

Cell 7 This cell is responsible for loading the MNIST dataset, which has 60,000 images for training, and 10,000 for testing. It applies the `transform` to both sets, and creates data loaders which are `trainloader` and `testloader` for iterating through the data in batches of size 64.

Cell 8 Here the architecture of the feed-forward network with the `Net` class is defined, where it inherits from `nn.Module`. The model `self.main` has several layers.

> **Input:** The 28×28 image (784 features) is fed into the sequence.
>
> **Hidden layer 1:** Fully connected layer, with 784 inputs to 128 outputs, followed by a `ReLU` activation function.
>
> **Hidden layer 2:** Fully connected layer with 128 inputs to 64 outputs, followed by a `ReLU` activation function.
>
> **Output layer:** Fully connected layer with 64 inputs and 10 outputs, with corresponds to the 10 classes. This is followed by a `LogSoftmax` activation function.

This results in a total parameter count of:

$$
\begin{aligned}
&(784 \cdot 128) + 128 + (128 \cdot 64) + 64 + (64 \cdot 10) + 10 = \\
&100,352 + 128 + 8,192 + 64 + 640 + 10 = \\
&109,386
\end{aligned}
$$

Cell 9 Lastly, cell 9 is responsible for setting up the training hyper-parameters, and starting the training. Training is set for 4 epochs, with a learning rate set to 0.002. It uses negative log likelihood loss (`nn.NLLLoss`), as a loss function, and stochastic gradient descent (`optim.SGD`) as an optimizer, with momentum set to 0.9. The cell then executes the `training_loop`, which is defined before the main program, on the 60,000 training images, and then calls `evaluate_model`, which is another function defined before the main program, on the 10,000 test images to report the overall accuracy, as well as the accuracy per class. After training for 4 epochs on the 60,000 images, the model managed to achieve an overall accuracy of 96% on the 10,000 test, and for each class is got the following accuracy:

Class	Accuracy
0	0.992857
1	0.985903
2	0.967054
3	0.965347
4	0.949084
5	0.941704
6	0.968685
7	0.965953
8	0.929158
9	0.930624

Table 1: Accuracy across all classes

1.2 Task 2:

In our feedforward network we are currently using `nn.LogSoftmax(dim=1)`, which outputs the logarithmic probabilities. Additionally, the loss function used together with `LogSoftmax` is the negative Log likelihood loss `nn.NLLLoss()`. If we remove line (5), which is the `nn.LogSoftmax(dim=1)` layer, the final output layer `nn.Linear(in_features=64, out_features=10)` will now output logits. When working with logits we should instead use Pytorch's loss function `nn.CrossEntropyLoss()` which works by applying the Softmax operation to the logits to get probabilities, and the calculating the negative log-likelihood of those probabilities.

It is generally a good idea to work with log softmax/logits output rather than softmax output, because it gives better numerical stability. Using the softmax function involves the exponential function. If the logits get too large, taking the exponent of these logits can lead too overflow. On the other hand, if the logits are vaery large negative numbers, they can potentially evaluate to zero. If this happens, when we then take the logarithm in the loss calculation, we get NaN, which will result in an error.

1.3 Task 3:

Our convolutional layers work on a 4 dimensional tensor, which consists of Batch size $\times$ Channels $\times$ height $\times$ Width. A standard linear layer expects input data to be 2 dimensional, and consist of Batch size $\times$ feature. The `nn.flatten()` operation converts the multi dimensional feature map into a single long vector of features. This keeps the batch dimension intact and prepares the data for the linear classifier.

The number of input features, in this case 800 is determined by the size of the feature map. We start by inputting a MNIST image, which has the input size of 28×28, output size of 28×28, and output size 1. Then `nn.Conv2d(1, 16, kernel_size=(3,3), stride=(1,1))` is applied. We still have an input size of 28×28, but the output size is now 26×26, and the output channels is 16. `nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)` now halves our dimensions, so with an input size of 26×26, we get an output size of 13×13, and still 16 output channels. `nn.Conv2d(16, 32, kernel_size=(3,3), stride=(1,1))` transforms the input size from 13×13 too 11×11 output size, and the output channels are 32. `nn.MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)`, halves our dimensions again, but rounded down, so we are left with an output size of 5×5, and 32 output channels. The tensor the is used with `nn.Flatten()` has a shape of 32 channels by 5 height and 5 width, so the total features are $32 \cdot 5 \cdot 5 = 800$.

The convolutional networks accuracy of the network on the 10,000 test images are 98%.

Feedforward network		**Convolutional network**		**Best Accuracy**
Class	Accuracy	Class	Accuracy	
0	0.992857	0	0.993878	Convolutional network
1	0.985903	1	0.992070	Convolutional network
2	0.967054	2	0.975775	Convolutional network
3	0.965347	3	0.989109	Convolutional network
4	0.949084	4	0.985743	Convolutional network
5	0.941704	5	0.984305	Convolutional network
6	0.968685	6	0.972860	Convolutional network
7	0.965953	7	0.968872	Convolutional network
8	0.929158	8	0.980493	Convolutional network
9	0.930624	9	0.965312	Convolutional network

Table 2: Feedforward network versus Convolutional network

Comparing the two networks, we can see that the convolutional network outperformed the feedforward network on every class.

In addition the performing better than the feedforward network, the convolutional network also uses less parameters. Layer (0) `nn.Conv2d(1, 16, kernel_size=(3,3), stride=(1,1))` Results in 160 parameters since $(1 \cdot 3 \cdot 3) \cdot 16 + 16 = 144 + 16 = 160$. Layer (3) `nn.Conv2d(16, 32, kernel_size=(3,3), stride=(1,1))` gives us $4,640$ parameters since $(16 \cdot 3 \cdot 3) \cdot 32 + 32 = 4,608 + 32 = 4640$. lastly, layer (8) `nn.Linear(in_features=800, out_features=10, bias=True)` gives us $8,010$ parameters, because $800 \cdot 10 + 10 = 8,010$. This results in a total parameter count of

$$160 + 4,640 + 8,010 = 12,810$$

This means that the convolutional network

$$\left(\frac{12810}{109386}\right) \cdot 100 \approx 11,7$$

uses only about 11.7% of the parameters that is used by the feedforward network.

2 Autoencoders

2.1 Task 1:

Cell 1 & 2: The first two cells are responsible for importing the libraries used, as well as checking for an availible GPU, and then setting it to either `cuda` or `cpu`.

Cell 3: Here we define a block named `FF`, which consists of two linear layers, with a `Relu` activation function applied after the first layer.

Cell 4: The next cell defines the `Autoencoder` model class. It creates the network using the `FF` class for both the `self.encoder` and the `self.decoder`.

The encoder uses the `FF` class, which consists of linear layers with `ReLU` activation in between, ending in a final linear layer `nn.Linear(dim2, dim3)`. No activation function is defined after this final layer. This means that the encoder uses a linear activation function i.e. the identity function.

The final activation function applied by the decoder is `nn.Sigmoid()`. This function maps values into the interval $(0, 1)$. This interval matches the input pixel values, which were normalized during the `transforms.ToTensor()` step. This ensures that the reconstructed pixels are between 0 and 1.

Cell 5: Cell 5 implements the function, which is used for training the model over multiple epochs. Additionally, it zeroes the gradient using `optimizer.zero_grad()`, it executes the forward propagation with `x_hat = model(x)`, it calculates the loss, and backpropagating errors with `loss.backward()`, and lastly updating the parameters with `optimizer.step()`

Cell 6: Here we define a way to visualize the learned latent space representations `z`, by scattering the points and coloring them by original class labels `y`.

Cell 7: In cell 7, we define a function to generate and show images created by the decoder based on a 2D grid. The decoder output is reshaped back into image dimensions, which are w$\times$h.

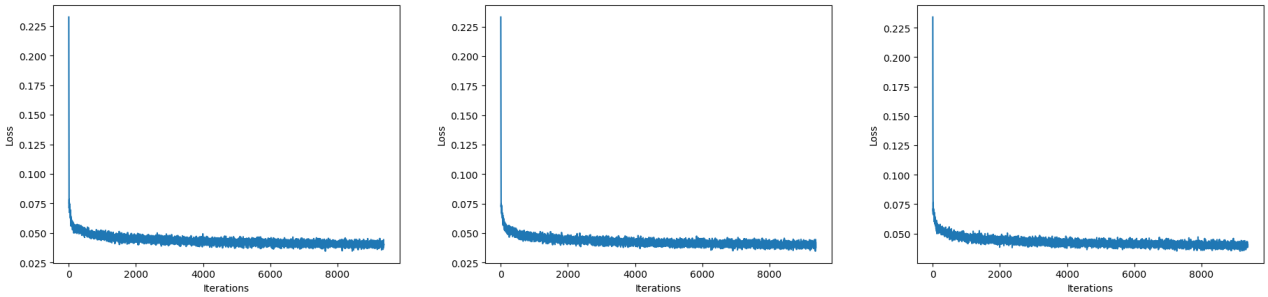
Cell 8: Now we have the main execution block. It defines `transform`, which loads the MNIST dataset, sets the hyperparameters, which are 28×28 inputs, 512 hidden units and 2 dimensions. It also selects the Adam optimizer, and the MSE loss, and begins a 20-epoch training process. Additionally, `transform` is responsible for data preprocessing steps to the input images. `transforms.ToTensor()` converts the input image data into Pytorch tensors and scaling the pixel values. `transforms.Lambda(lambda x: torch.flatten(x,start_dim=-2))` takes the image tensor and flattens its dimensions into a single vector. This converts the images from a 2D format into 1D format, which is necessary for the linear layers. In this cell we also define the loss function `torch.nn.MSELoss()`. Mean Squared Error Loss calculates the distance between the predicted output and the actual input, which is great for optimizing, since we want the output `x_hat` to regress toward the input.

Cell 9, 10 & 11: Executes the code to plot the losses over iterations, visualize the resulting latent space, and display reconstructed images from the latent space.

We could also use the beta distribution as likelihood function, since the problem of training the autoencoder can be viewed as minimizing the negative log-likelihood of the data given the model parameters, instead of minimizing the mean squared error loss.
Since the decoder outputs normalized pixel values between 0 and 1, the beta distribution could be used as a probability density function to use as a likelihood mode, as it is supported only on the interval (0,1).

2.2 Task 2:

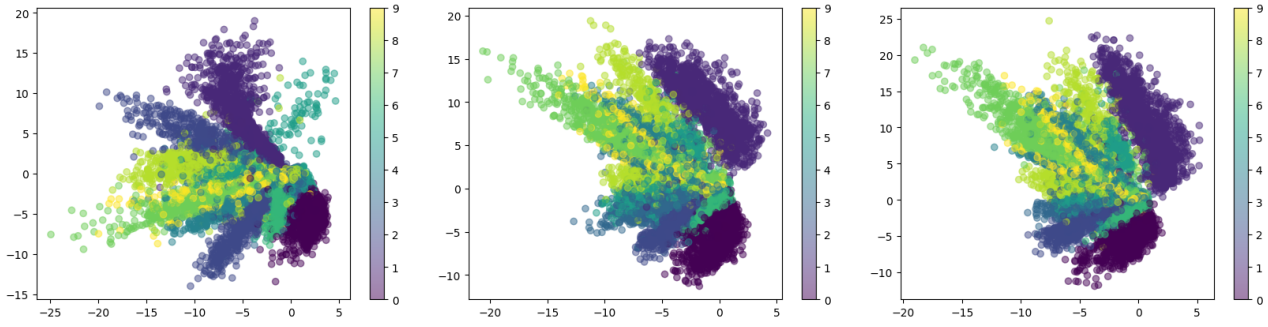
We ran `autoencoders.ipynb` at least 3 times, and below are the results visualized in Figures 1, 2, and 3.



(a) Run 1 (b) Run 2 (c) Run 3

Figure 1: Results of plotting the loss 3 times.

Comparison: As can be seen in figure 1 the loss curves across all the three runs are very similar. In every run the loss drops sharply after the first few epochs and converges to a low and stable value. This indicates the optimization process is not sensitive to random initialization when it comes to convergence speed or final error.



(a) Run 1 (b) Run 2 (c) Run 3

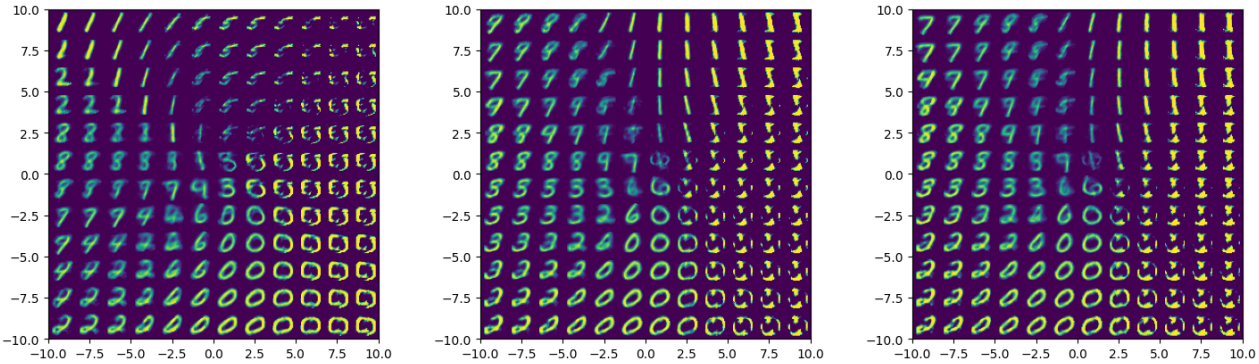
Figure 2: Results of plotting the latent space 3 times.

Comparison: In all three runs, the points are not randomly scattered. They form groups based on their digit labels. This indicates that the model learns features, and groups similar

looking digits to the same location. Additionally, if we compare the dark blue cluster (2-3) and the yellow cluster (8-9), we can see that for the first plot, the dark blue cluster goes to the top-left, and the yellow is somewhat central. For the second plot, the dark blue cluster now goes toward the top-right, and the yellow cluster has moved to the top left.

Plot 3 is very similar too plot 2, but the clouds are shaped slightly different. The reason we get different plots are that the network can freely rotate, mirror or scale the latent space, as long as the encoder knows how to interpret it. The coordinate (-10,10) might mean digit 1 in the first run, but could mean digit 7 in the second run.

Finally, even though the groups rotate, the neighbors stay the same. The green clusters are always between the dark blue and yellow clusters. This is due to the model learning that a '3' is similar looking to an '8' so they would be near each other in the latent space.



(a) Run 1 (b) Run 2 (c) Run 3

Figure 3: Result of generating examples from latent space 3 times.

Comparison: In all three images the digits are recognizable, but blurry, as a result of the mean squared error. We can also see that the digits change as we move across the grid, but the most significant difference is that the location of the digits change from run to run. Run 2 and 3, are very similar. Both appears to have 7-9 displayed at the top left, but run 1 clearly has 1-2 at this position.

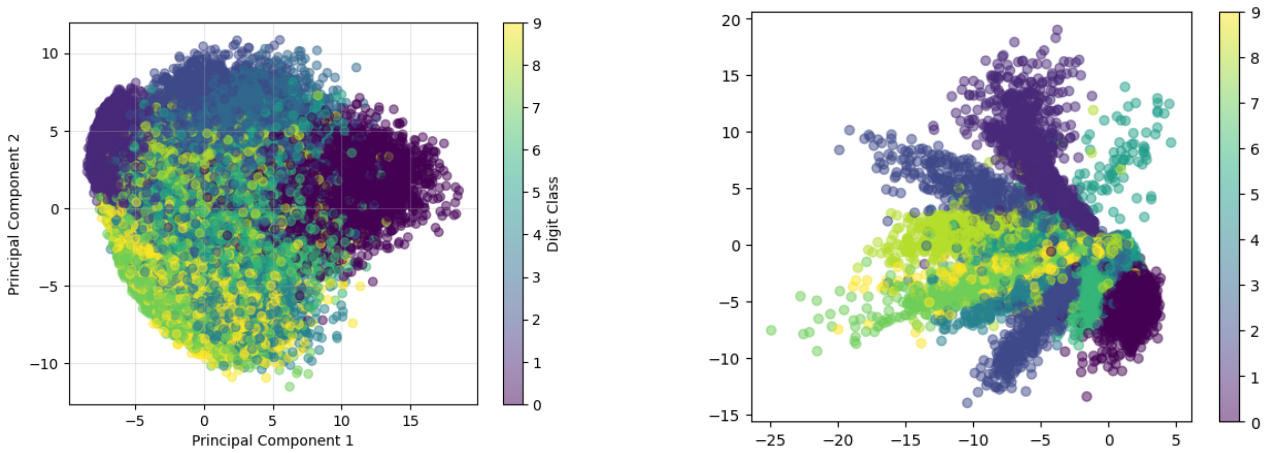
2.3 Task 3:

Both the autoencoder and PCA try to reduce the dimensionality of the data into a lower dimensional latent space, but they use different approaches. The autoencoder is highly non-linear. It uses multiple dense layers and a `ReLu` activations to learn the complex relationship. The PCA instead find principal components that maximize variance in the data.

The autoencoders training objective is to minimize reconstruction loss (MSE) between our input and output, where as the PCA's objective is to find orthogonal axes that capture maximum variance.

The autoencoder learns a nonlinear mapping from high-dimensional image space to the 2D latent space. This allows it to capture more complex structures from the data. The PCA as a linear estimation is limited to finding the straight line, that best explains the spread of the data. This difference in approach can also be seen when comparing the two scatter-plots

generated by the PCA and autoencoder.



(a) Scatterplot of PCA

(b) Scatterplot of autoencoder (run 1)

Figure 4: Comparison between scatterplot of PCA and scatterplot of autoencoder.

As mentioned, the PCA is linear so it can only rotate and scale the data, so it ends up putting a lot of different digits on top of each other. It is possible to see some separation. The purples are on the left and yellows on the bottom, but there is a lot of overlapping in the middle.

The autoencoder is different. The clusters are more clearly defined. We can see the purple cluster is completely separated from the yellow cluster. Here the network has learned to group similar digits together and keep dissimilar ones apart.

If we where to run the PCA algorithm twice on the same dataset, then we wouldn't see the result change. The principal components are based on linear algebra, so there is no random initialization. There is only one set of eigenvectors that maximizes the variance.

2.4 Task 4:

We trained both the autoencoder and PCA models with latent dimension sizes, where the range is between 2 and 20. The mean squared error for both models can be seen below.

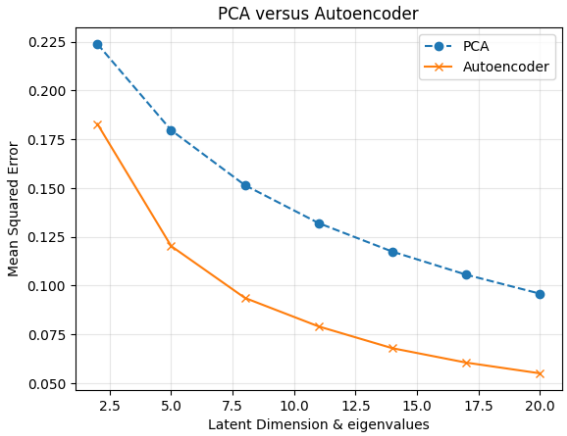


Figure 5: PCA Versus Autoencoder

Dim	PCA MSE	AE MSE
2	0.2238	0.1828
5	0.1796	0.1202
8	0.1515	0.0937
11	0.1320	0.0791
14	0.1174	0.0679
17	0.1056	0.0606
20	0.0959	0.0551

Table 3: Performance comparison.

From this we can see that the mean squared error decreases for both models, when we increase the dimensions.

The autoencoder consistently outperforms PCA at every dimension. For example, at dimension 5, the Autoencoder achieves a mean squared error of 0.12, but the PCA needs nearly 14 eigenvectors, which gives us an mean squared error of 0.1174, to reach the same level of error.

To determine the optimal number of eigenvectors, we can simply make a scree plot of the values, and then determine the elbow point. We can do something similar for the autoencoder. We can plot the reconstruction loss with the dimension size, and similar to PCA, we can look for where the error curve begins to flatten out.

2.5 Task 5:

Cell 1 & 2: Setup cells that import libraries and set the device to `cuda` or `cpu`.

Cell 3: Defines the `FF` class, consisting of a Linear layer, `ReLu` activation, and a final Linear layer.

Cell 4: Defines the VAE architecture and the encoder $\rightarrow$ Latent $\rightarrow$ decoder flow. The `encoder` maps input x to the parameters mean μ and log-variance $\log \sigma^2$. To make sure we can differentiate, the `reparameterize` method applies the so called 'reparameterization' trick, where it samples noise and calculates $z = \mu + \epsilon \odot \exp(0.5 \cdot \log \sigma^2)$. The `decoder` then ends by applying a sigmoid activation function.

Cell 5: Implements the VAE loss function, which minimizes the Negative ELBO. Based on the variational objective from the lecture slides, the loss is defined as:

$$\text{Loss} = \underbrace{-\mathbb{E}_{q_\phi(\mathbf{z}|\mathbf{x})}[\log p_\theta(\mathbf{x}|\mathbf{z})]}_{\text{BCE}} \underbrace{-\mathbb{E}_{q_\phi(z|x)}\left[\log \frac{p(\mathbf{z})}{q_\phi(\mathbf{z}|\mathbf{x})}\right]}_{\text{KLD}}$$

The code calculates the `BCE` term using `nn.functional.binary_cross_entropy`, which measures how well the output matches the input. It then calculates the `KLD` term using the formula `-0.5 * torch.sum(1 + logvar - mu.pow(2)- logvar.exp())`. Finally, the function sums these terms (`return BCE + KLD`) to get the total loss.

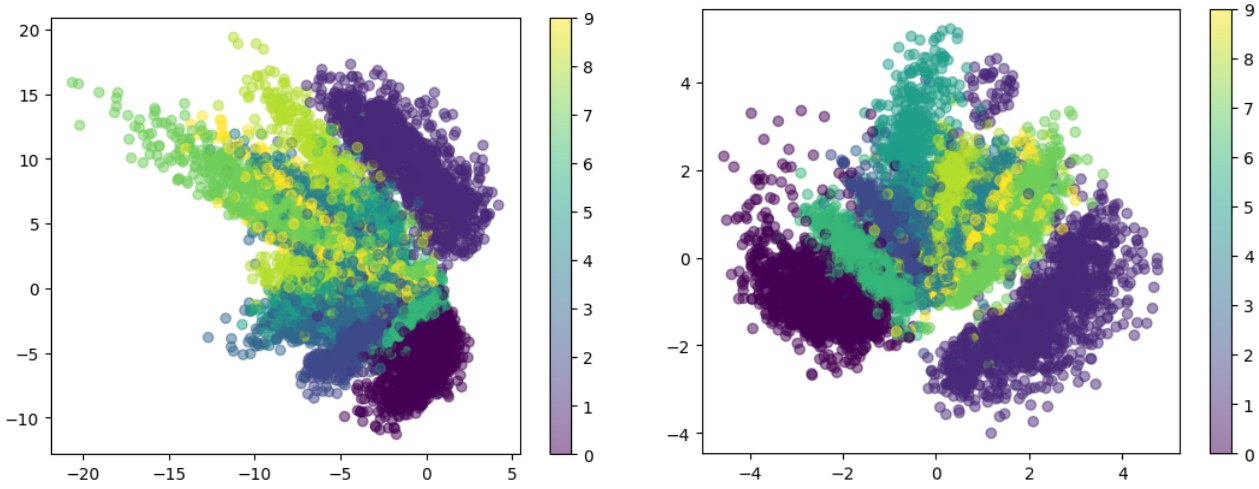
Cell 6: Defines the training loop. For every epoch, it performs the forward pass, calculates loss, executes backpropagation (`loss.backward()`), and updates parameters (`optimizer.step()`).

Cell 8: Handles data preprocessing (`transform_vae`), loads MNIST, and initializes the VAE with 512 hidden units and 2 latent dimensions. It uses the Adam optimizer with a learning rate of 10^{-3} and starts the training.

Cell 7, 9 & 10: Visualization functions for the 2D latent space and generating new images by sampling from the prior distribution.

2.6 Task 6:

Comparing the latent representations between the autoencoder and variational autoencoder, we can see a interesting detail.



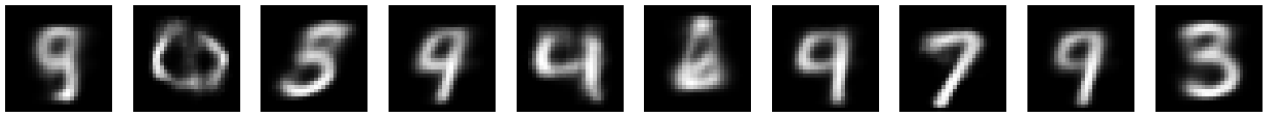
(a) Latent space for autoencoder

(b) Latent space for VAE autoencoder

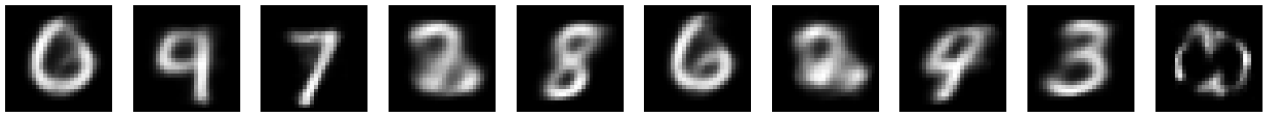
Figure 6: Autoencoder displayed next too VAE autoencoder.

Comparison: For the latent space of the regular autoencoder, it appears that the clusters extend further out, and are spread out over higher ranges. There are also more empty space between colors. This is different from the variational autoencoder. Here we see the clusters center around (0,0), and almost form a sphere with no empty gaps (compared to the latter). Additionally, the clusters aren't as spread out. They stay between (-5,5).

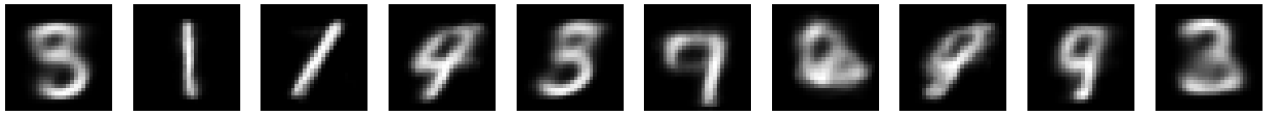
When running `plot_generated_samples` multiple times, we get 10 different images each time.



(a) First run



(b) Second run



(c) Third run

Figure 7: Main caption describing all three images.

We use `torch.randn` to sample random latent vectors from a standard normal distribution. These are then passed through the decoder to create new samples every time we call the function `plot_generated_samples`, so this is why we see different images each time.