Welcome!
#pod-031
Week #3, Day 5

(Reviewed by: Deepak)

neuromatch academy

facebook Reality Labs

Agenda

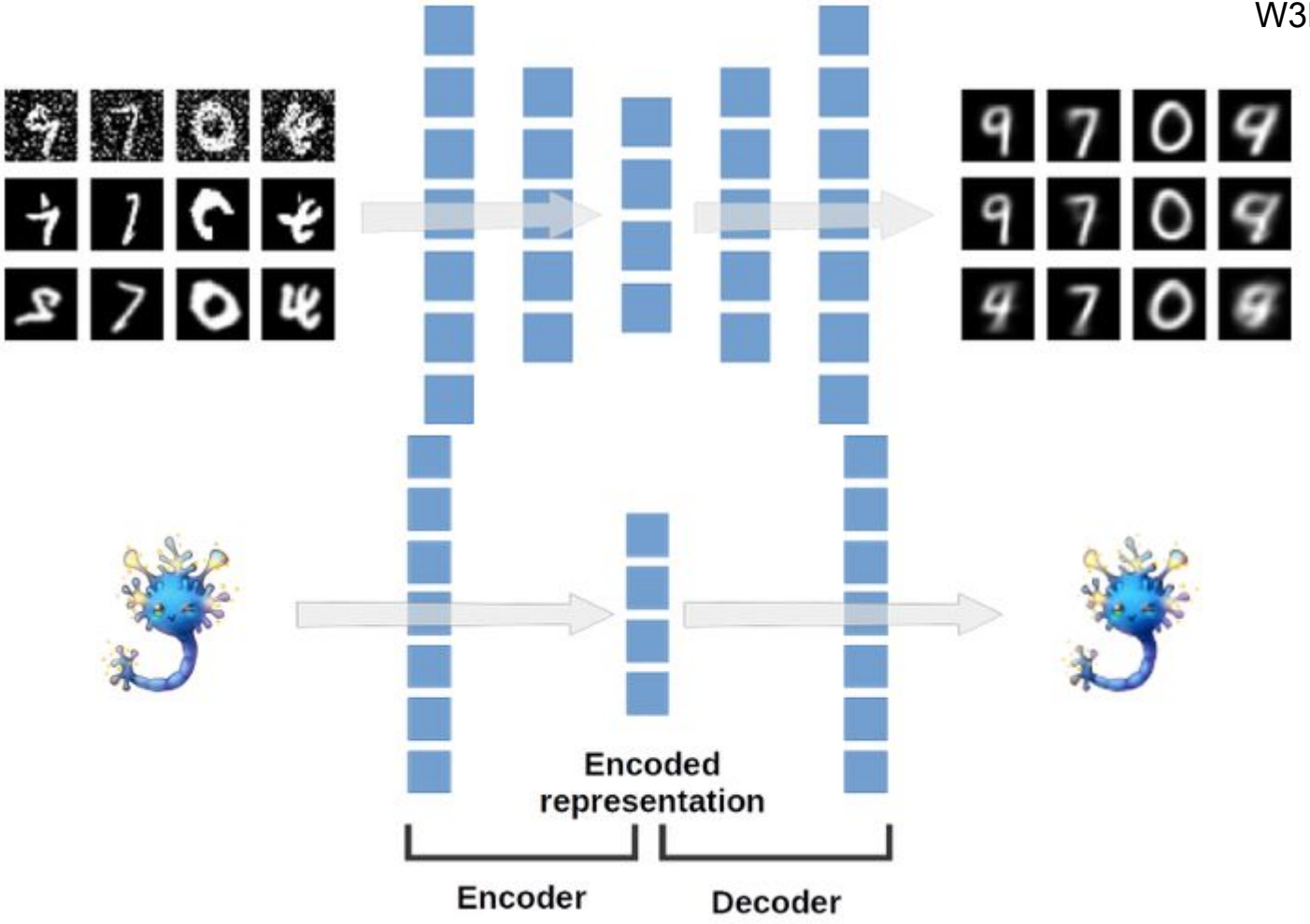
- Tutorial 1 (autoencoders)
  - 2 exercises + 1 bonus
- Tutorial 2 (extensions)
  - 2 exercises (+ bonus)
- Tutorial 3 (applications)
  - 0 exercises

Internal representations and autoencoders

Simple algorithms capture relevant aspects of data and build robust models of the world: Autoencoders are a family of artificial neural networks (ANNs) that learn internal representations through auxiliary tasks, i.e., learning by doing.

The primary task is to reconstruct output images based on a compressed representation of the inputs. This task teaches the network which details to throw away while still producing images that are similar to the inputs.

A fictitious MNIST cognitive task bundles more elaborate tasks such as removing noise from images, guessing occluded parts, and recovering original image orientation. We use the handwritten digits from the MNIST dataset since it is easier to identify similar images or issues with reconstructions than in other types of data, such as spiking data time series.


Encoded
representation
Encoder
Decoder

MNIST

The MNIST dataset contains handwritten digits in square images of 28x28 pixels of grayscale levels. There are 60,000 training images and 10,000 testing images from different writers.

- download the dataset,
- transform it into torch.Tensor and
- assign train and test datasets to (x_train, y_train) and (x_test, y_test), respectively. (x_train, x_test) contain images and (y_train, y_test) contain labels from 0 to 9.

The original pixel values are integers between 0 and 255. We rescale them between 0 and 1, a more favorable range for training the autoencoders in this tutorial. The images are *vectorized*, i.e., stretched as a line. We reshape training and testing images to *vectorized* versions with the method .reshape and store them in variable input_train and input_test, respectively. The variable image_shape stores the shape of the images, and input_size stores the size of the *vectorized* versions.

Tutorial #1 Explanations

Objective

The beauty of autoencoders is the possibility to see these internal representations. The bottleneck layer enforces data compression by having fewer units than input and output layers. Further limiting this layer to two or three units enables us to see how the autoencoder is organizing the data internally in two or three-dimensional latent space.

Roadmap: learn about typical elements of autoencoder architecture, how to extend their performance, and use them to solve the MNIST cognitive task in Tutorial 3.

- *Get acquainted with latent space visualizations and apply them to Principal Component Analysis (PCA) and Non-negative Matrix Factorization (NMF)*
- *Build and train a single hidden layer ANN autoencoder*
- *Inspect the representational power of autoencoders with latent spaces of different dimensions*
- *introduce typical elements of autoencoders, that learn low dimensional representations of data through an auxiliary task of compression and decompression. In general, these networks are characterized by an equal number of input and output units and a bottleneck layer with fewer units.*

Encoders, Decoders

Autoencoder architectures have encoder and decoder components:

- The encoder network compresses high dimensional inputs into lower-dimensional coordinates of the bottleneck layer
- The decoder expands bottleneck layer coordinates back to the original dimensionality

Each input presented to the autoencoder maps to a coordinate in the bottleneck layer that spans the lower-dimensional latent space.

Differences between inputs and outputs trigger the backpropagation of loss to adjust weights and better compress/decompress data. Autoencoders are examples of models that automatically build internal representations of the world and use them to predict unseen data.

Use fully-connected AAN architectures due to their lower computational requirements.

The inputs to ANNs are vectorized versions of the images (i.e., stretched as a line).

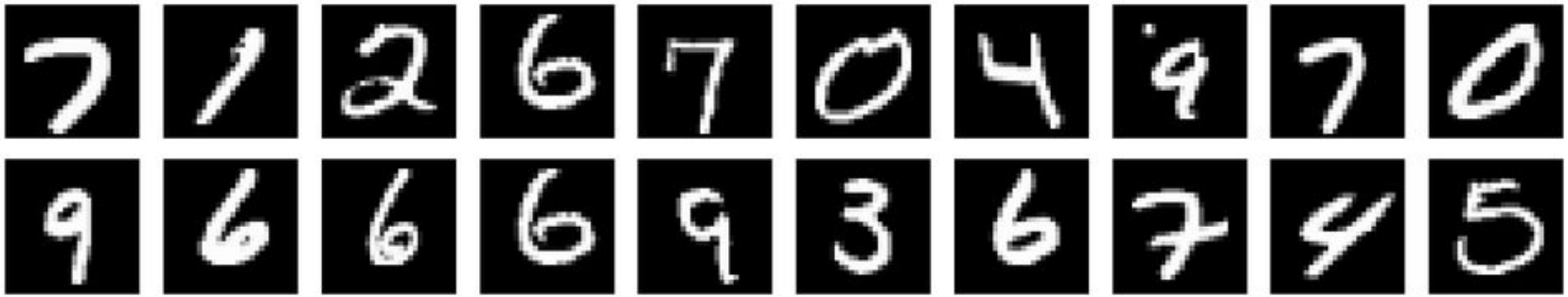
Shape and numeric representations

- What are the shape and numeric representations of x_train and input_train?
- What is the image shape?

shape x_train	torch.Size([60000, 28, 28])
shape x_test	torch.Size([10000, 28, 28])
shape image	torch.Size([28, 28])
shape input_train	torch.Size([60000, 784])
shape input_test	torch.Size([10000, 784])

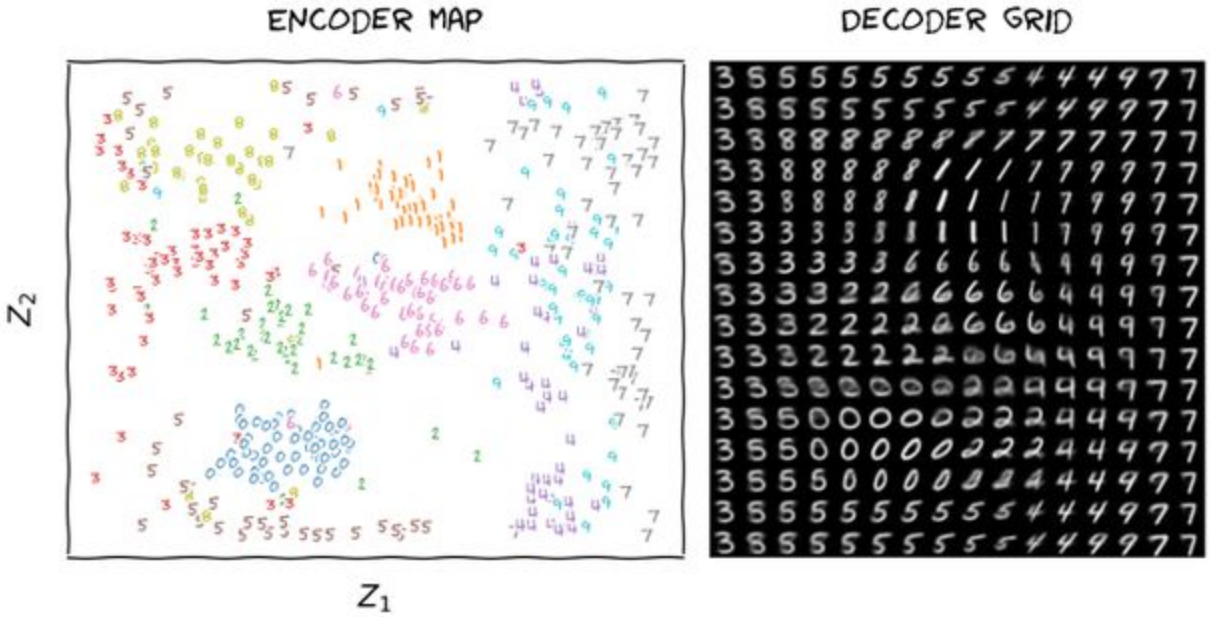
Visualize samples

train_selected_idx and test_selected_idx store 10 random indexes from the train and test data. We use the function np.random.choice to select 10 indexes from x_train and y_train without replacement (replacement=False).



Latent space visualization

- introduce tools for visualization of latent space and applies them to Principal Component Analysis (PCA)
- plotting function plot_latent_generative helps visualize the encoding of inputs from high dimension into 2D latent space, and decoding back to the original dimension. This function produces two plots:
  - **Encoder map** shows the mapping from input images to coordinates(z1,z2) in latent space, with overlaid digit labels
  - **Decoder grid** shows reconstructions from a grid of latent space coordinates(z1,z2)

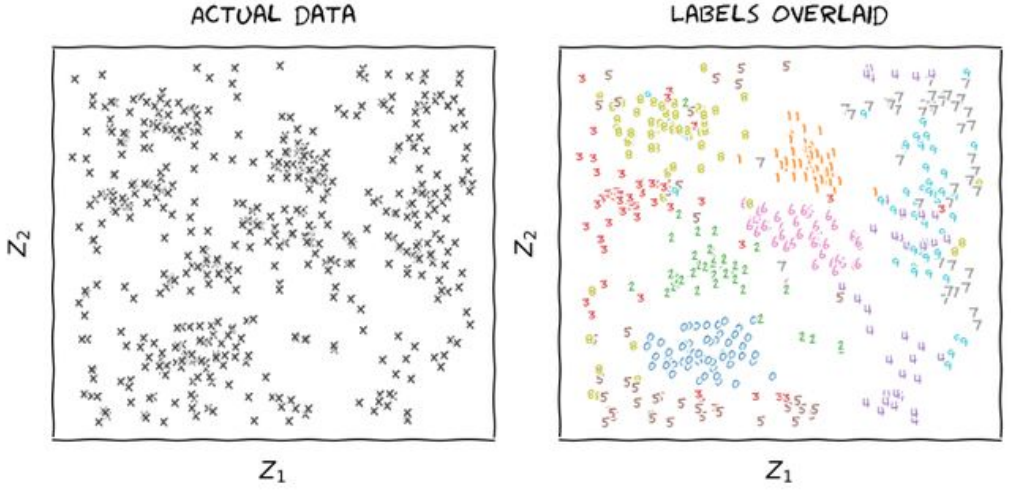

ENCODER MAP
DECODER GRID
Z_2
Z_1

Latent space representation

The latent space representation is a new coordinate system(z1,z2) that hopefully captures relevant structure from high-dimensional data. The coordinates of each input only matter relative to those of other inputs, i.e., separability between different classes of digits rather than their location.

The encoder map provides direct insight into the organization of latent space. Keep in mind that latent space only contains coordinates(z1,z2). We overlay additional information such as digit labels for insight into the latent space structure.

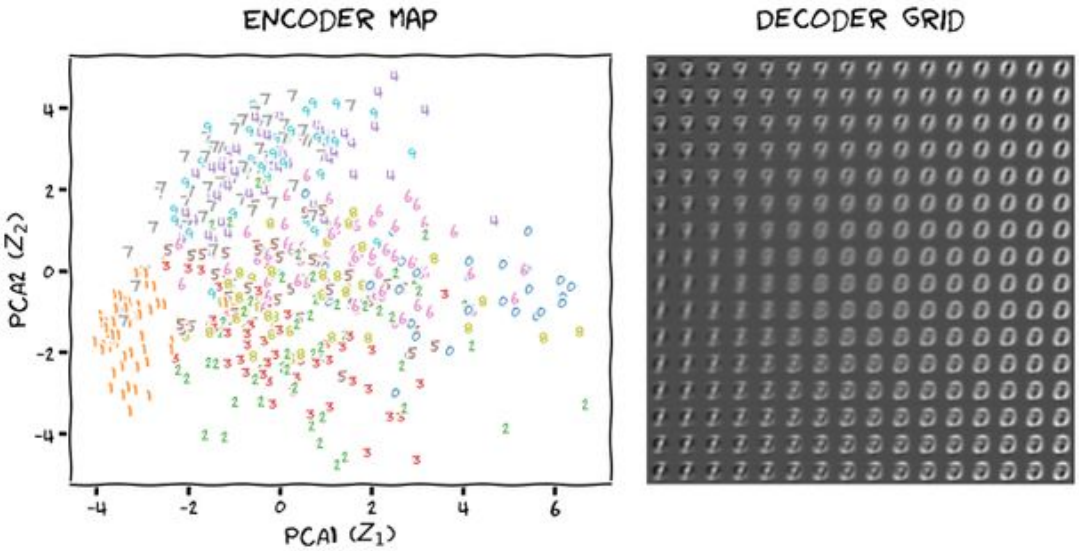
The plot on the left is the raw latent space representation corresponding to the plot on the right with digit labels overlaid.



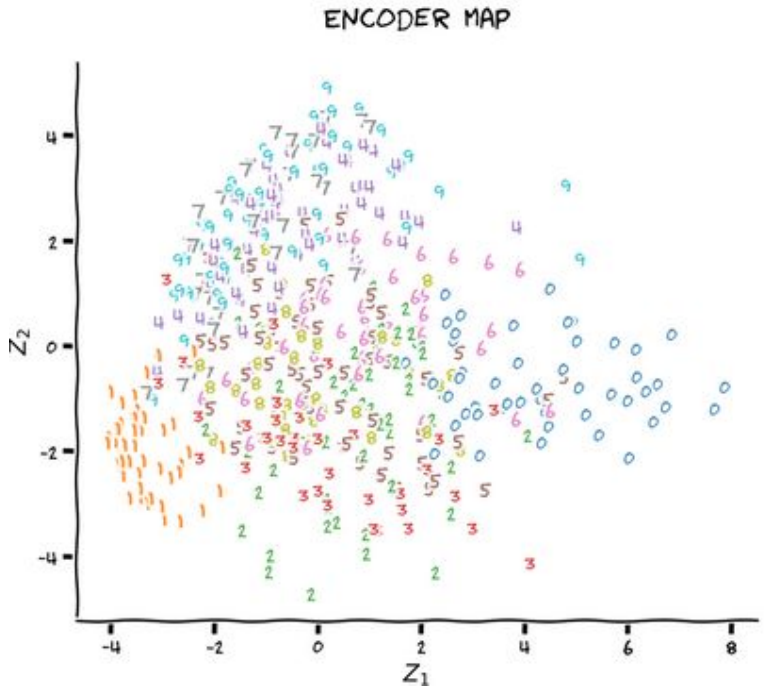
PCA Decomposition

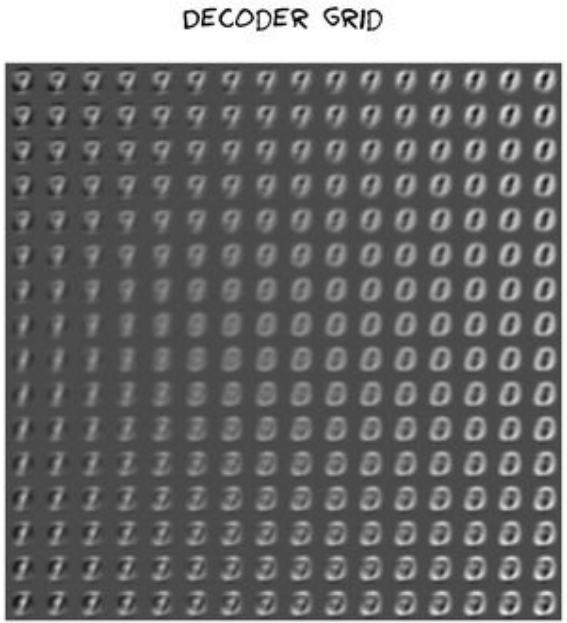
The case of two principal components (PCA1 and PCA2) generates a latent space in 2D.

- Initialize decomposition.PCA in 2 dimensions
- Fit input_train with .fit method of decomposition.PCA
- Obtain latent space representation of input_test
- Visualize latent space with plot_latent_generative



Without PCA

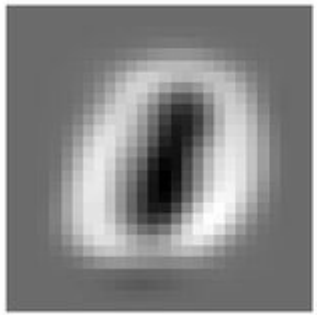




Qualitative analysis PCA - observations

The encoder map shows how well the encoder is distinguishing between digit classes. We see that digits 1 and 0 are in opposite regions of the first principal component axis, and similarly for digits 9 and 3 for the second principal component.

The decoder grid indicates how well the decoder is recovering images from latent space coordinates. Overall, digits 1, 0, and 9 are the most recognizable.



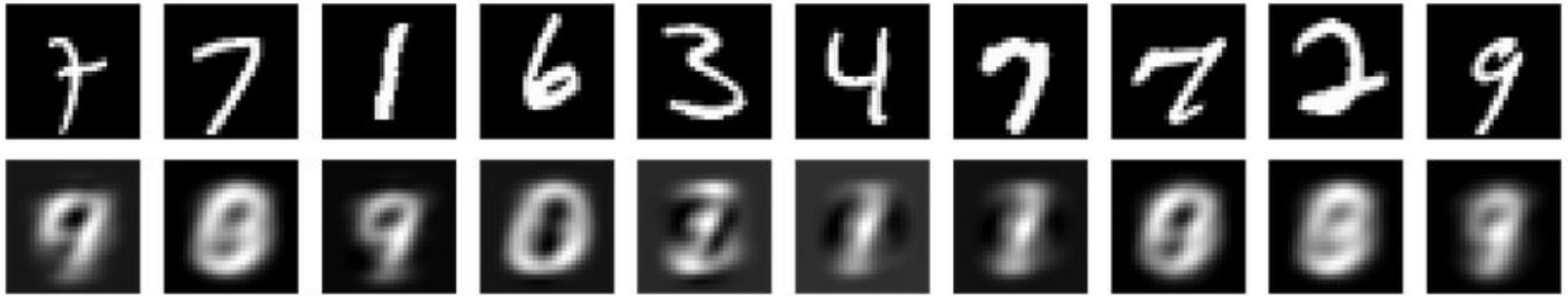
PCA - observations

First principal component encodes digit 0 with positive values (in white) and digit 1 in negative values (in Black). The colormap encodes the minimum values in Black and maximum values in white, and know their signs by looking at coordinates in the first principal component axis for digits 0 and 1.

The first principal component axis encodes the "thickness" of the digits: thin digits on the left and tick digits on the right. Similarly, the second principal component encodes digit 9 with positive values (in white) and digit 3 with negative values (in Black).

The second principal component axis is encoding, well, another aspect besides "thickness" of digits. The reconstruction grid also shows that digits 4 and 7 are indistinguishable from digit 9 and similarly for digits 2 and 3.

PCA_OUTPUT_TEST = PCA.INVERSE_TRANSFORM(PCA_LATENT_TEST)



Design ANN autoencoder (32D)

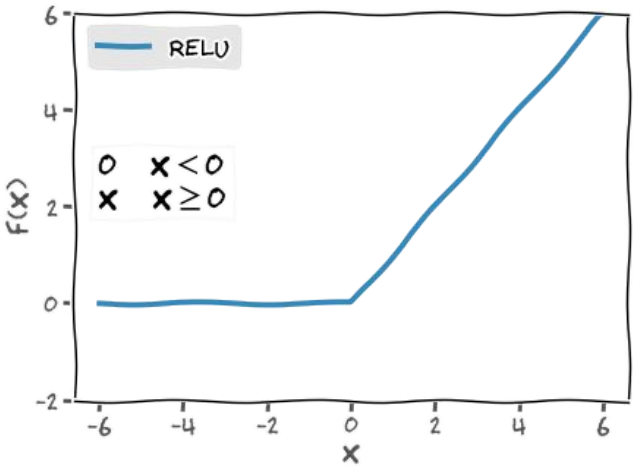
ANN with a single hidden layer ! A network equivalent to DeepNetReLU is defined as:

```
model = nn.Sequential(nn.Linear(n_input, n_hidden),
                      nn.ReLU(),
                      nn.Linear(n_hidden, n_output))
```

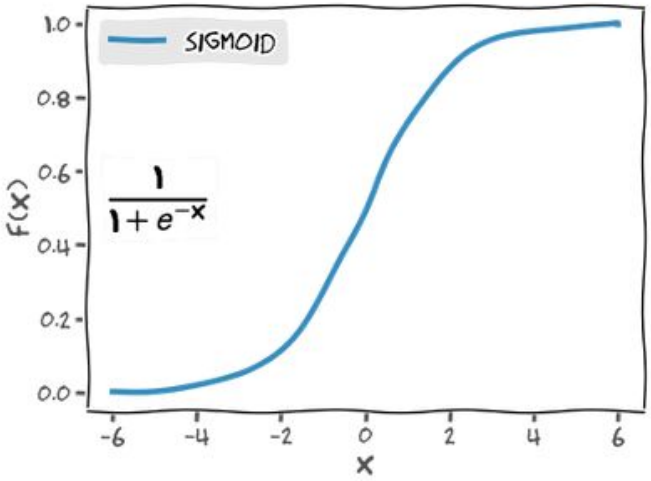
Designing and training efficient neural networks currently requires some thought, experience, and testing for choosing between available options, such as the number of hidden layers, loss function, optimizer function, mini-batch size, etc. Choosing these hyper-parameters may soon become more of an engineering process with our increasing analytical understanding of these systems and their learning dynamics.

The references below are great to learn more about neural network design and best practices:

- Neural Networks and Deep Learning by Michael Nielsen is an excellent reference for beginners
- Deep Learning by Ian Goodfellow, Yoshua Bengio, and Aaron Courville provides in-depth and extensive coverage
- A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay by L. Smith covers efficient ways to set hyper-parameters



We will use a rectifier ReLU units in the bottleneck layer with encoding_dim=32 units, and sigmoid units in the output layer.



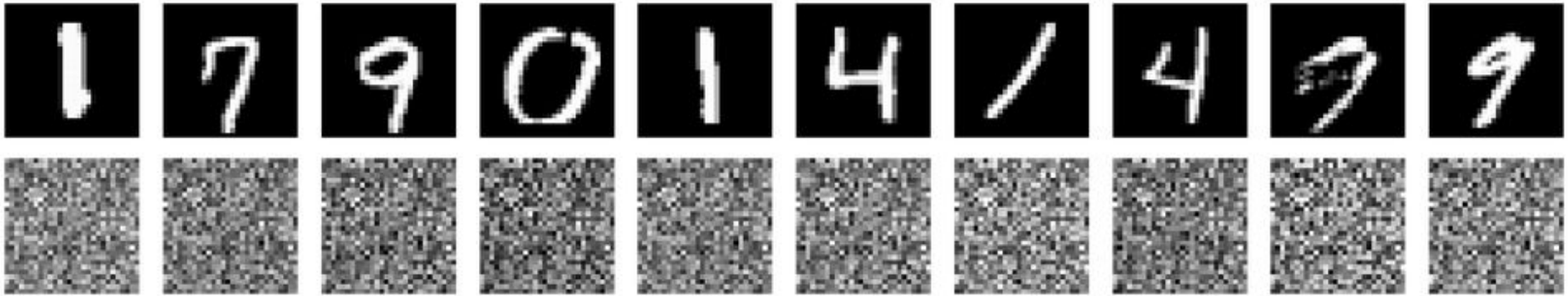
Non linearity

We rescaled images to values between 0 and 1 for compatibility with sigmoid units in the output. Such mapping is without loss of generality since any (finite) range can map a one-to-one correspondence to values between 0 and 1.

Both ReLU and sigmoid units provide non-linear computation to the encoder and decoder components. The sigmoid units, additionally, ensure output values to be in the same range as the inputs. These units could be swapped by ReLU, in which case output values would sometimes be negative or greater than 1. The sigmoid units of the decoder enforce a numerical constraint that expresses our *domain knowledge* of the data.

- nn.Sequential defines and initializes an ANN with layer sizes (input_shape, encoding_dim, input_shape)
- nn.Linear defines a linear layer with the size of the inputs and outputs as arguments
- nn.ReLU and nn.Sigmoid encode ReLU and sigmoid units
- Visualize the initial output using plot_row with input and output images

```
Sequential(  (0): Linear(in_features=784, out_features=32, bias=True)
             (1): ReLU()
             (2): Linear(in_features=32, out_features=784, bias=True)
             (3): Sigmoid())
```

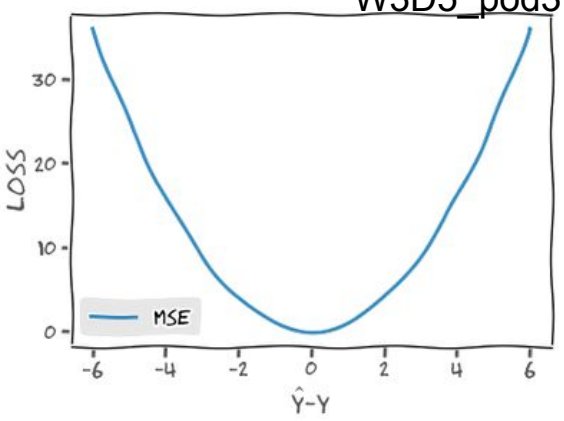


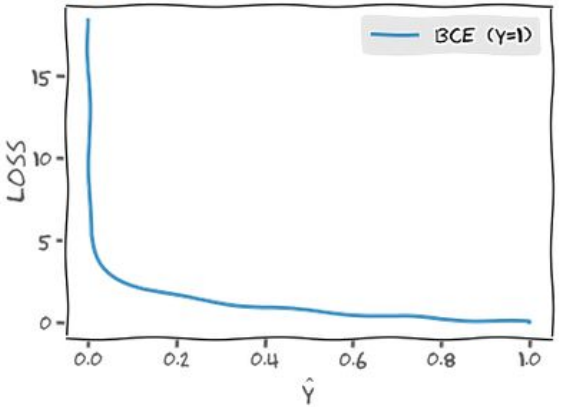
Train autoencoder (32D)

The function runSGD trains the autoencoder with stochastic gradient descent u
Adam optimizer (optim.Adam) and provides a choice between Mean Square Er
(MSE with nn.MSELoss) and Binary Cross-entropy (BCE with nn.BCELoss).

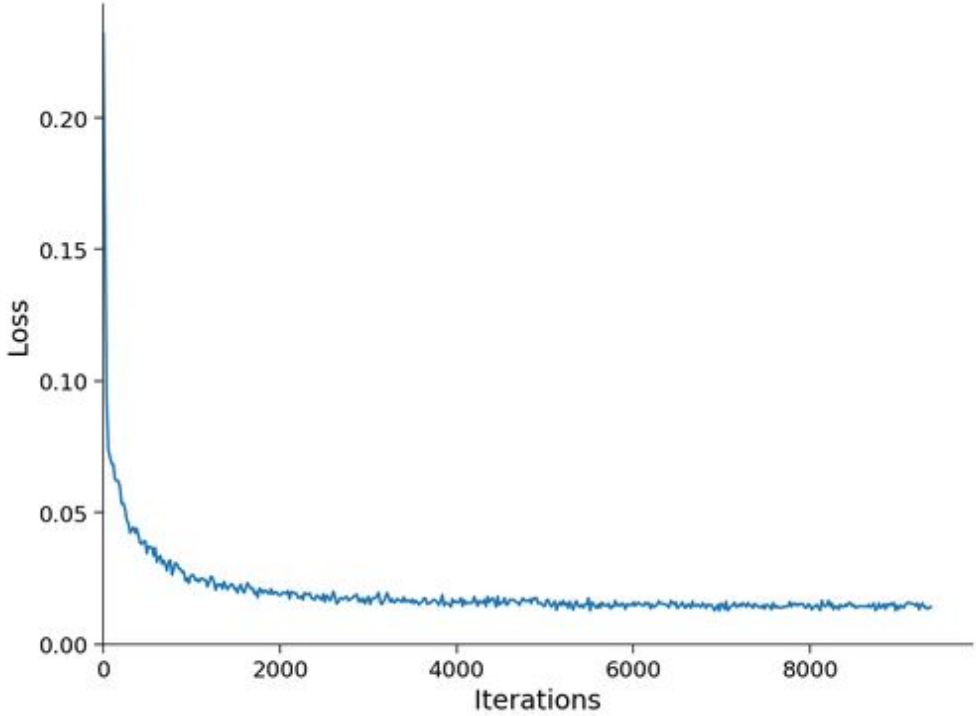
$\hat{Y}$ is the output value, and Y is the target value.

n_epochs=10 epochs and batch_size=64 with runSGD and MSE loss





Epoch	Loss train	Loss test
1 / 10	0.0268	0.0266
2 / 10	0.0196	0.0192
3 / 10	0.0172	0.0168
4 / 10	0.0163	0.0159
5 / 10	0.0158	0.0154
6 / 10	0.0149	0.0145
7 / 10	0.0147	0.0143
8 / 10	0.0146	0.0142
9 / 10	0.0145	0.0142
10 / 10	0.0144	0.0141



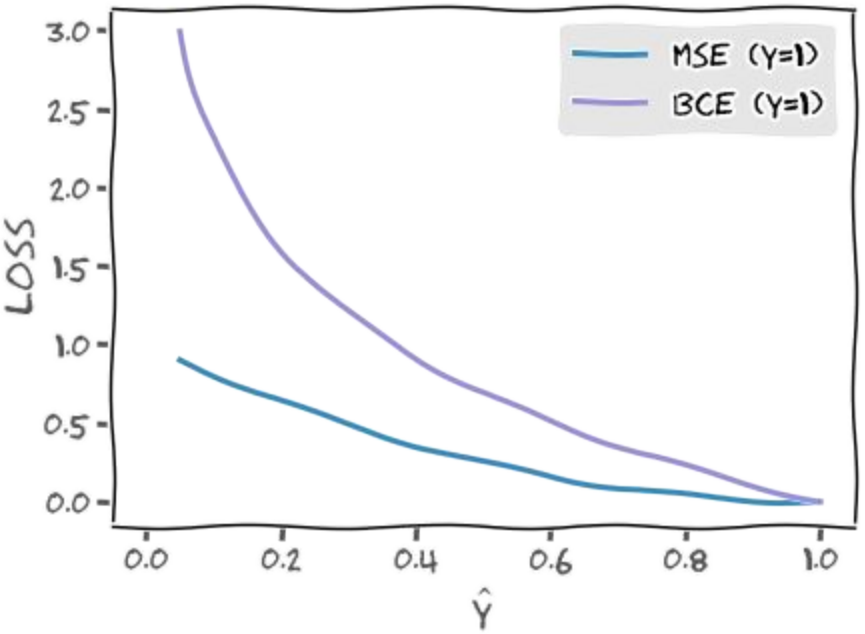
Choose the loss function

The loss function determines what the network is optimizing during training, and this translates to the visual aspect of reconstructed images.

For example, isolated black pixels in the middle of white regions are very unlikely and look noisy. The network can prioritize avoiding such scenarios by maximally penalizing white pixels that turn out black and vice-versa.

$Y=1$, and the output ranging from $\hat{Y} \in [0,1]$. The MSE loss has a gentle quadratic rise in this range.

BCE loss dramatically increases for dark pixels $\hat{Y}$ lower than 0.4.



dLoss/dŶ

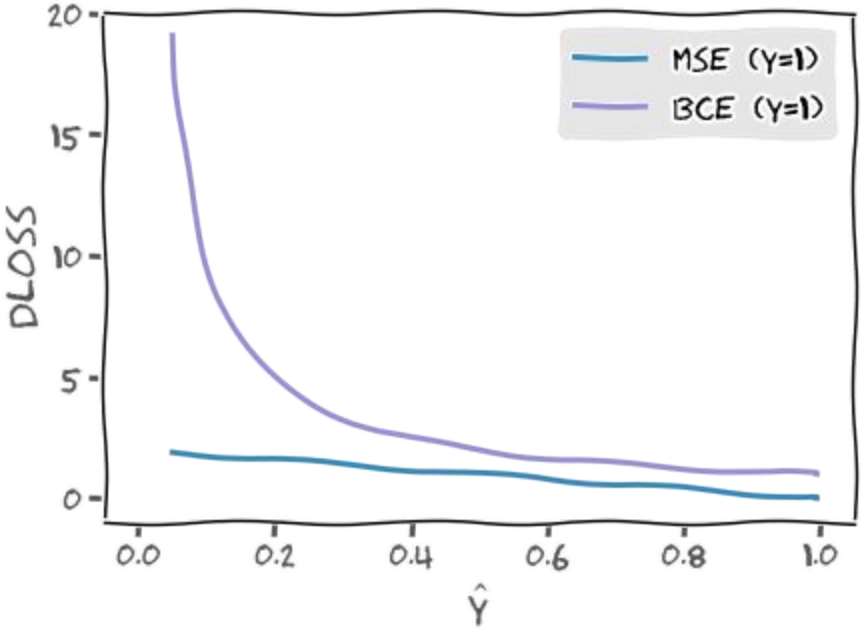
The derivative of MSE loss is linear with slope

-2

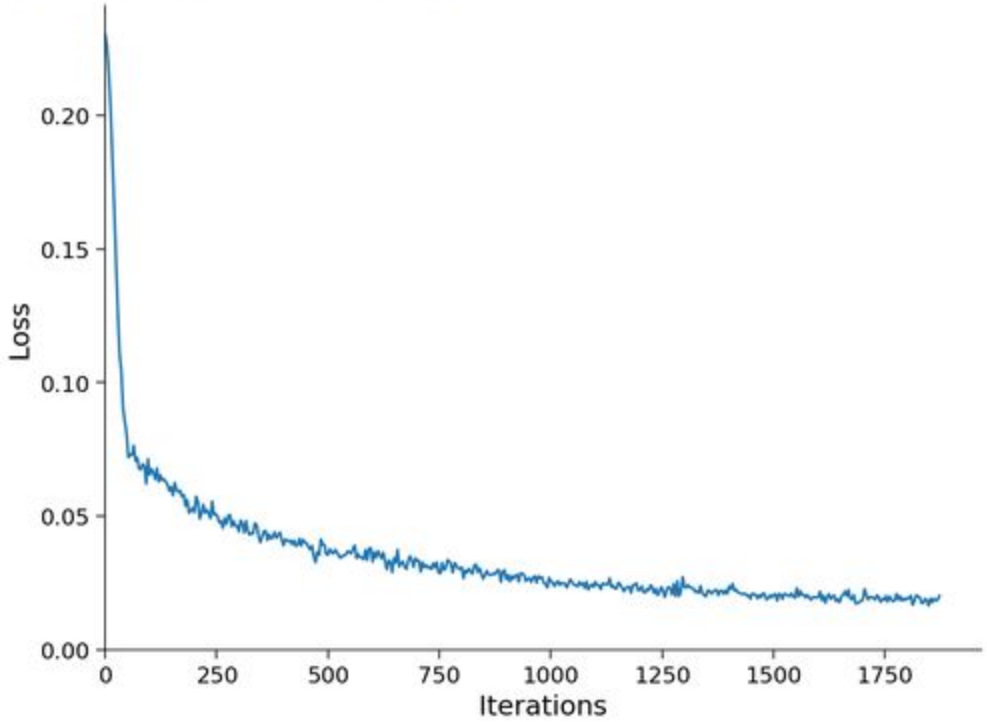
, whereas BCE takes off as 1/Ŷ for dark pixel values

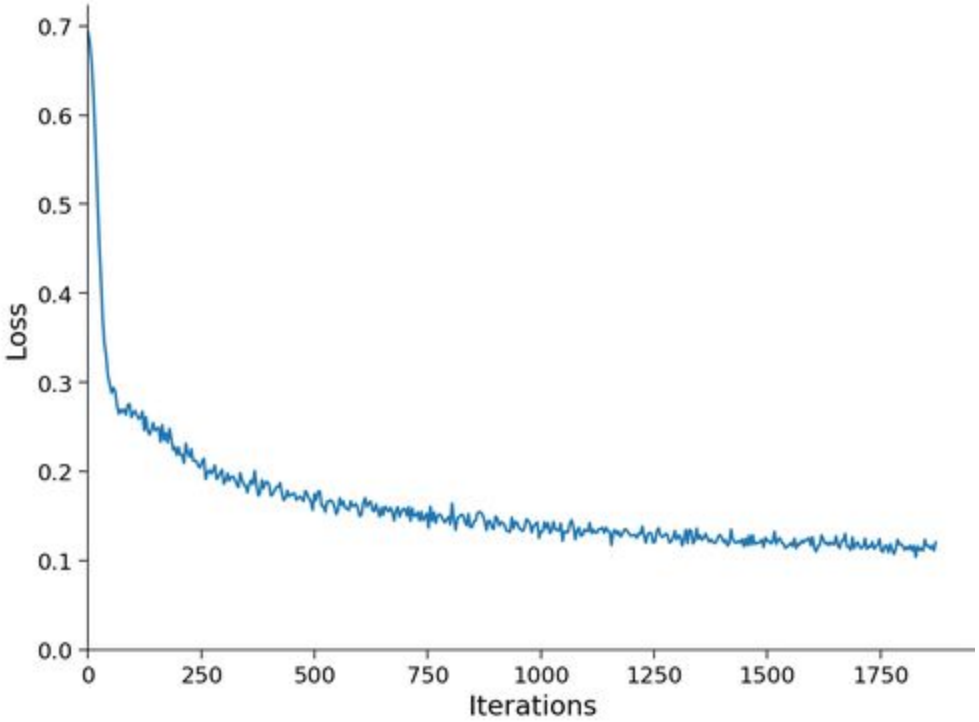
We reduced the plotting range to [0.05,1] to share the same y-axis scale for both loss functions.

Switch to BCE loss and verify the effects of maximally penalizing white pixels that turn out black and vice-versa. The visual differences between losses will be subtle since the network is converging well in both cases.



Epoch	Loss train	Loss test
1 / 2	0.0269	0.0265
2 / 2	0.0183	0.0179
MSE	0.0183	0.0179
BCE	0.1341	0.1322

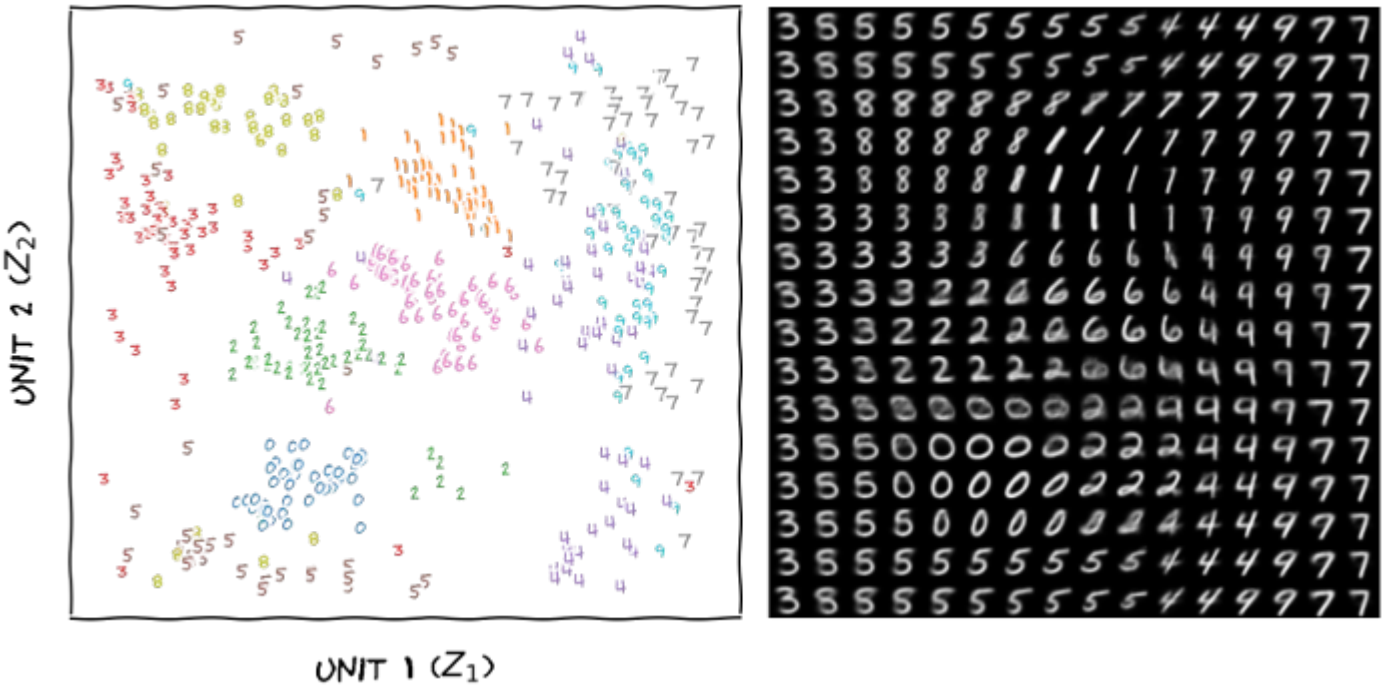



0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Loss
0
250
500
750
1000
1250
1500
1750
Iterations

Design ANN autoencoder (2D)

Reducing the number of bottleneck units to encoding_size=2 generates a 2D latent space as for PCA before. The coordinates (z1,z2) of the encoder map represent unit activations in the bottleneck layer.

The encoder component provides (z1, z2) coordinates in latent space, and the decoder component generates image reconstructions from (z1 , z2). Specifying a sequence of layers from the autoencoder network defines these sub-networks.

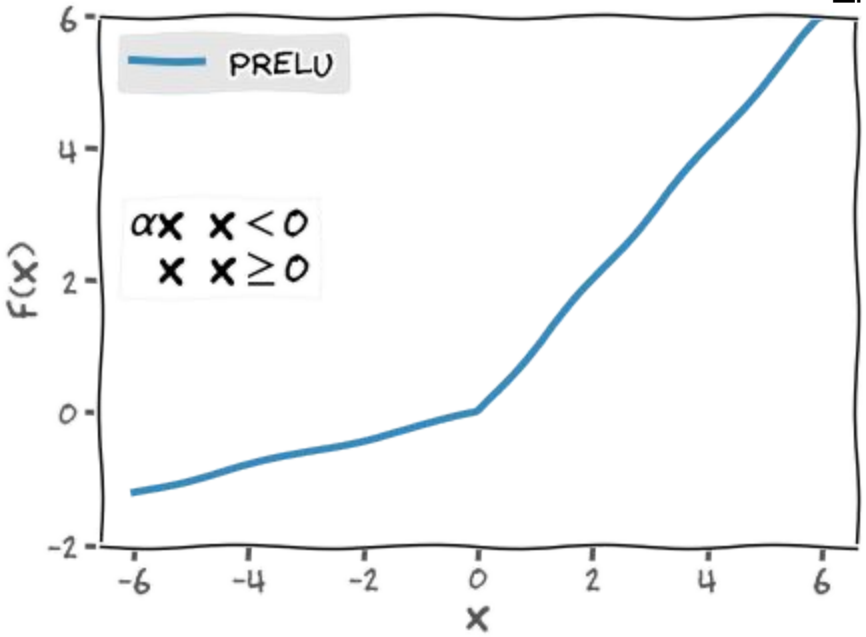

UNIT 2 (Z_2)
UNIT 1 (Z_1)

```
model = nn.Sequential(...)
encoder = model[:n]
decoder = model[n:]
```

This architecture works well with a bottleneck layer with 32 units but fails to converge with two units.

Understand this failure more and two options to address it: better weight initialization and changing the activation function.

Here we opt for PReLU units in the bottleneck layer to add negative activations with a learnable parameter. This change affords additional wiggle room for the autoencoder to model data with only two units in the bottleneck layer.



```
model = nn.Sequential(
    nn.Linear(input_size, encoding_size),
    nn.PReLU(),
    nn.Linear(encoding_size, input_size),
    nn.Sigmoid()
    )

encoder = model[:2]
decoder = model[2:]

print(f'Autoencoder \n\n {model}')
```

```
Autoencoder

 Sequential(
  (0): Linear(in_features=784, out_features=2, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=2, out_features=784, bias=True)
  (3): Sigmoid()
)
```

```
print(f'Encoder \n\n {encoder}')
```

```
Encoder

 Sequential(
  (0): Linear(in_features=784, out_features=2, bias=True)
  (1): PReLU(num_parameters=1)
)
```

```
print(f'Decoder \n\n {decoder}')
```

```
Decoder

 Sequential(
  (2): Linear(in_features=2, out_features=784, bias=True)
  (3): Sigmoid()
)
```

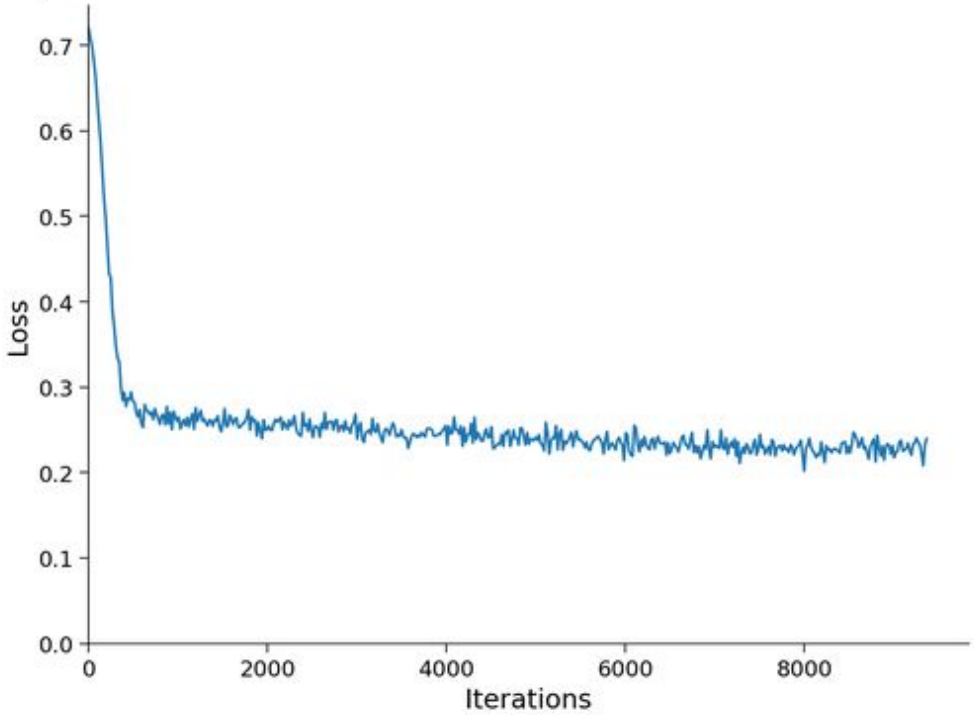
W3D5_pod31

n_epochs=10

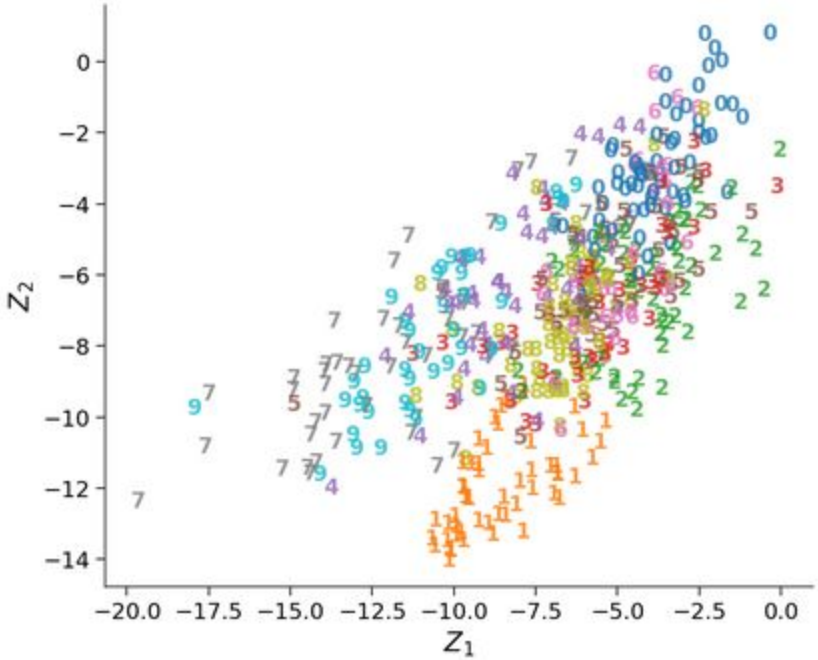
and batch_size=64

with runSGD and BCE loss

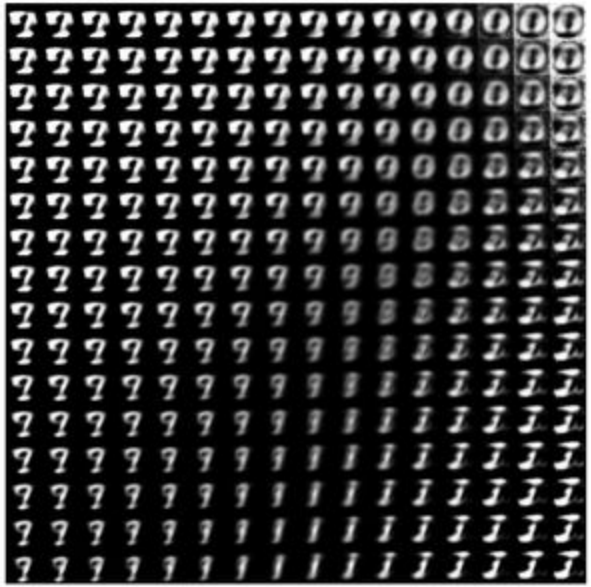
Epoch	Loss train	Loss test
1 / 10	0.2632	0.2629
2 / 10	0.2573	0.2571
3 / 10	0.2523	0.2521
4 / 10	0.2466	0.2462
5 / 10	0.2405	0.2400
6 / 10	0.2350	0.2345
7 / 10	0.2316	0.2309
8 / 10	0.2297	0.2288
9 / 10	0.2285	0.2276
10 / 10	0.2279	0.2270



Encoder map



Decoder grid



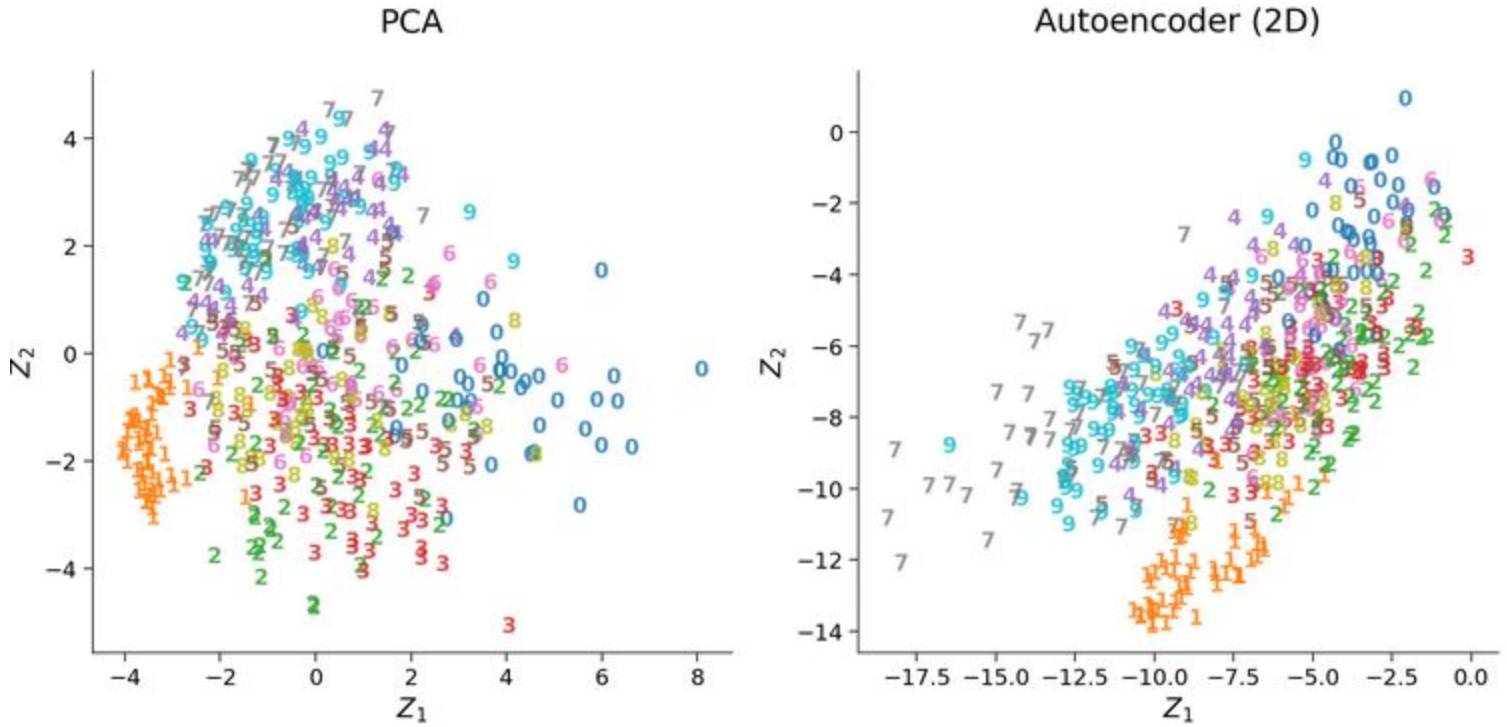
Expressive power in 2D

The latent space representation of shallow autoencoder with a 2D bottleneck is similar to that of PCA.

How is this linear dimensionality reduction technique be compared to our non-linear autoencoder?

Training an autoencoder with linear activation functions under MSE loss is very similar to performing PCA. Using piece-wise linear units, sigmoidal output unit, and BCE loss doesn't seem to change this behavior qualitatively. The network lacks capacity in terms of learnable parameters to make good use of its non-linear operations and capture non-linear aspects of the data.

The similarity between representations is apparent when plotting decoder maps side-by-side. Look for classes of digits that cluster successfully, and those still mixing with others.


PCA
Autoencoder (2D)
Z_1
Z_2
Z_1
Z_2

Summary

We saw that PCA and shallow autoencoder have similar expressive power in 2D latent space, despite the autoencoder's non-linear character.

The shallow autoencoder lacks learnable parameters to take advantage of non-linear operations in encoding/decoding and capture non-linear patterns in data.

Tutorial #1 bonus Explanations

Failure mode with ReLU units in 2D

An architecture with two units in the bottleneck layer, ReLU units, and default weight initialization may fail to converge, depending on the minibatch sequence, choice of the optimizer, etc. To illustrate this failure mode, we first set the random number generators (RNGs) to reproduce an example of failed convergence:

```
torch.manual_seed(0)
np.random.seed(0)
```

Afterward, we set the RNGs to reproduce an example of successful convergence:

```
torch.manual_seed(1)
```

Train the network for n_epochs=10 epochs and batch_size=64 and check the encoder map and reconstruction grid in each case.

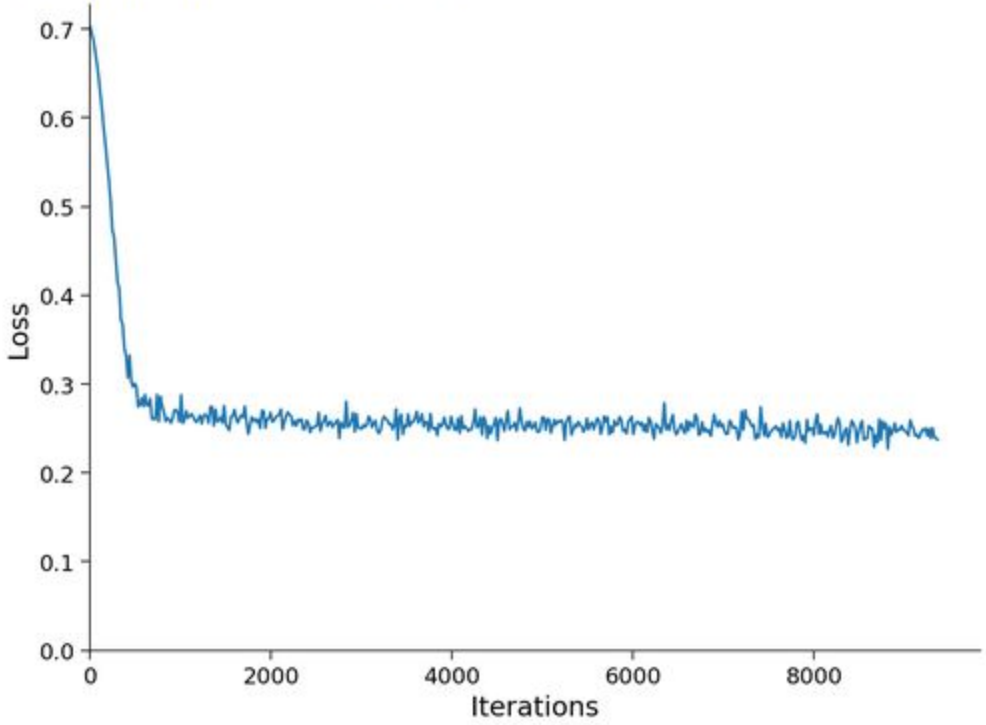
We then activate our x-ray vision and check the distribution of weights in encoder and decoder components. Recall that encoder maps input pixels to botttleneck units (encoder weights shape=(2, 784)), and decoder maps bottleneck units to output pixels (decoder weights shape=(784, 2)).

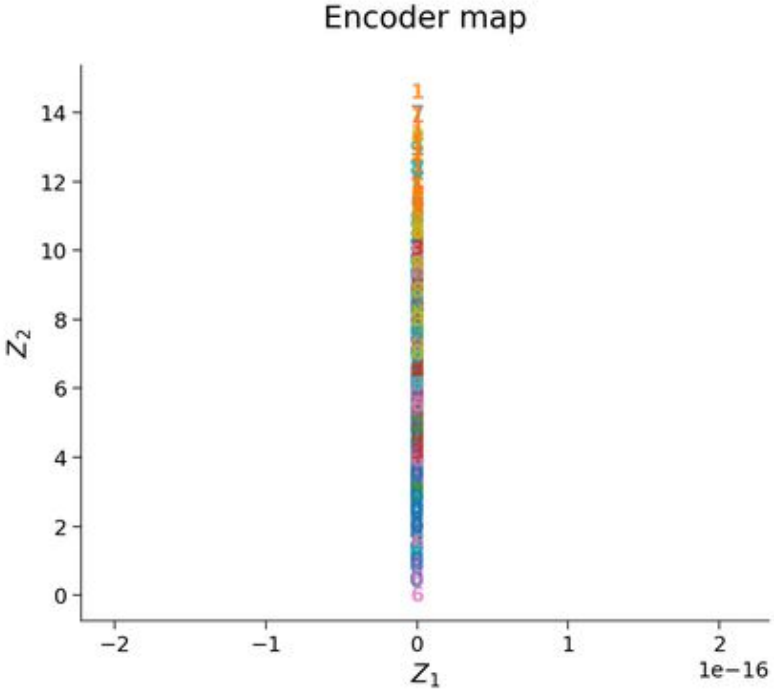
Network models often initialize with random weights close to 0. The default weight initialization for linear layers in Pytorch is sampled from a uniform distribution [-limit, limit] with limit=1/sqrt(fan_in), where fan_in is the number of input units in the weight tensor.

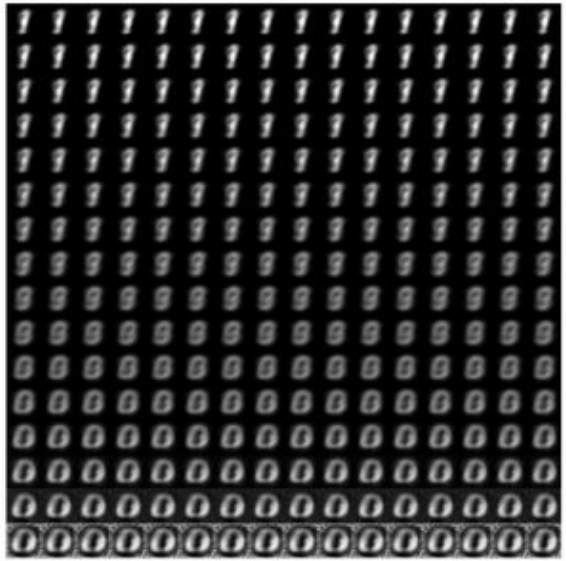
We compare the distribution of weights on network initialization to that after training. Weights that fail to learn during training keep to their initial distribution. On the other hand, weights that are adjusted by SGD during training are likely to have a change in distribution.

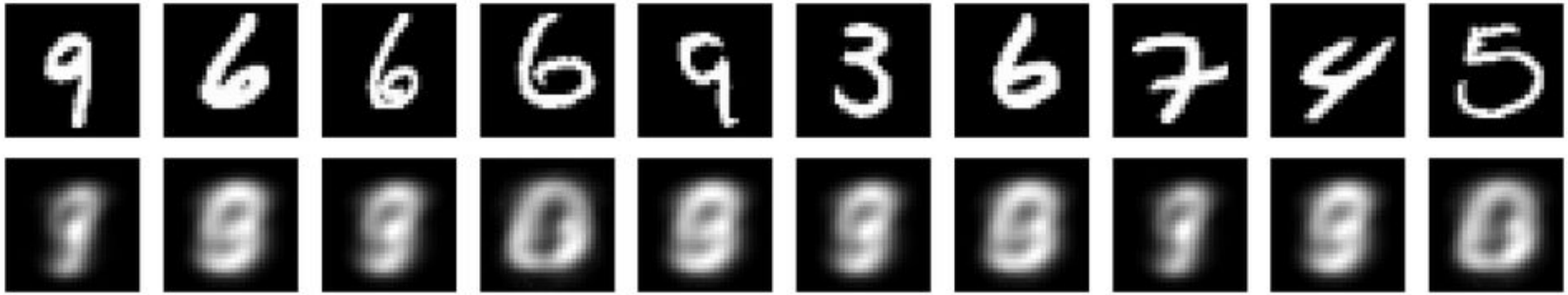
Encoder weights may even acquire a bell-shaped form. This effect may be related to the following: SGD adds a sequence of positive and negative increments to each initial weight. The Central Limit Theorem (CLT) would predict a gaussian histogram if increments were independent in sequences and between sequences. The deviation from gaussianity is a measure of the inter-dependency of SGD increments.

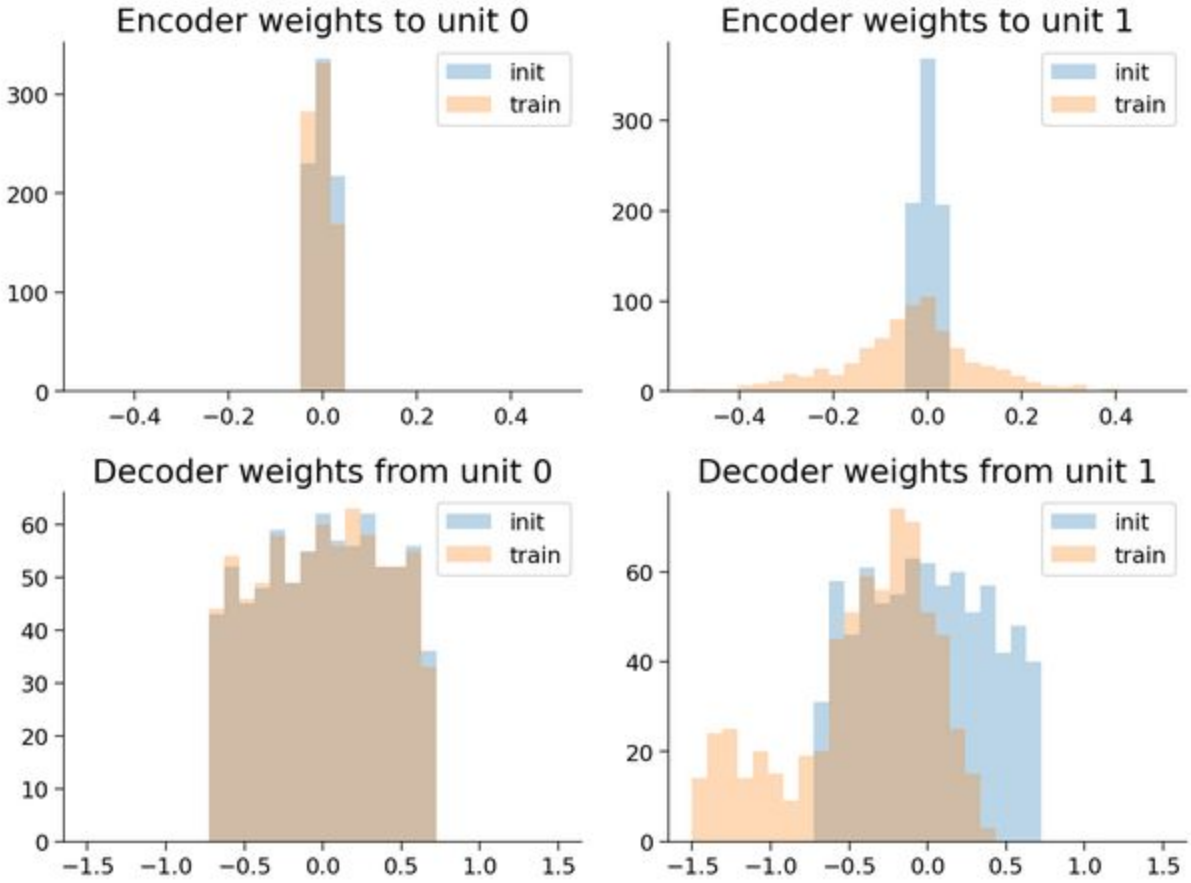
Epoch	Loss train	Loss test
1 / 10	0.2654	0.2651
2 / 10	0.2600	0.2598
3 / 10	0.2573	0.2572
4 / 10	0.2554	0.2552
5 / 10	0.2539	0.2538
6 / 10	0.2524	0.2523
7 / 10	0.2509	0.2508
8 / 10	0.2493	0.2491
9 / 10	0.2476	0.2474
10 / 10	0.2460	0.2458




Encoder map
14
12
10
8
6
4
2
0
Z_2
−2
−1
0
1
2
1e−16
Z_1


Decoder grid




Encoder weights to unit 0
init
train
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4
Encoder weights to unit 1
init
train
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4
Decoder weights from unit 0
init
train
60
50
40
30
20
10
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
Decoder weights from unit 1
init
train
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

An improved weight initialization for ReLU units avoids the failure mode:

A popular choice for rectifier units is Kaiming uniform: sampling from uniform distribution

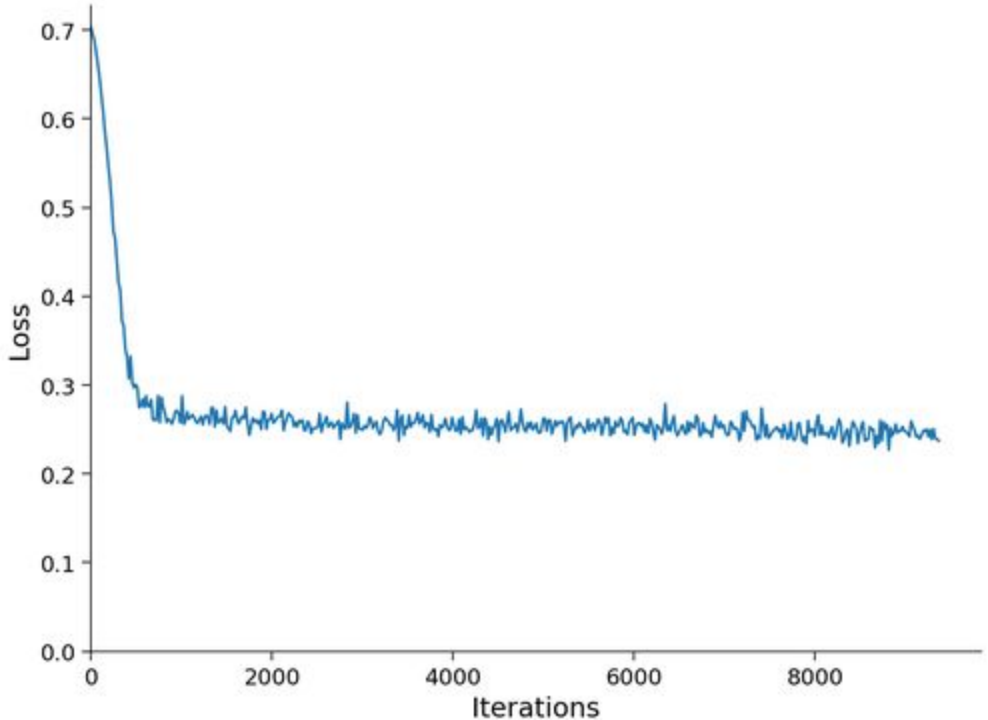
$$\mathcal{U}(-limit, limit) \text{ with } limit = \sqrt{6/fan\_in},$$

Where *fan_in* is the number of input units in the weight tensor.

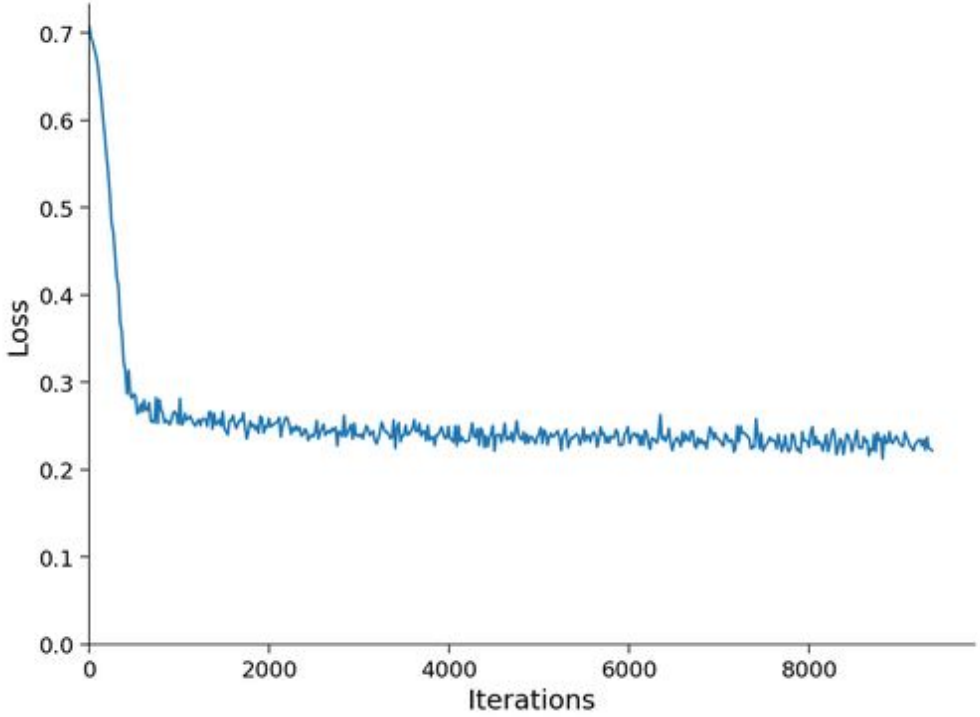
An alternative is to sample from a gaussian distribution $\mathcal{N}(\mu, \sigma 2)$ with $\mu = 0$ and

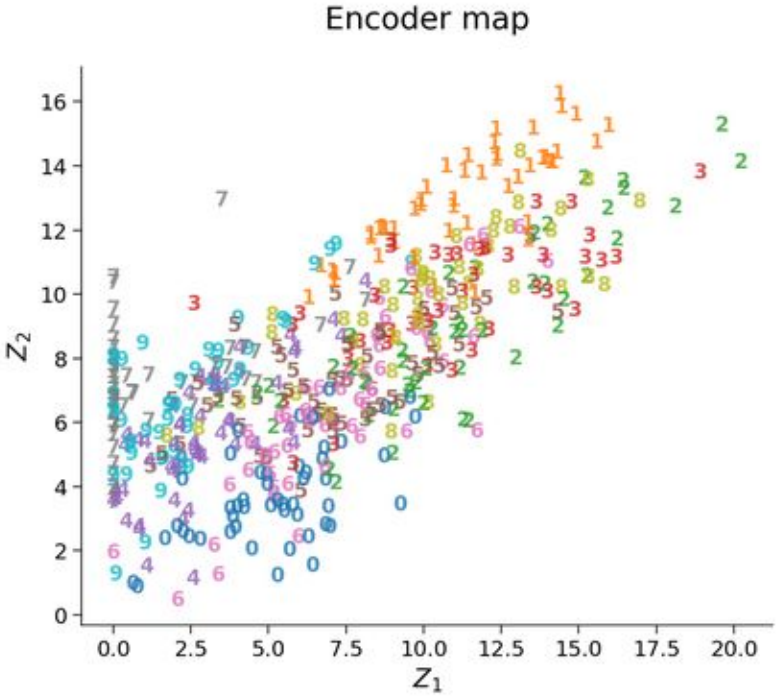
Example for reseting all but the two last autoencoder layers to Kaiming normal: $\sigma = 1/\sqrt{fan\_in}.$

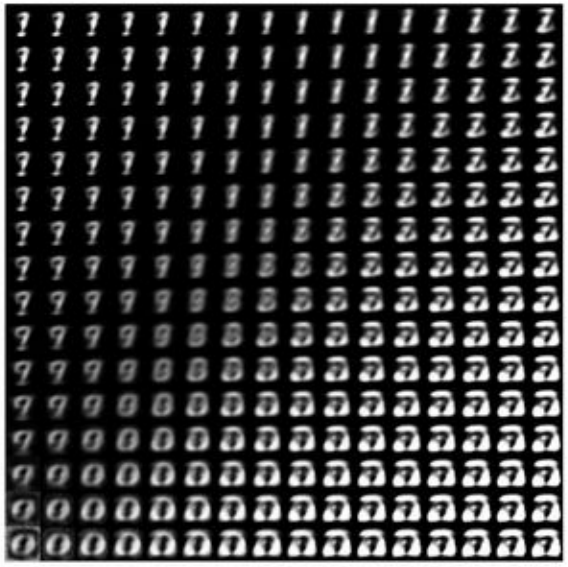
Epoch	Loss train	Loss test
1 / 10	0.2654	0.2651
2 / 10	0.2600	0.2598
3 / 10	0.2573	0.2572
4 / 10	0.2554	0.2552
5 / 10	0.2539	0.2538
6 / 10	0.2524	0.2523
7 / 10	0.2509	0.2508
8 / 10	0.2493	0.2491
9 / 10	0.2476	0.2474
10 / 10	0.2460	0.2458

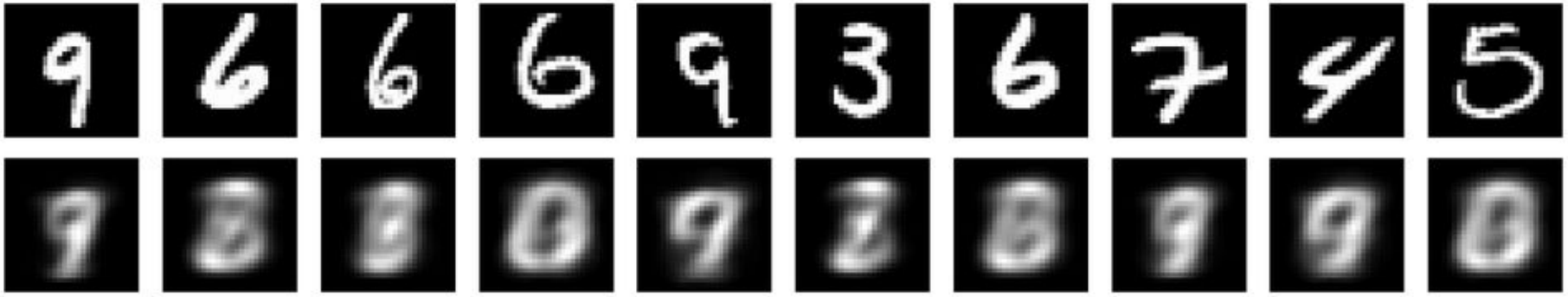


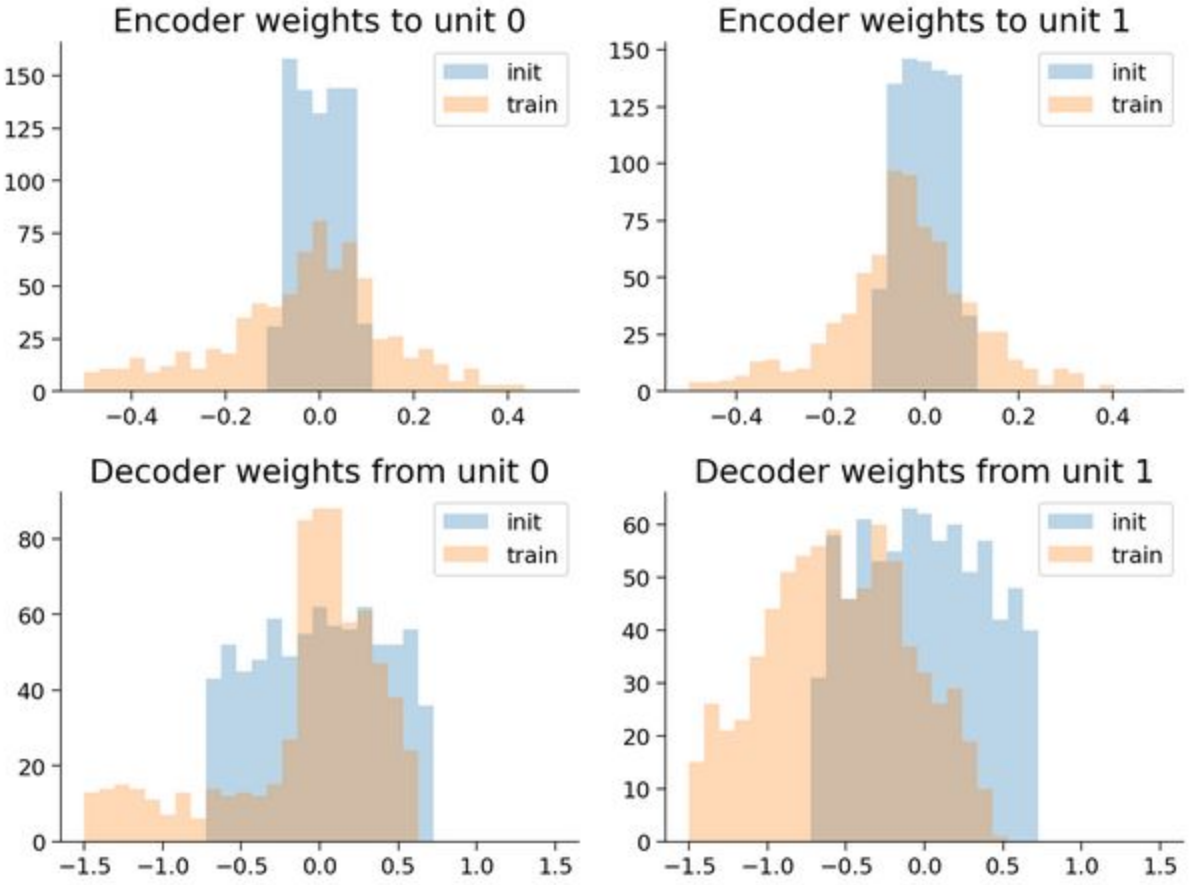
Epoch	Loss train	Loss test
1 / 10	0.2602	0.2599
2 / 10	0.2505	0.2501
3 / 10	0.2447	0.2442
4 / 10	0.2409	0.2401
5 / 10	0.2380	0.2370
6 / 10	0.2357	0.2347
7 / 10	0.2339	0.2328
8 / 10	0.2325	0.2314
9 / 10	0.2312	0.2300
10 / 10	0.2303	0.2292




Encoder map
Z_2
Z_1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
2
4
6
8
10
12
14
16


Decoder grid

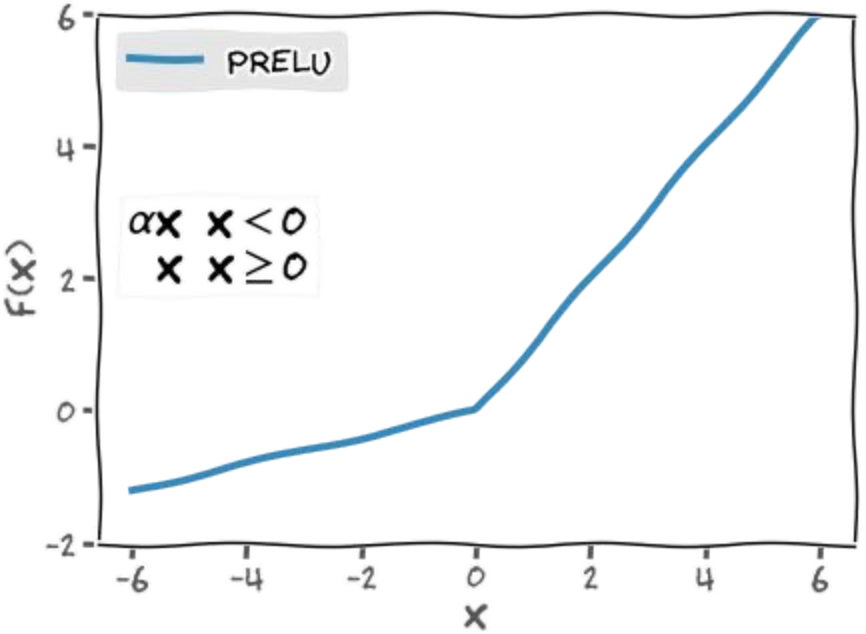



Encoder weights to unit 0
init
train
150
125
100
75
50
25
0
−0.4
−0.2
0.0
0.2
0.4
Encoder weights to unit 1
init
train
150
125
100
75
50
25
0
−0.4
−0.2
0.0
0.2
0.4
Decoder weights from unit 0
init
train
80
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
Decoder weights from unit 1
init
train
60
50
40
30
20
10
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

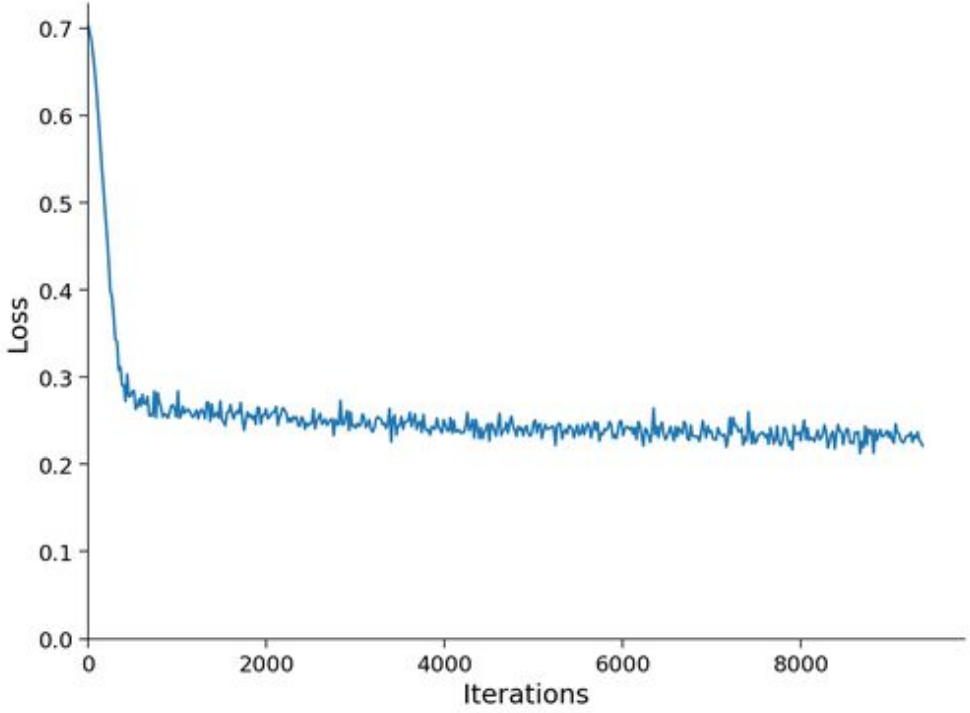
Choose the activation function

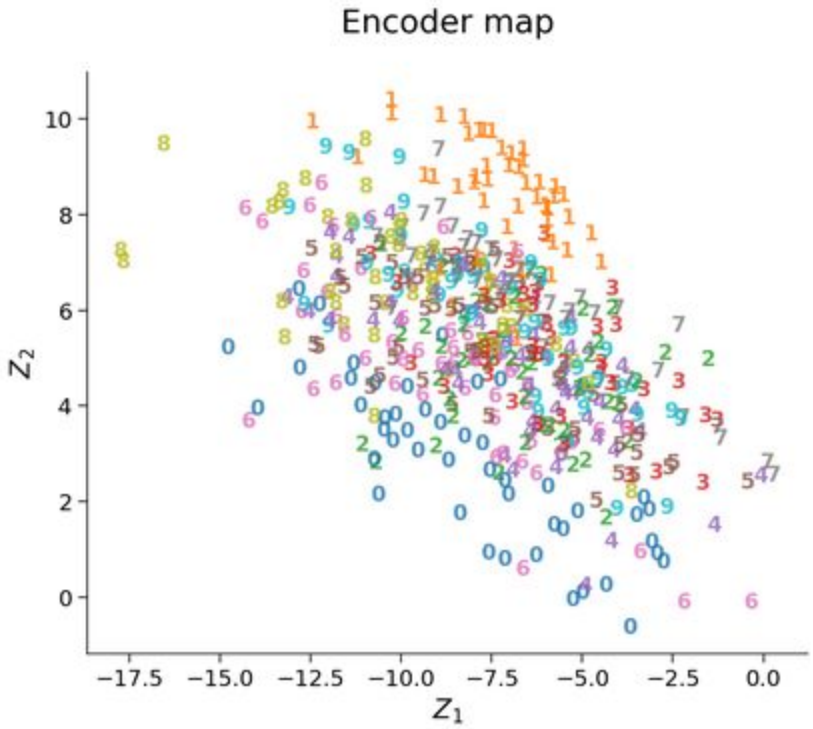
An alternative to specific weight initialization is to choose an activation unit that performs better in this context. We will use PReLU units in the bottleneck layer, which adds a learnable parameter for negative activations.

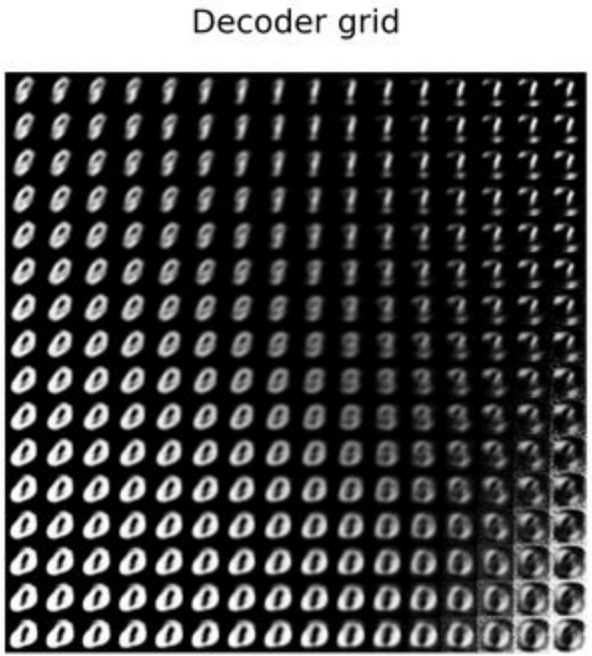
This change affords a little bit more of wiggle room for the autoencoder to model data compared to ReLU units.

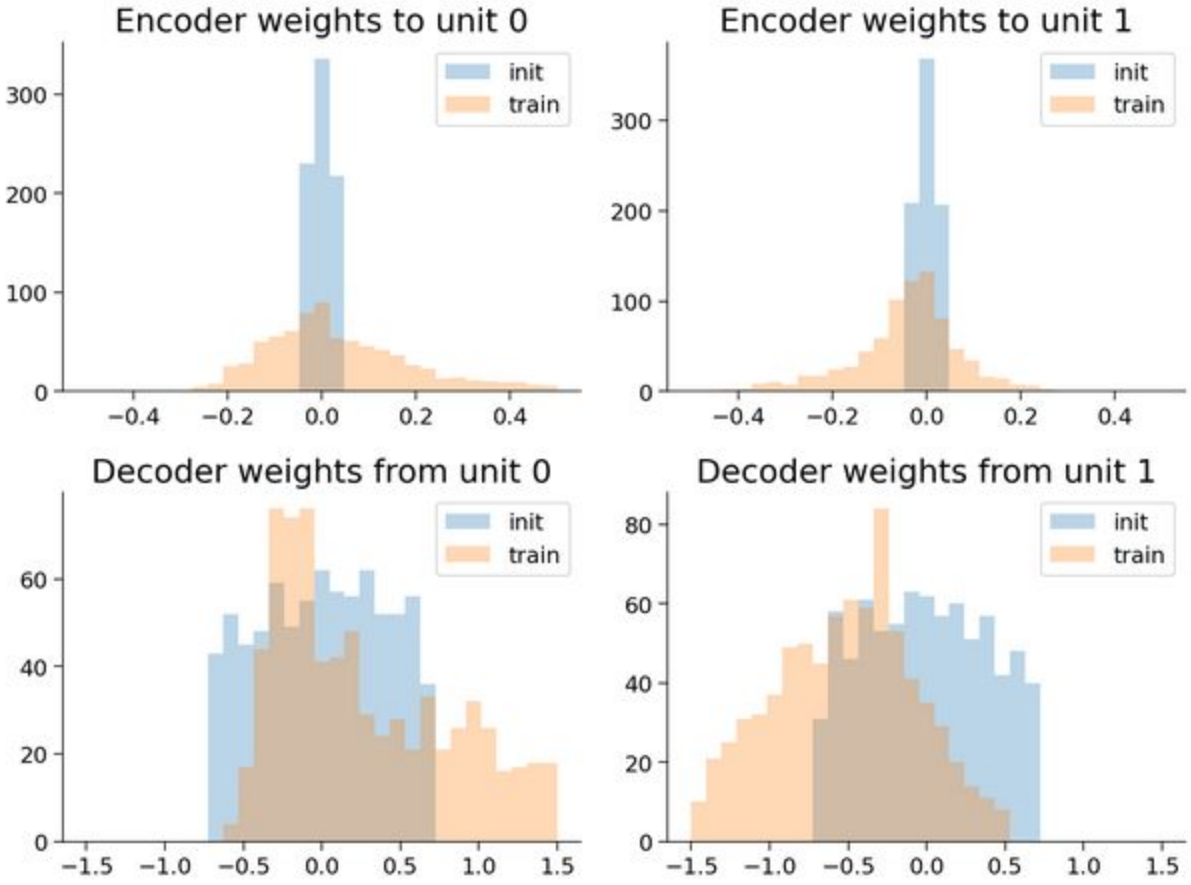


Epoch	Loss train	Loss test
1 / 10	0.2622	0.2619
2 / 10	0.2560	0.2558
3 / 10	0.2509	0.2505
4 / 10	0.2457	0.2451
5 / 10	0.2411	0.2405
6 / 10	0.2378	0.2373
7 / 10	0.2356	0.2352
8 / 10	0.2340	0.2337
9 / 10	0.2327	0.2325
10 / 10	0.2318	0.2319




Encoder map
10
8
6
4
2
0
Z_2
−17.5
−15.0
−12.5
−10.0
−7.5
−5.0
−2.5
0.0
Z_1


Decoder grid


Encoder weights to unit 0
init
train
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4
Encoder weights to unit 1
init
train
300
200
100
0
−0.4
−0.2
0.0
0.2
0.4
Decoder weights from unit 0
init
train
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
Decoder weights from unit 1
init
train
80
60
40
20
0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5

Qualitative analysis NMF

A product of positive matrices W and H approximates data matrix X

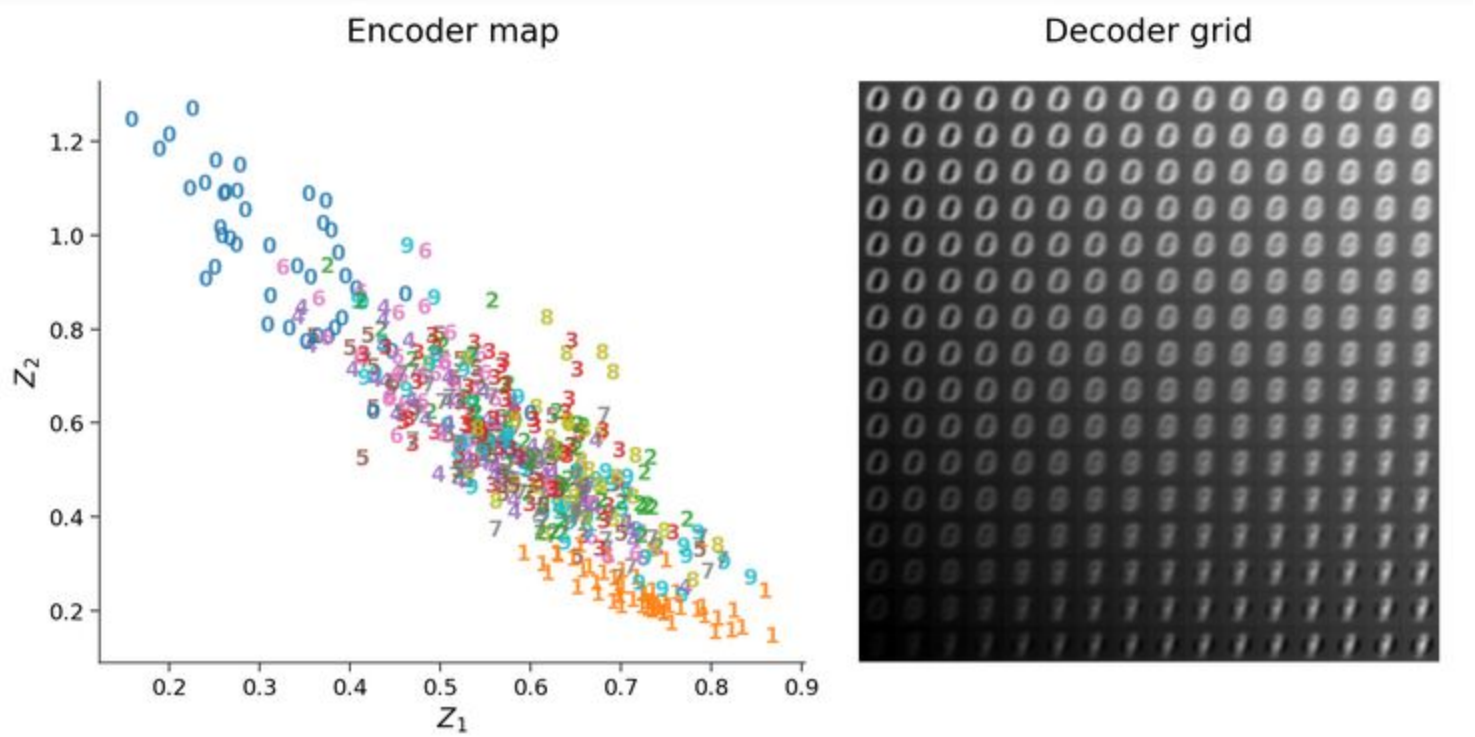
, i.e., $X \approx WH$

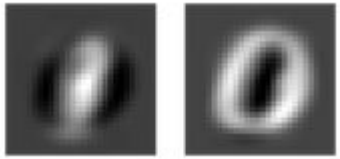
The columns of W play the same role as the principal components in PCA.

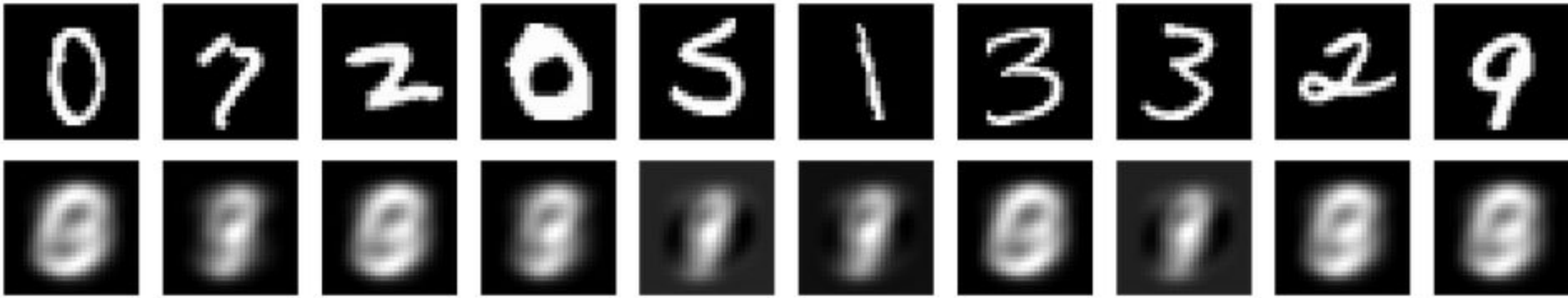
Digit classes 0 and 1 are the furthest apart in latent space and better clustered.

Looking at the first component, we see that images gradually resemble digit class 0. A mix between digits classes 1 and 9 in the second component shows a similar progression.

That data is shifted by 0.5 to avoid failure modes near 0 - this is probably related to our scaling choice.


Encoder map
Decoder grid
Z_1
Z_2
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.2
1.0
0.8
0.6
0.4
0.2





Tutorial #2
Explanations

Architecture

How can we improve the internal representation of shallow autoencoder with 2D bottleneck layer?

We may try the following architecture changes:

- *Introducing additional hidden layers*
- *Wrapping latent space as a sphere*

OBJECTIVE

Adding hidden layers increases the number of learnable parameters to better use non-linear operations in encoding/decoding. Spherical geometry of latent space forces the network to use these additional degrees of freedom more efficiently.

Let's dive deeper into the technical aspects of autoencoders and improve their internal representations to reach the levels required for the *MNIST* cognitive task.

- Increase the capacity of the network by introducing additional hidden layers
- Understand the effect of constraints in the geometry of latent space

Deeper autoencoder (2D)

The internal representation of shallow autoencoder with 2D latent space is similar to PCA, which shows that the autoencoder is not fully leveraging non-linear capabilities to model data. Adding capacity in terms of learnable parameters takes advantage of non-linear operations in encoding/decoding to capture non-linear patterns in data.

Adding hidden layers enables us to introduce additional parameters, either layerwise or depthwise. The same amount N of additional parameters can be added in a single layer or distributed among several layers. Adding several hidden layers reduces the compression/decompression ratio of each layer.

Build deeper autoencoder (2D)

Adding four hidden layers. The number of units per layer in the encoder is the following: 784 -> 392 -> 64 -> 2

The shallow autoencoder has a compression ratio of **784:2 = 392:1**. The first additional hidden layer has a compression ratio of **2:1**, followed by a hidden layer that sets the bottleneck compression ratio of **32:1**.

The choice of hidden layer size aims to reduce the compression rate in the bottleneck layer while increasing the count of trainable parameters. For example, if the compression rate of the first hidden layer doubles from **2:1** to **4:1**, the count of trainable parameters halves from 667K to 333K.

This deep autoencoder's performance may be further improved by adding additional hidden layers and by increasing the count of trainable parameters in each layer. These improvements have a diminishing return due to challenges associated with training under high parameter count and depth.

add a first hidden layer with 2x - 3x the input size. This size increase results in millions of parameters at the cost of longer training time.

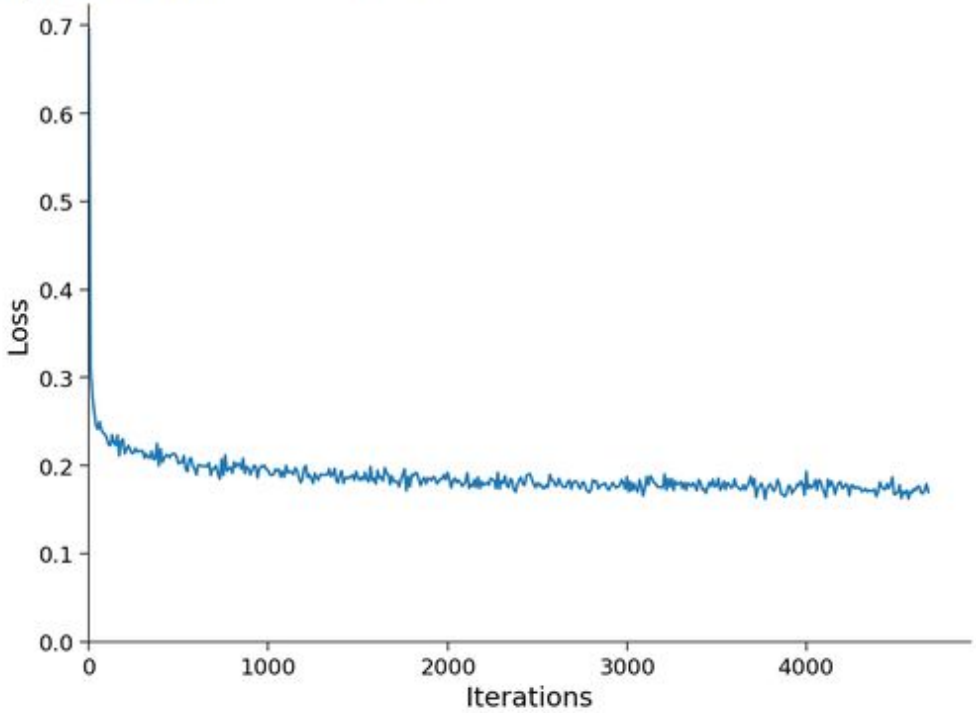
Weight initialization is particularly important in deep networks. The availability of large datasets and weight initialization likely drove the deep learning revolution of 2010.

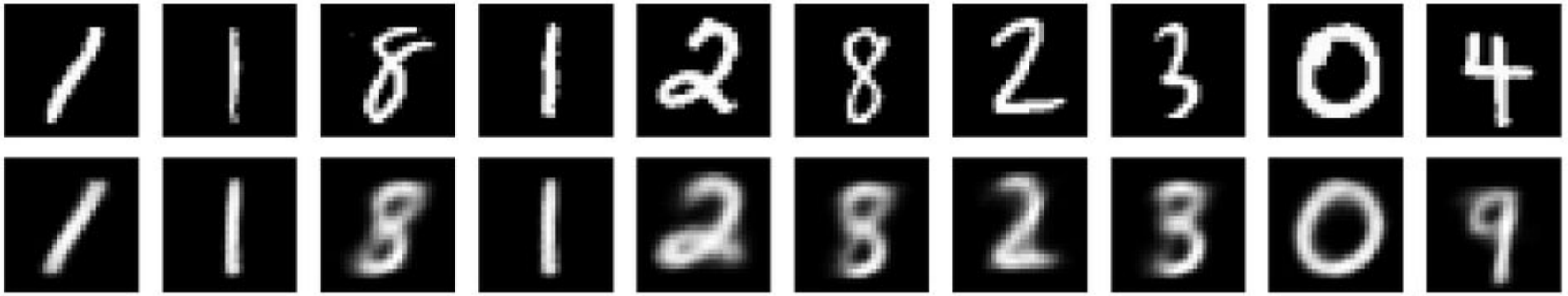
```
Autoencoder

 Sequential(
  (0): Linear(in_features=784, out_features=392, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=392, out_features=64, bias=True)
  (3): PReLU(num_parameters=1)
  (4): Linear(in_features=64, out_features=2, bias=True)
  (5): PReLU(num_parameters=1)
  (6): Linear(in_features=2, out_features=64, bias=True)
  (7): PReLU(num_parameters=1)
  (8): Linear(in_features=64, out_features=392, bias=True)
  (9): PReLU(num_parameters=1)
  (10): Linear(in_features=392, out_features=784, bias=True)
  (11): Sigmoid()
)

Encoder

 Sequential(
  (0): Linear(in_features=784, out_features=392, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=392, out_features=64, bias=True)
  (3): PReLU(num_parameters=1)
  (4): Linear(in_features=64, out_features=2, bias=True)
  (5): PReLU(num_parameters=1)
)

Decoder

 Sequential(
  (6): Linear(in_features=2, out_features=64, bias=True)
  (7): PReLU(num_parameters=1)
  (8): Linear(in_features=64, out_features=392, bias=True)
  (9): PReLU(num_parameters=1)
  (10): Linear(in_features=392, out_features=784, bias=True)
  (11): Sigmoid()
)
```

Train the autoencoder

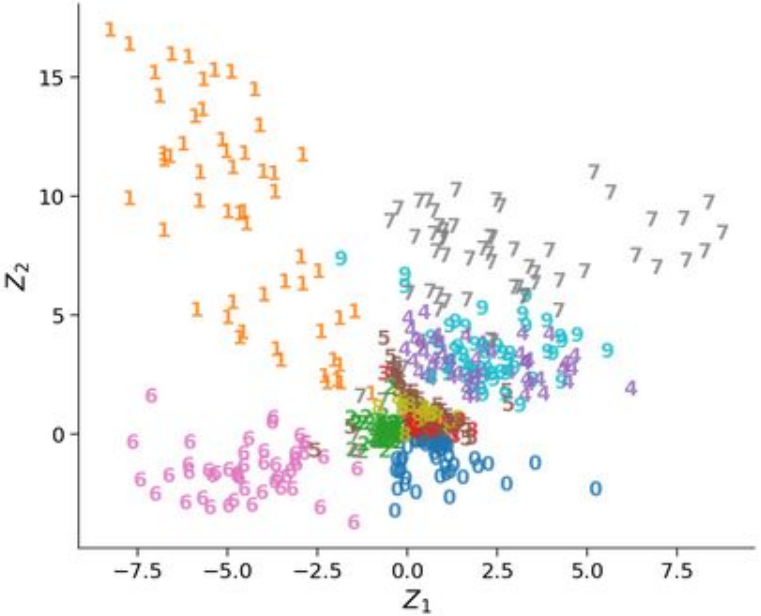
How the internal representation successfully captures additional digit classes. The encoder map shows well-separated clusters that correspond to the associated digits in the decoder grid. The decoder grid also shows that the network is robust to digit skewness, i.e., digits leaning to the left or the right are recognized in the same digit class.

Epoch	Loss train	Loss test
1/10	0.2065	0.2057
2/10	0.1948	0.1946
3/10	0.1871	0.1871
4/10	0.1827	0.1832
5/10	0.1800	0.1807
6/10	0.1778	0.1785
7/10	0.1764	0.1774
8/10	0.1749	0.1761
9/10	0.1740	0.1755
10/10	0.1727	0.1741

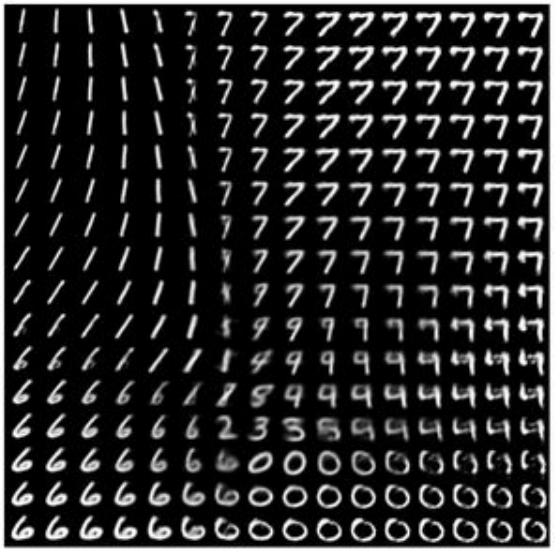




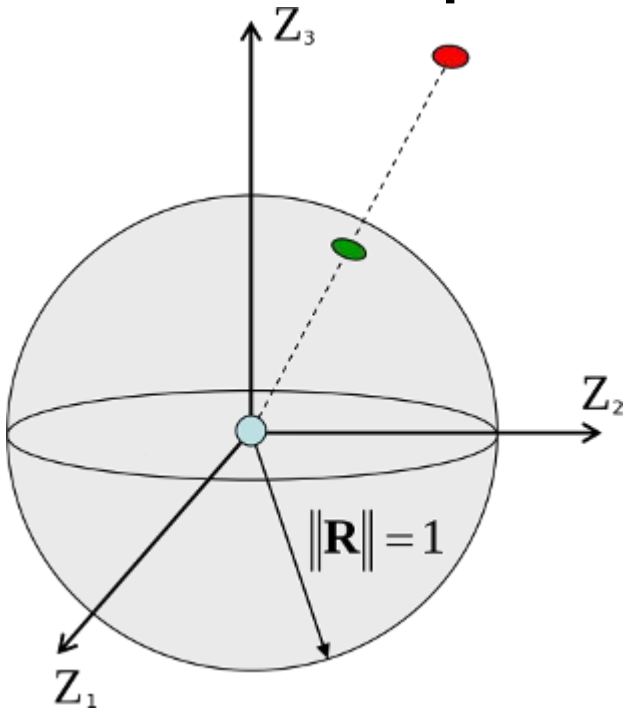
Encoder map



Decoder grid



Spherical latent space



The previous architecture generates representations that typically spread in different directions from coordinate $(z1,z2)=(0,0)$. This effect is due to the initialization of weights distributed randomly around 0.

Adding a third unit to the bottleneck layer defines a coordinate $(z1,z2,z3)$ in 3D space. The latent space from such a network will still spread out from $(z1,z2,z3)=(0,0,0)$.

Collapsing the latent space on the surface of a sphere removes the possibility of spreading indefinitely from the origin $(0,0,0)$ in any direction since this will eventually lead back to the origin. This constraint generates a representation that fills the surface of the sphere.

Projecting to the surface of the sphere is implemented by dividing the coordinates $(z1,z2,z3)$ by their $\mathcal{L}2$ norm.

$$(z_1, z_2, z_3) \longmapsto (s_1, s_2, s_3) = (z_1, z_2, z_3)/\|(z_1, z_2, z_3)\|_2 = (z_1, z_2, z_3)/\sqrt{z_1^2 + z_2^2 + z_3^2}$$

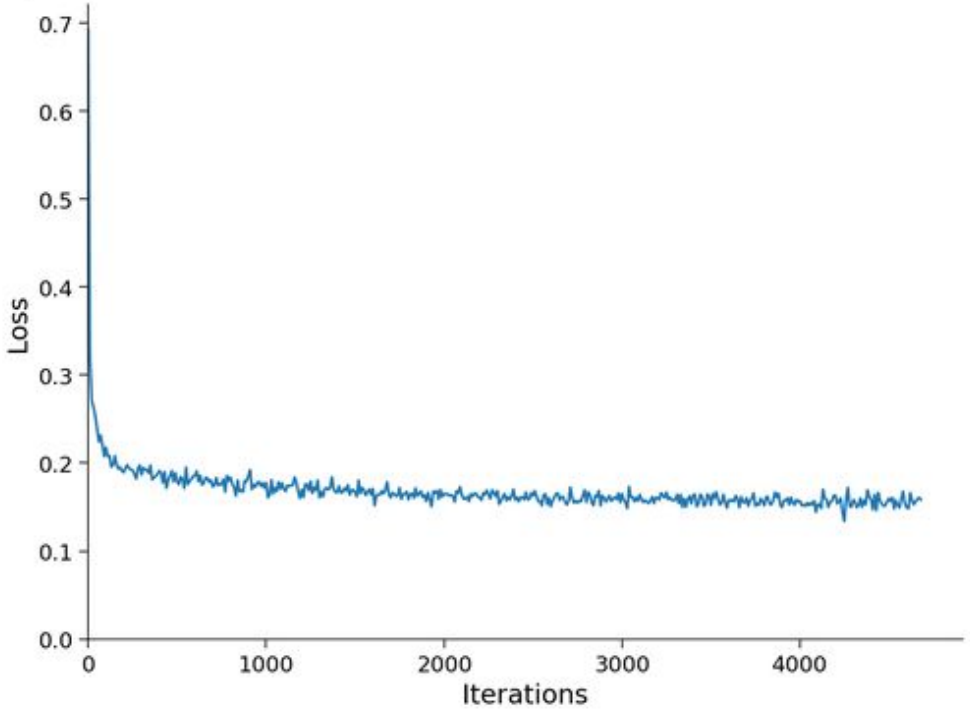
This mapping projects to the surface of the $S2$sphere with unit radius.

Build and train autoencoder (3D)

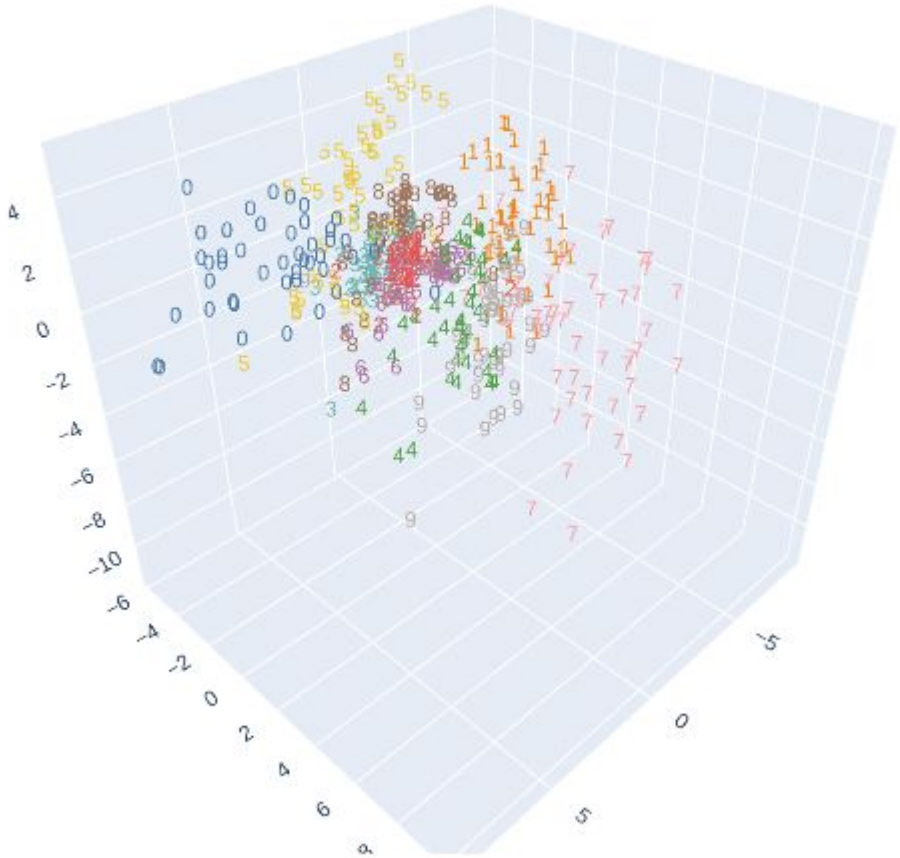
We start by adding one unit to the bottleneck layer and visualize the latent space in 3D.

```
Autoencoder

 Sequential(
  (0): Linear(in_features=784, out_features=392, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=392, out_features=96, bias=True)
  (3): PReLU(num_parameters=1)
  (4): Linear(in_features=96, out_features=3, bias=True)
  (5): PReLU(num_parameters=1)
  (6): Linear(in_features=3, out_features=96, bias=True)
  (7): PReLU(num_parameters=1)
  (8): Linear(in_features=96, out_features=392, bias=True)
  (9): PReLU(num_parameters=1)
  (10): Linear(in_features=392, out_features=784, bias=True)
  (11): Sigmoid()
)
```

Epoch	Loss train	Loss test
1/10	0.1825	0.1830
2/10	0.1733	0.1742
3/10	0.1675	0.1684
4/10	0.1639	0.1650
5/10	0.1616	0.1629
6/10	0.1595	0.1612
7/10	0.1582	0.1600
8/10	0.1567	0.1588
9/10	0.1557	0.1580
10/10	0.1548	0.1576



Visualize the latent space in 3D



Build deep autoencoder (2D) with latent spherical space

We now constrain the latent space to the surface of a sphere $S2$.

Instructions:

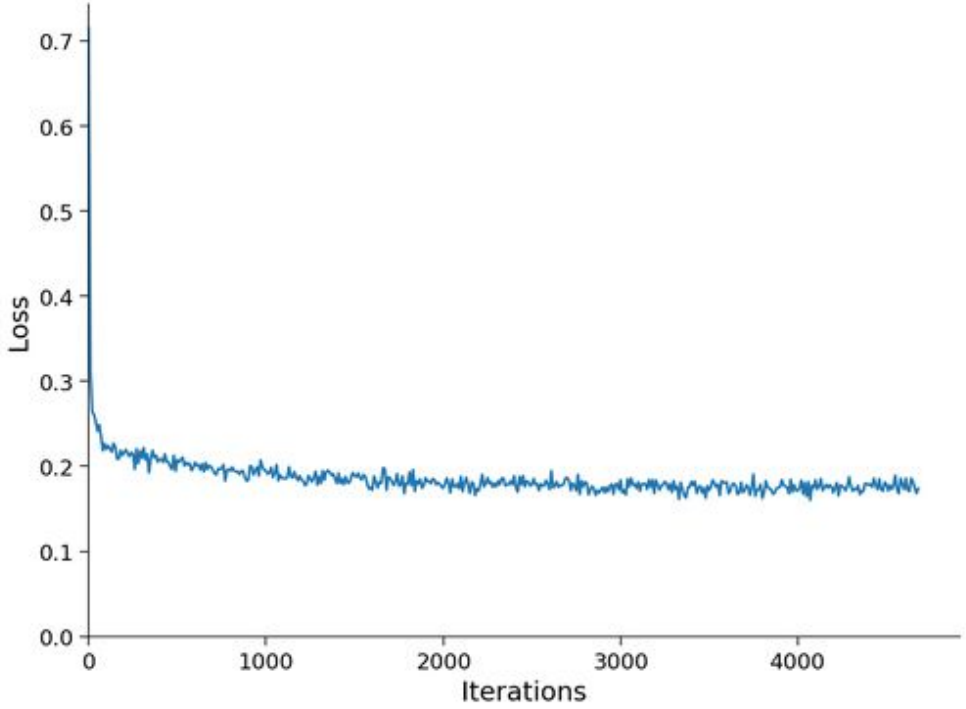
- Add the custom layer NormalizeLayer after the bottleneck layer
- Adjust the definitions of encoder and decoder
- Experiment with keyword show_text=False for plot_latent_3d

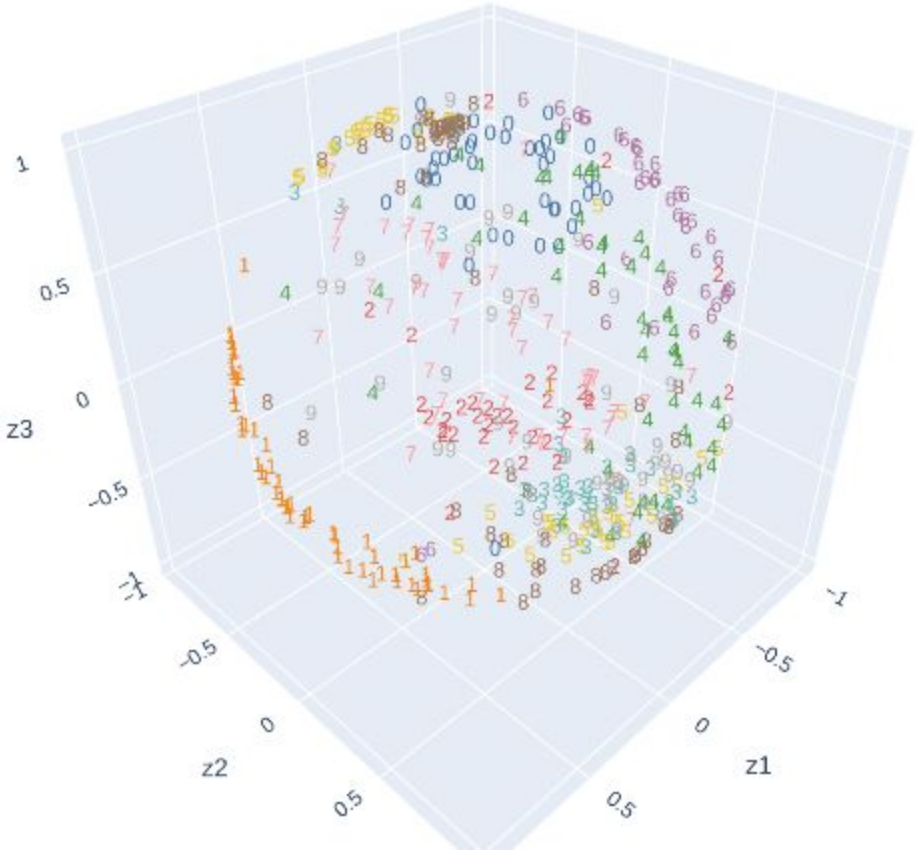
```
Autoencoder

 Sequential(
  (0): Linear(in_features=784, out_features=392, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=392, out_features=96, bias=True)
  (3): PReLU(num_parameters=1)
  (4): Linear(in_features=96, out_features=3, bias=True)
  (5): PReLU(num_parameters=1)
  (6): NormalizeLayer()
  (7): Linear(in_features=3, out_features=96, bias=True)
  (8): PReLU(num_parameters=1)
  (9): Linear(in_features=96, out_features=392, bias=True)
  (10): PReLU(num_parameters=1)
  (11): Linear(in_features=392, out_features=784, bias=True)
  (12): Sigmoid()
)

Encoder

 Sequential(
  (0): Linear(in_features=784, out_features=392, bias=True)
  (1): PReLU(num_parameters=1)
  (2): Linear(in_features=392, out_features=96, bias=True)
  (3): PReLU(num_parameters=1)
  (4): Linear(in_features=96, out_features=3, bias=True)
  (5): PReLU(num_parameters=1)
  (6): NormalizeLayer()
)

Decoder

 Sequential(
  (7): Linear(in_features=3, out_features=96, bias=True)
  (8): PReLU(num_parameters=1)
  (9): Linear(in_features=96, out_features=392, bias=True)
  (10): PReLU(num_parameters=1)
  (11): Linear(in_features=392, out_features=784, bias=True)
  (12): Sigmoid()
)
```

Train the autoencoder

Train the network for n_epochs=10 epochs with batch_size=128 and observe how loss raises again and is comparable to the model with 2D latent space.

1/10	0.2023	0.2015
2/10	0.1924	0.1921
3/10	0.1854	0.1849
4/10	0.1815	0.1813
5/10	0.1780	0.1782
6/10	0.1771	0.1774
7/10	0.1750	0.1757
8/10	0.1743	0.1748
9/10	0.1733	0.1737
10/10	0.1744	0.1745




z3
1
0.5
0
-0.5
-1
z2
-0.5
0
0.5
z1
-1
-0.5
0
0.5

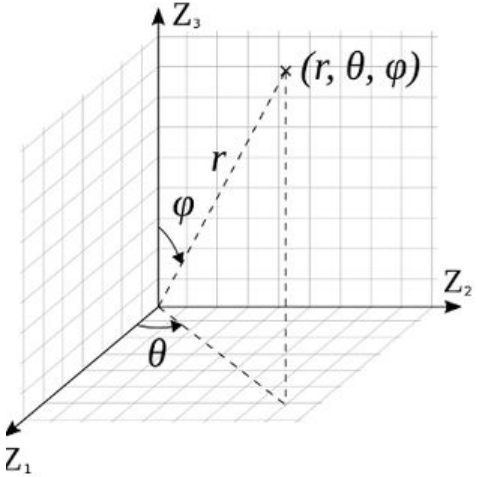
Visualize latent space on surface of S_2

The 3D coordinates (s_1, s_2, s_3) on the surface of the unit sphere S_2 can be mapped to spherical coordinates (r, θ, ϕ), as follows:

$$r = \sqrt{s_1^2 + s_2^2 + s_3^2}$$

$$\phi = \arctan \frac{s_2}{s_1}$$

$$\theta = \arccos \frac{s_3}{r}$$

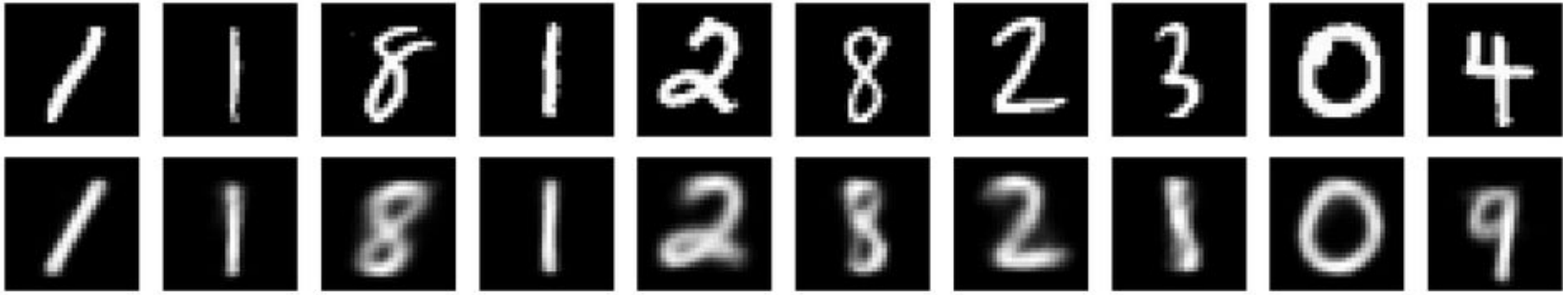


Food for thought

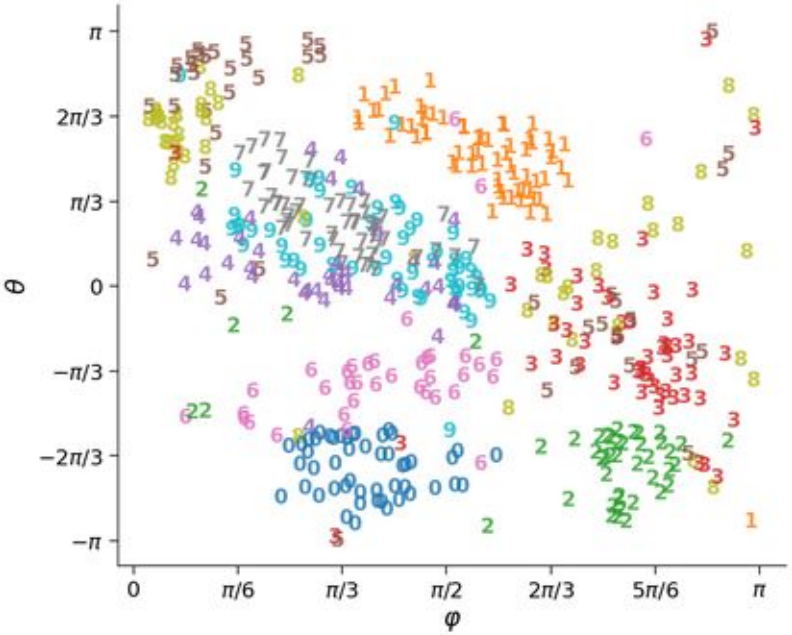
What is the domain (numerical range) spanned by (θ,ϕ)?

We return to a 2D representation since the angles (θ,ϕ) are the only degrees of freedom on the surface of the sphere.

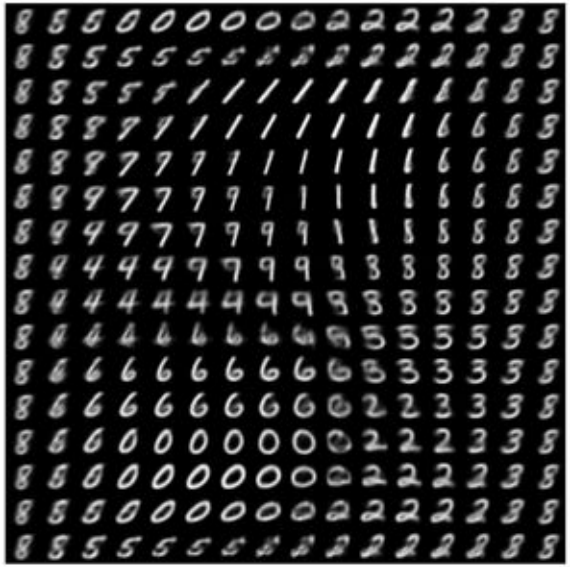
Task: Check the numerical range of the plot axis to help identify θ and ϕ, and visualize the unfolding of the 3D plot.



Encoder map



Decoder grid



Summary

We learned two techniques to improve representation capacity: adding a few hidden layers and projecting latent space on the sphere S^2. The expressive power of autoencoder improves with additional hidden layers.

Projecting latent space on the surface of S^2 spreads out digits classes in a more visually pleasing way but may not always produce a lower loss.

Deep autoencoder architectures have rich internal representations to deal with sophisticated tasks such as the MNIST cognitive task.

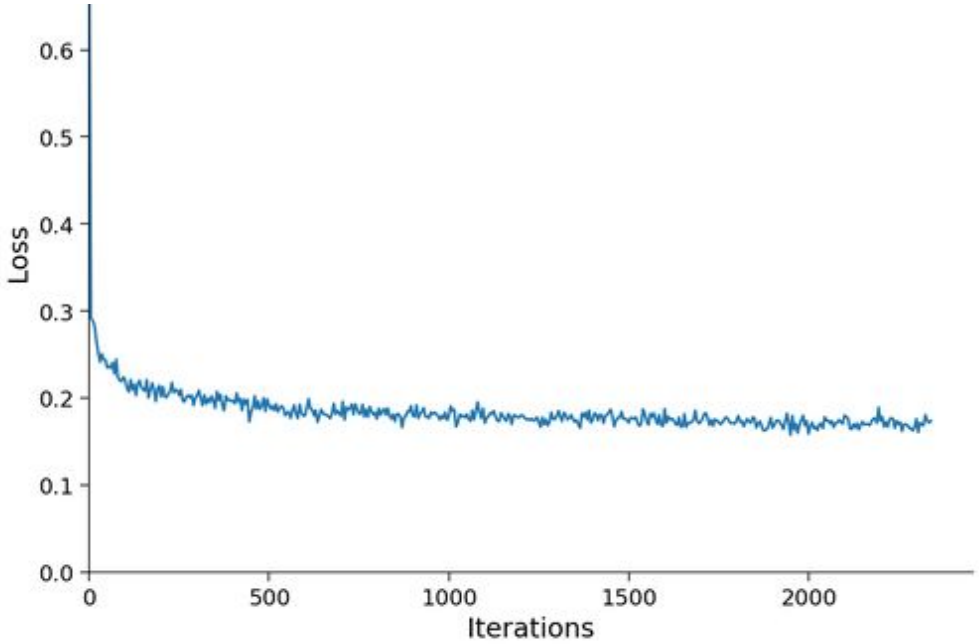
We now have powerful tools to explore how simple algorithms build robust models of the world by capturing relevant data patterns.

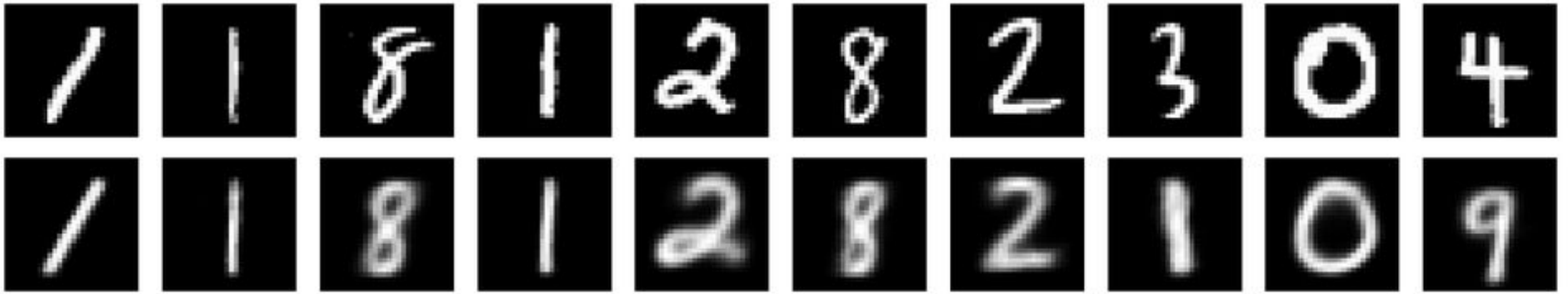
Tutorial #2 bonus Explanations

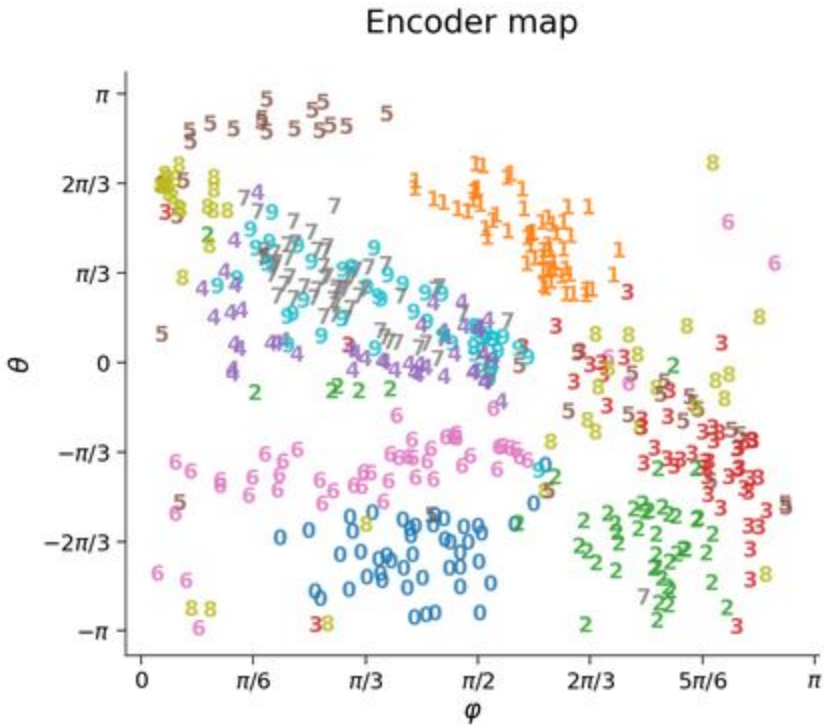
Deep and thick autoencoder

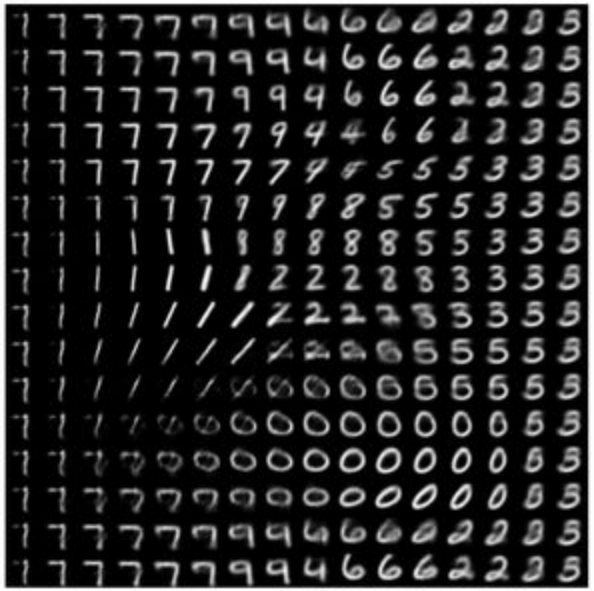
Expand the first hidden layer to double the input size, followed by compression to half the input size leading to 3.8M parameters.

```
0         1230880         Linear(in_features=784, out_features=1568, bias=True)
1         1       PReLU(num_parameters=1)
2         615048  Linear(in_features=1568, out_features=392, bias=True)
3         1       PReLU(num_parameters=1)
4         37728   Linear(in_features=392, out_features=96, bias=True)
5         1       PReLU(num_parameters=1)
6         291     Linear(in_features=96, out_features=3, bias=True)
7         1       PReLU(num_parameters=1)
8         0       NormalizeLayer()
9         384     Linear(in_features=3, out_features=96, bias=True)
10        1       PReLU(num_parameters=1)
11        38024   Linear(in_features=96, out_features=392, bias=True)
12        1       PReLU(num_parameters=1)
13        616224  Linear(in_features=392, out_features=1568, bias=True)
14        1       PReLU(num_parameters=1)
15        1230096         Linear(in_features=1568, out_features=784, bias=True)
16        0       Sigmoid()

Total:    3768682
```


Loss
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
500
1000
1500
2000
Iterations




Encoder map
π
2π/3
π/3
0
−π/3
−2π/3
−π
θ
0
π/6
π/3
π/2
2π/3
5π/6
π
φ


Decoder grid

Tutorial #3
Explanations

Autoencoder applications

- How do autoencoders with rich internal representations perform on the MNIST cognitive task?
- How do autoencoders perceive unseen digit classes?
- How does ANN image encoding differ from human vision?

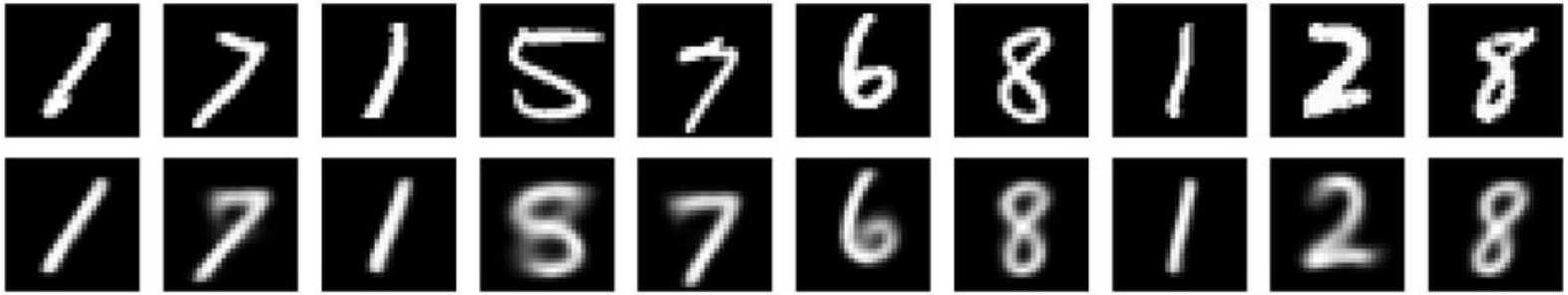
In this tutorial, you will:

- Analyze how autoencoders perceive transformed data (added noise, occluded parts, and rotations), and how that evolves with short re-train sessions
- Use autoencoders to visualize unseen digit classes
- Understand visual encoding for fully connected ANN autoencoders

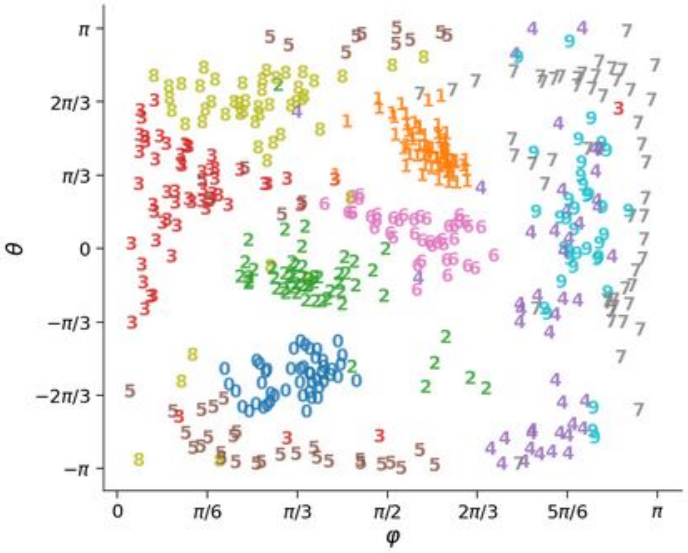
Download a pre-trained model

Setting the boolean parameter s2=True specifies the model with projection onto the S2 sphere.

Experiments run from the identical initial conditions by resetting the autoencoder to the reference state.



Encoder map



Decoder grid

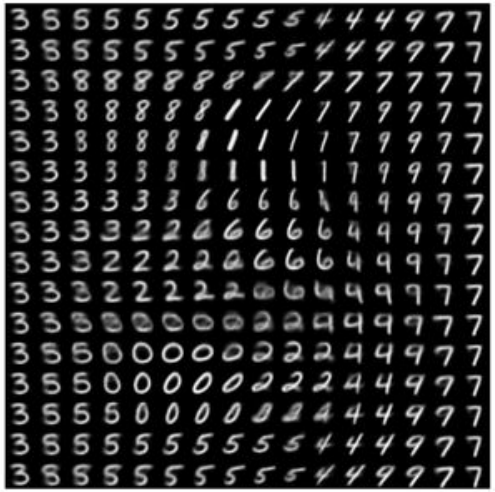


Image noise

Removing noise added to images is often showcased in dimensionality reduction techniques.

We observe that autoencoders trained with noise-free images output noise-free images when receiving noisy images as input. However, the reconstructed images will be different from the original images (without noise) since the added noise maps to different coordinates in latent space.

The ability to map noise-free and noisy versions to similar regions in latent space is known as robustness or invariance to noise. How can we build such functionality into the autoencoder?

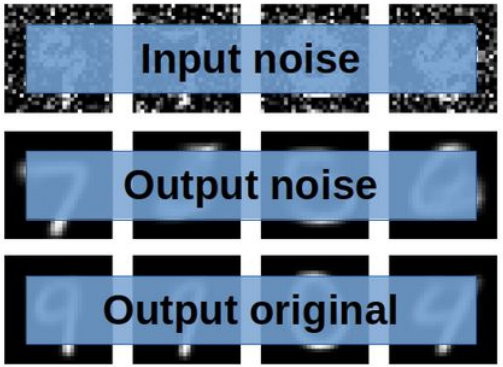
The solution is to train the autoencoder with noise-free and noisy versions mapping to the noise-free version. A faster alternative is to re-train the autoencoder for few epochs with noisy images. These short training sessions fine-tune the weights to map noisy images to their noise-free versions from similar latent space coordinates.

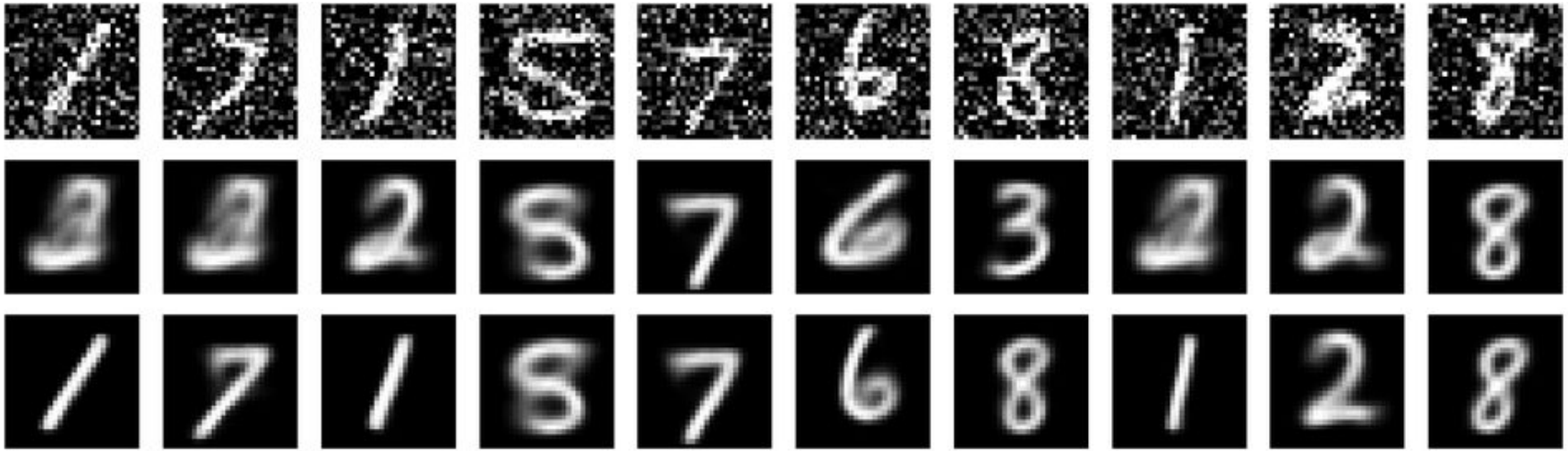
Reconstructions before fine-tuning

Verify that an autoencoder trained on clean images will output clean images from noisy inputs. Visualize this by plotting three rows:

- Top row with noisy images inputs
- Middle row with reconstructions of noisy images
- Bottom row with reconstructions of the original images (noise-free)

The bottom row helps identify samples with reconstruction issues before adding noise. This row shows the baseline reconstruction quality for these samples rather than the original images. (Why?)



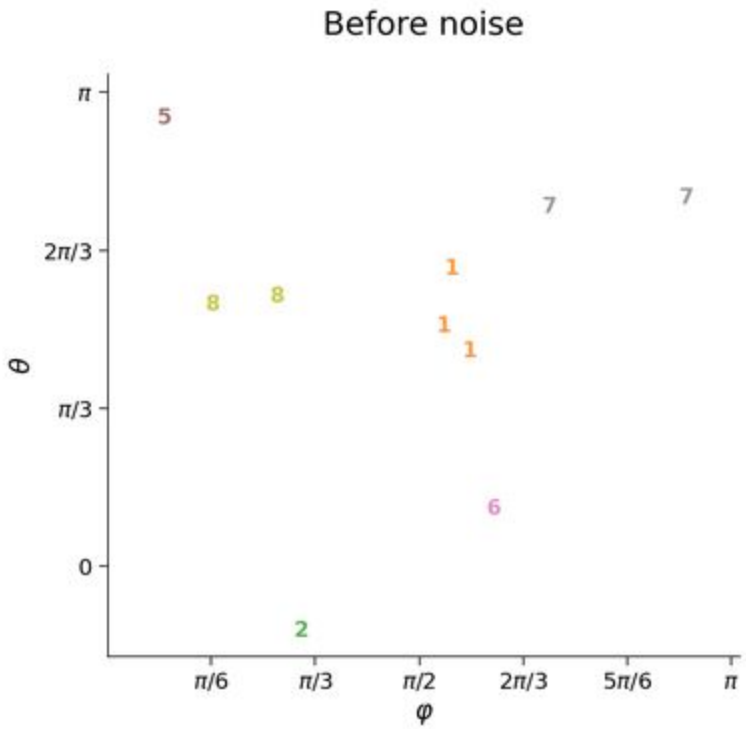


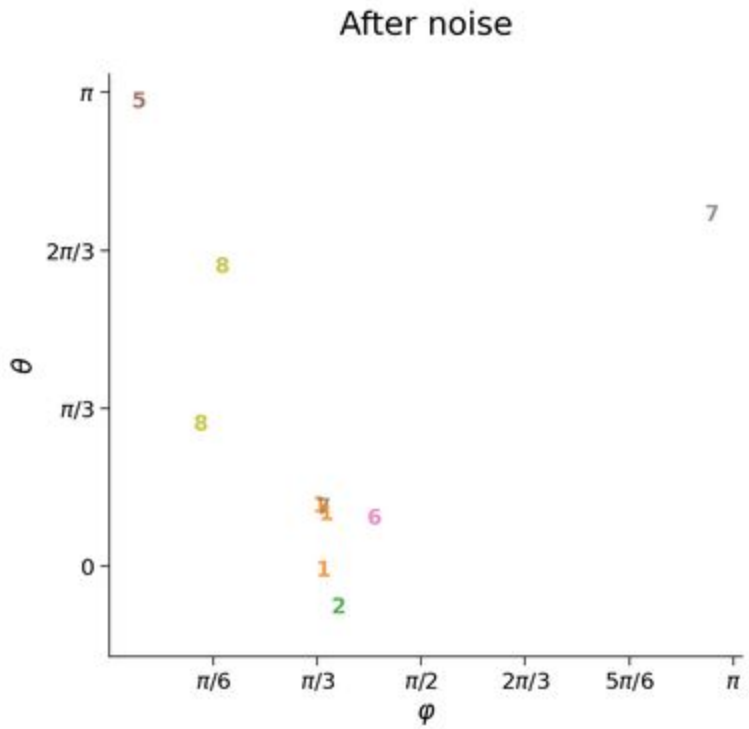
Latent space before fine-tuning

We investigate the origin of reconstruction errors by looking at how adding noise to input affects latent space coordinates. The decoder interprets significant coordinate changes as different digits.

The function plot_latent_ab compares latent space coordinates for the same set of samples between two conditions. Here, display coordinates for the ten samples adding noise:

- The left plot shows the coordinates of the original samples (noise-free)
- The plot on the right shows the new coordinates after adding noise

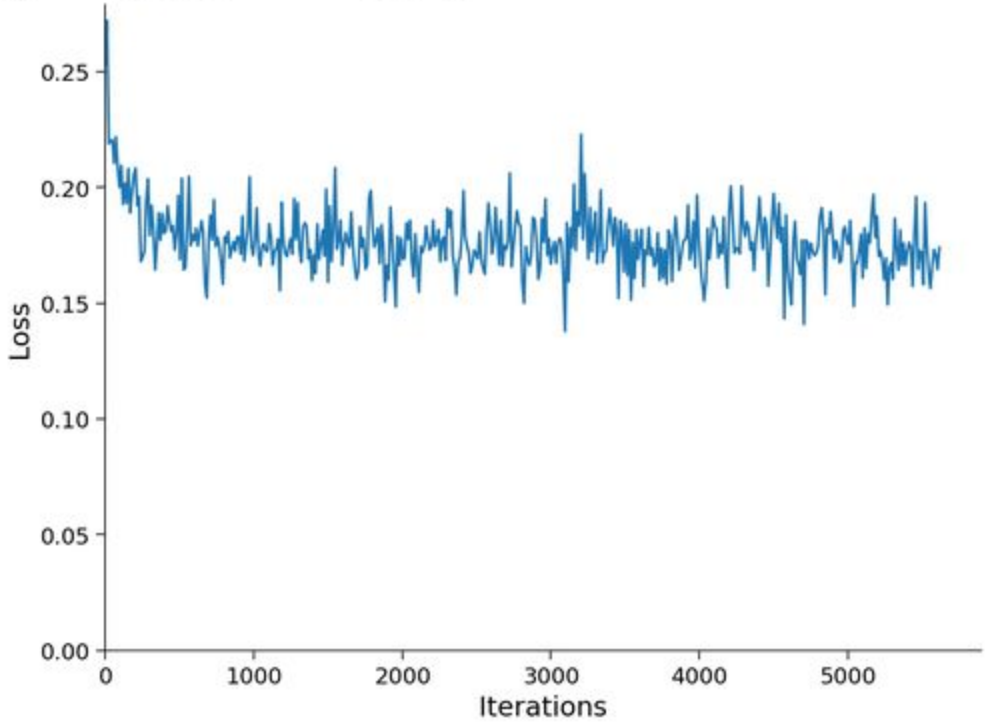

Before noise
π
2π/3
π/3
0
θ
π/6
π/3
π/2
2π/3
5π/6
π
φ
5
7
7
1
8
8
1
1
6
2

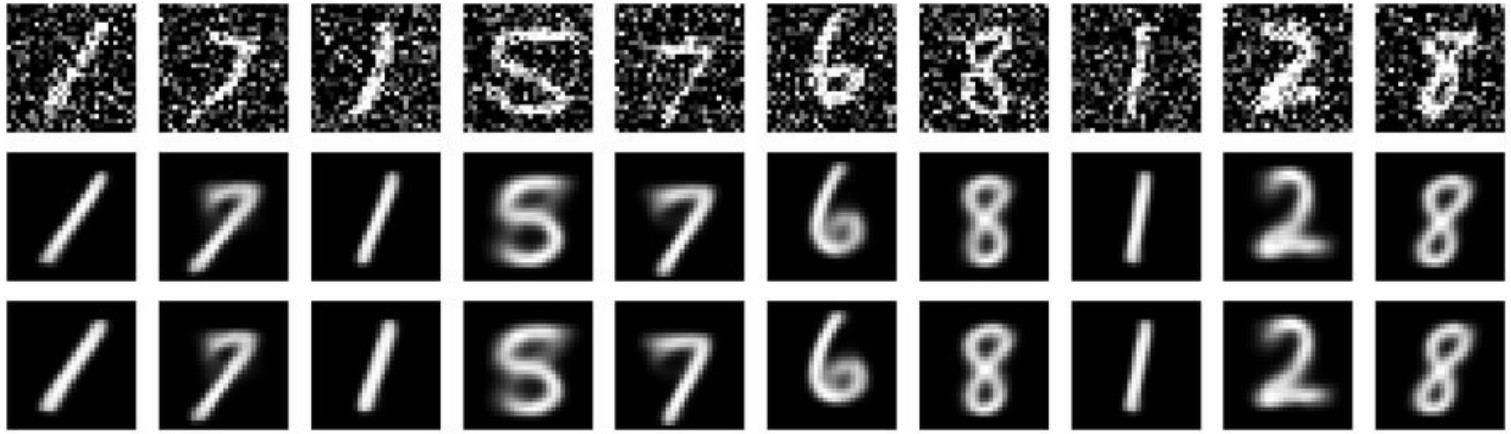

After noise
π
2π/3
π/3
0
θ
π/6
π/3
π/2
2π/3
5π/6
π
φ
5
7
8
8
6
1
2

Fine-tuning the autoencoder with noisy images

Re-train the autoencoder with noisy images on the input and original (noise-free) images on the output, and regenerate the previous plots. We now see that both noisy and noise-free images match similar locations in latent space. The network denoises the input with a latent-space representation that is more robust to noise.

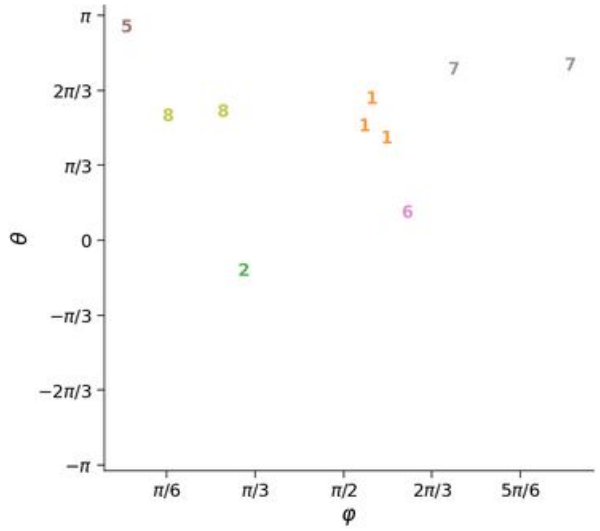
Epoch	Loss train	Loss test
1/3	0.1746	0.1755
2/3	0.1730	0.1740
3/3	0.1736	0.1746

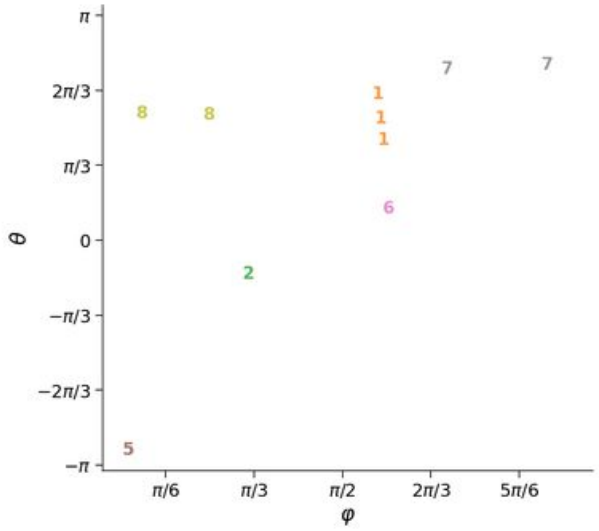




Before fine-tuning

After fine-tuning





W3D5_pod31

Global latent space shift

The new latent space representation is more robust to noise and may result in a better internal representation of the dataset. We verify this by inspecting the latent space with clean images before and after fine-tuning with noisy images.

Fine-tuning the network with noisy images causes a domain shift in the dataset, i.e., a change in the distribution of images since the dataset was initially composed of noise-free images. Depending on the task and the extent of changes during re-train, (number of epochs, optimizer characteristics, etc.), the new latent space representation may become less well adapted to the original data as a side-effect. How could we address domain shift and improve both noisy and noise-free images?

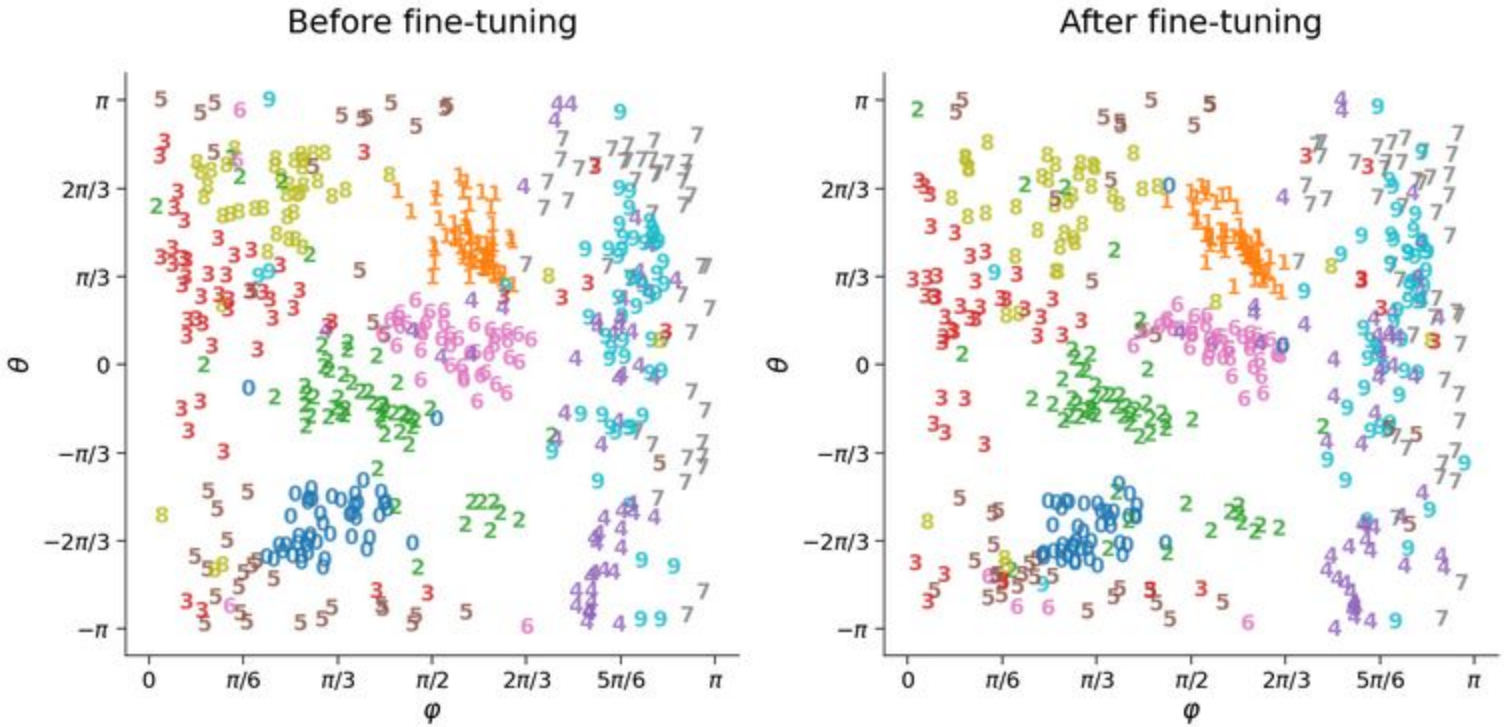

Before fine-tuning
After fine-tuning
θ
φ
π
2π/3
π/3
0
−π/3
−2π/3
−π
0
π/6
π/3
π/2
2π/3
5π/6
π

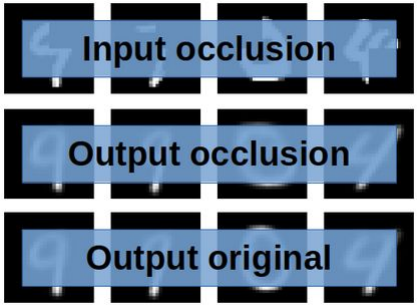
Image occlusion

Investigate the effects of image occlusion.

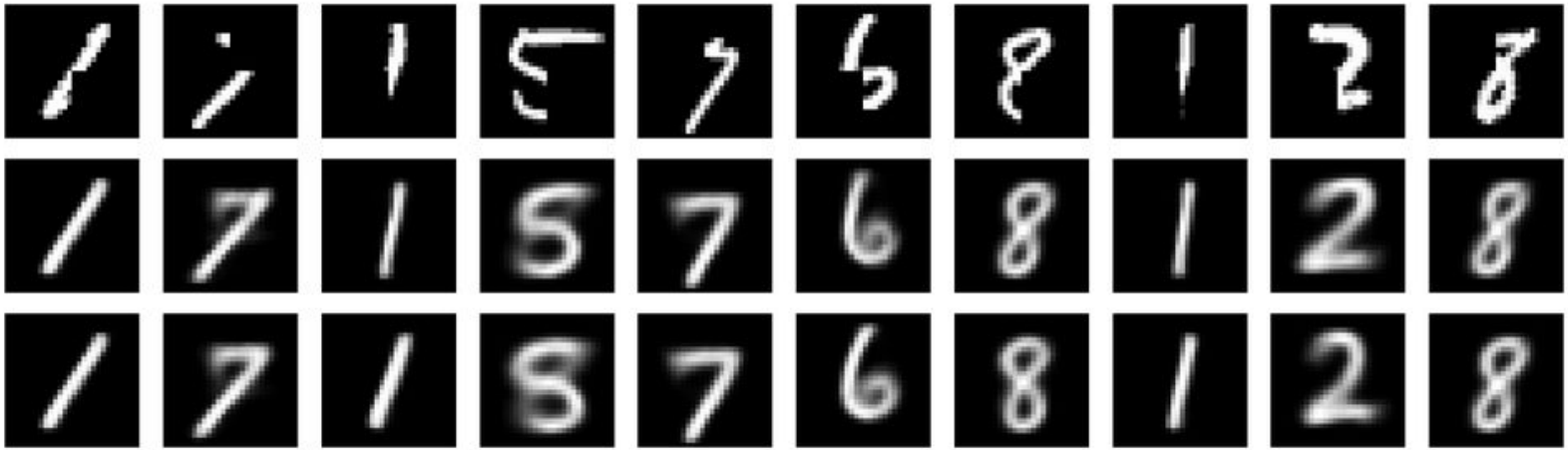
Expect the autoencoder to reconstruct complete images since the train set does not contain occluded images (right?).

Visualize this by plotting three rows:

- Top row with occluded images
- Middle row with reconstructions of occluded images
- Bottom row with reconstructions of the original images

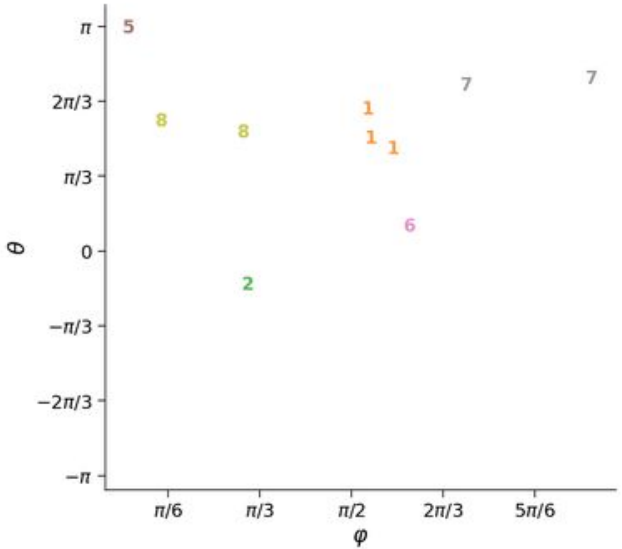


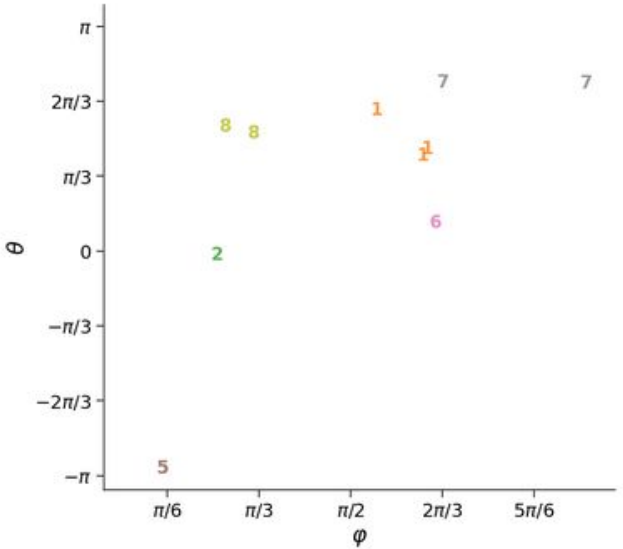
Investigate the source of this issue by looking at the representation of partial images in latent space and how it adjusts after fine-tuning.



Before occlusion

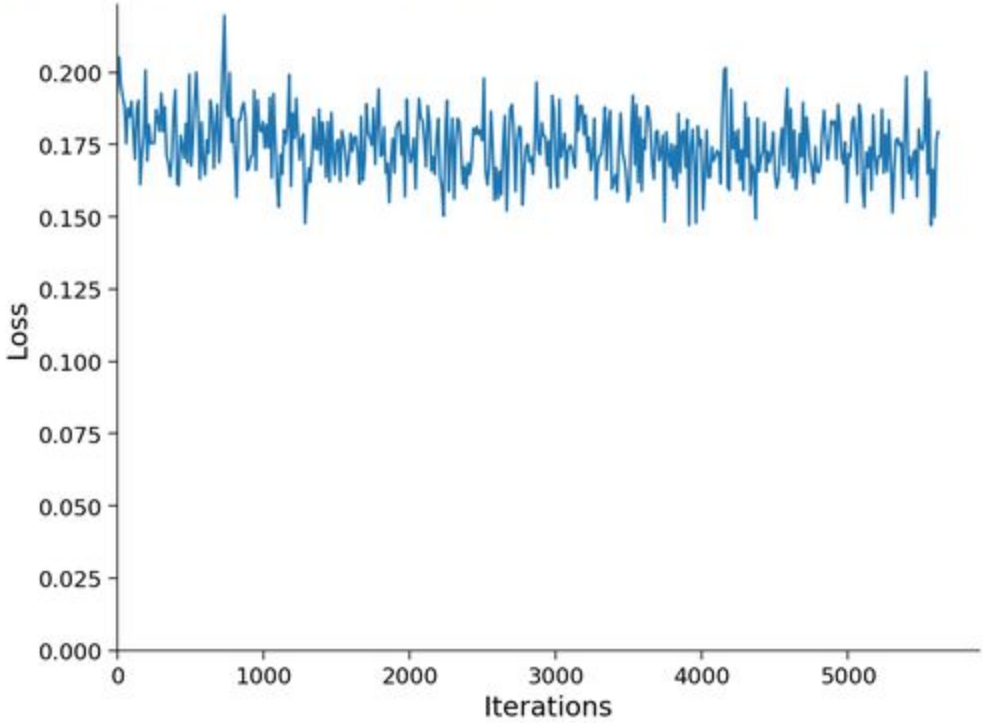
After occlusion

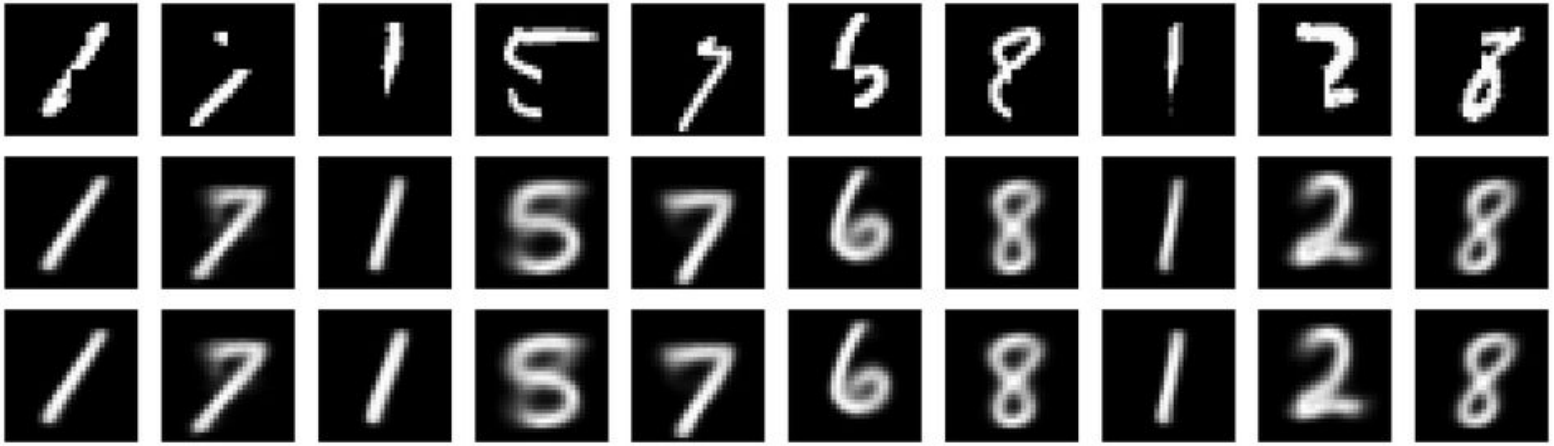




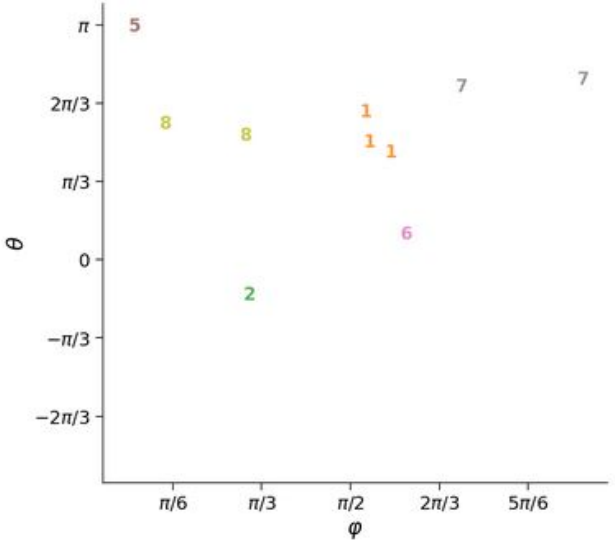
W3D5_pod31

Epoch	Loss train	Loss test
1/3	0.1723	0.1739
2/3	0.1719	0.1733
3/3	0.1709	0.1725

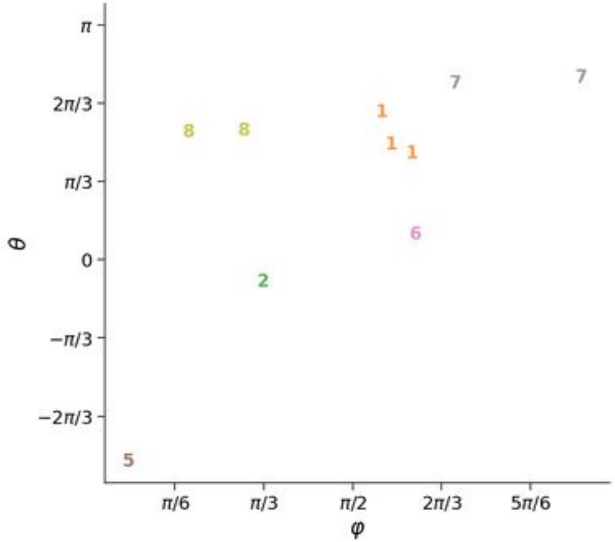




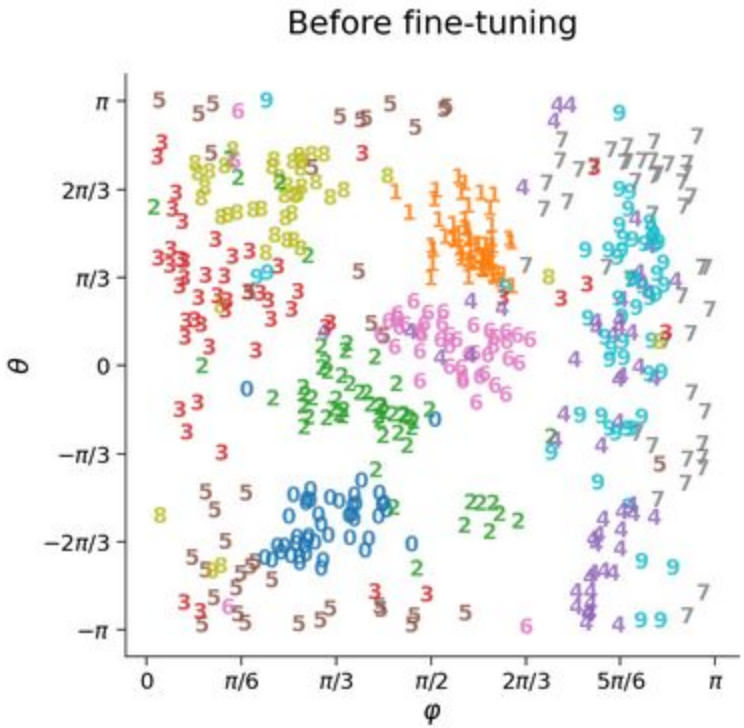
Before fine-tuning



After fine-tuning



W3D5_pod31


Before fine-tuning
π
2π/3
π/3
0
−π/3
−2π/3
−π
θ
0
π/6
π/3
π/2
2π/3
5π/6
π
φ

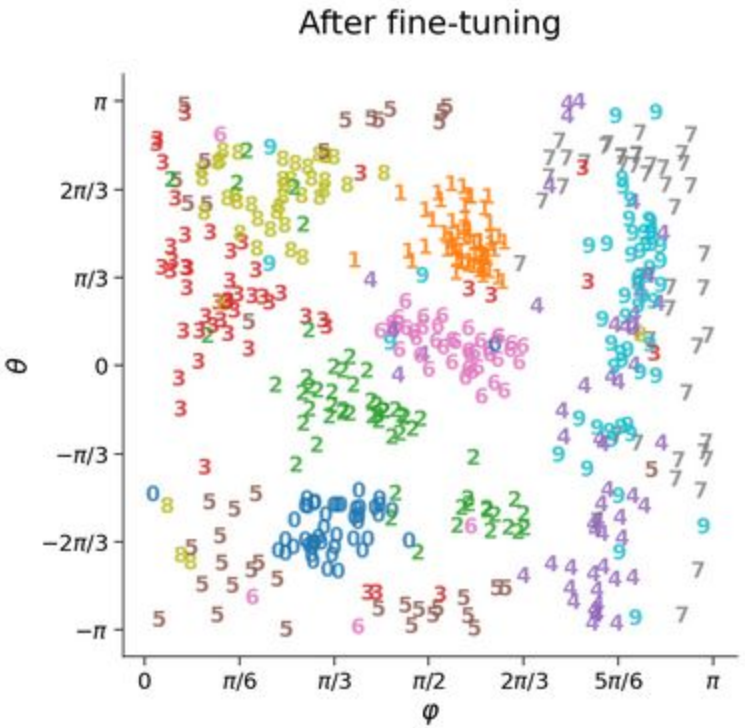
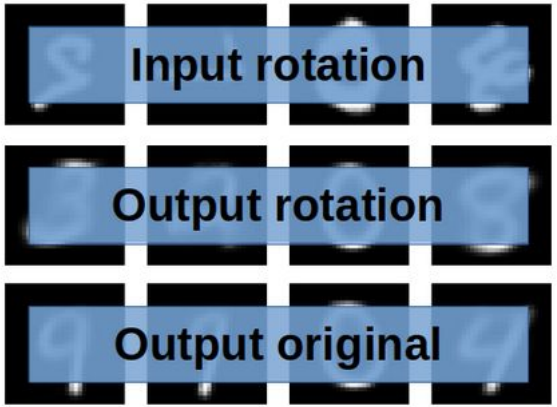

After fine-tuning
π
2π/3
π/3
0
−π/3
−2π/3
−π
θ
0
π/6
π/3
π/2
2π/3
5π/6
π
φ

Image rotation

Look at the effect of image rotation in latent space coordinates. This task is arguably more challenging since it may require a complete rewrite of image reconstruction.

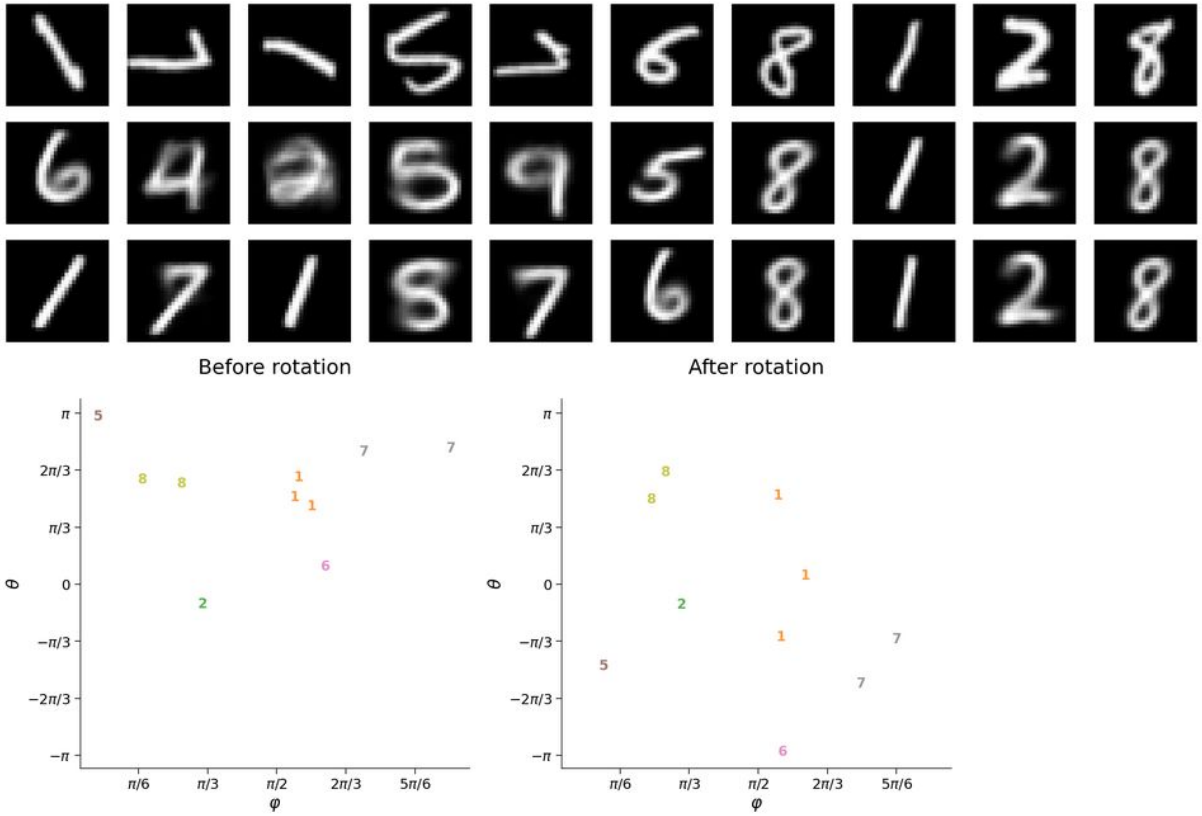
We visualize this by plotting three rows:

- Top row with rotated images
- Middle row with reconstructions of rotated images
- Bottom row with reconstructions of the original images

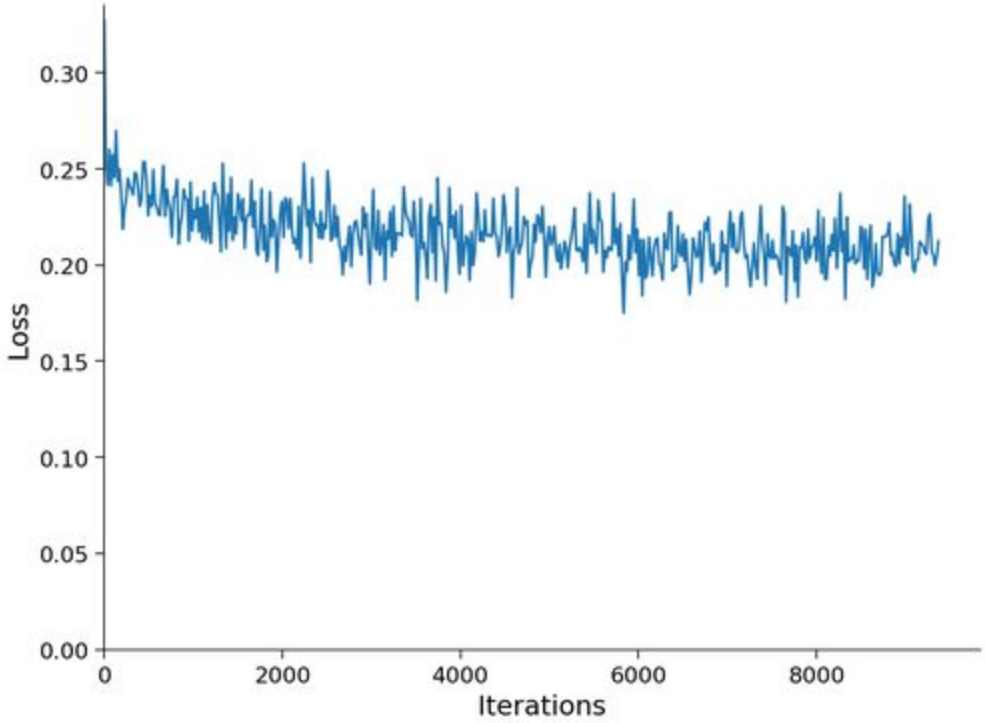


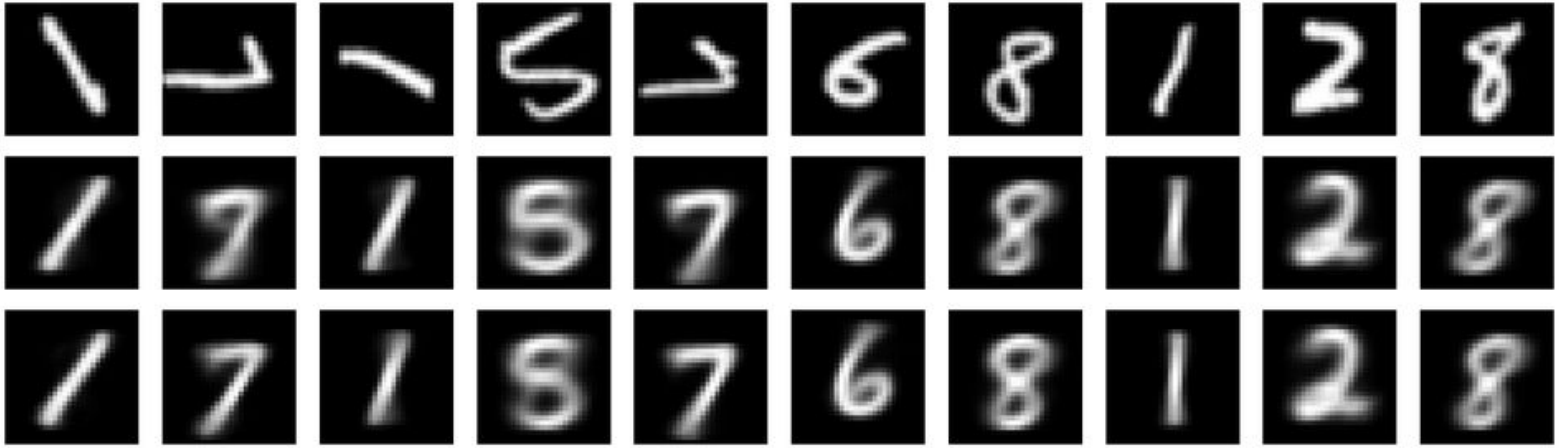
We investigate the source of this issue by looking at the representation of rotated images in latent space and how it adjusts after fine-tuning.

Before fine-tuning

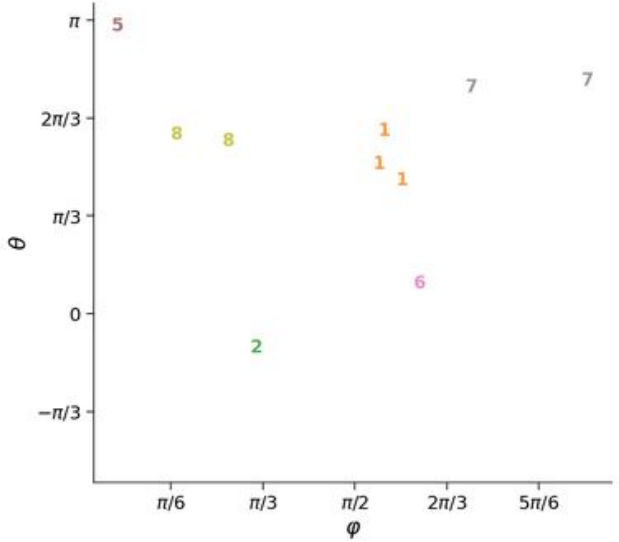


Epoch	Loss train	Loss test
1/5	0.2228	0.2224
2/5	0.2163	0.2164
3/5	0.2099	0.2103
4/5	0.2084	0.2089
5/5	0.2048	0.2061

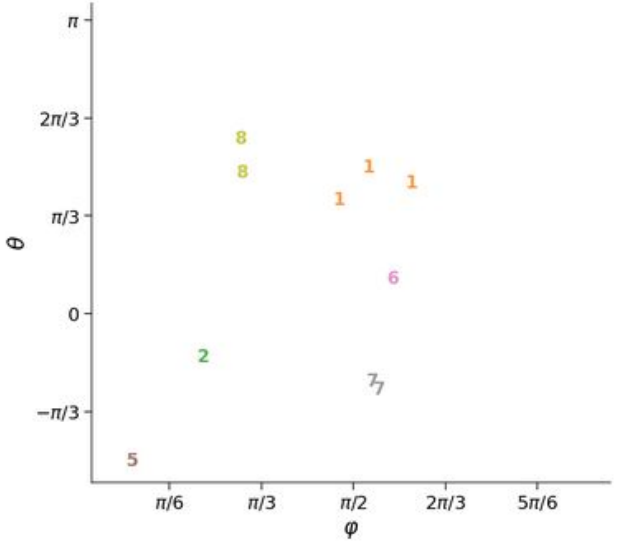




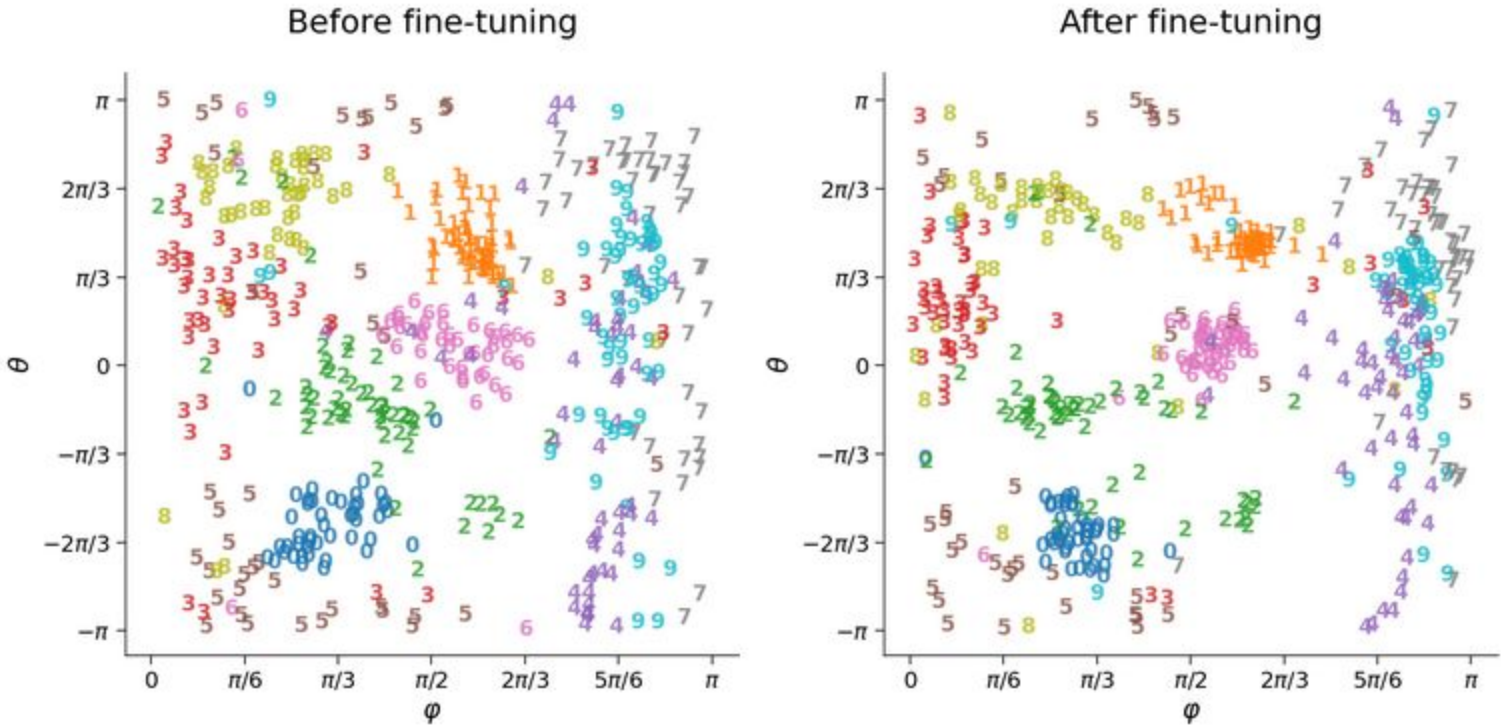
Before fine-tuning



After fine-tuning



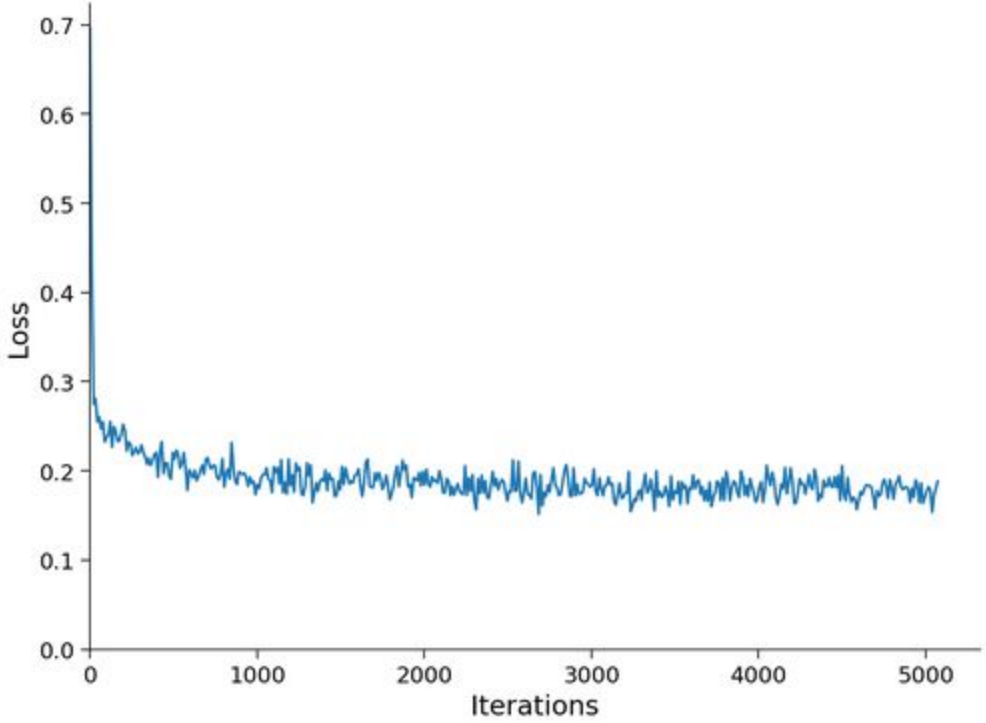
W3D5_pod31

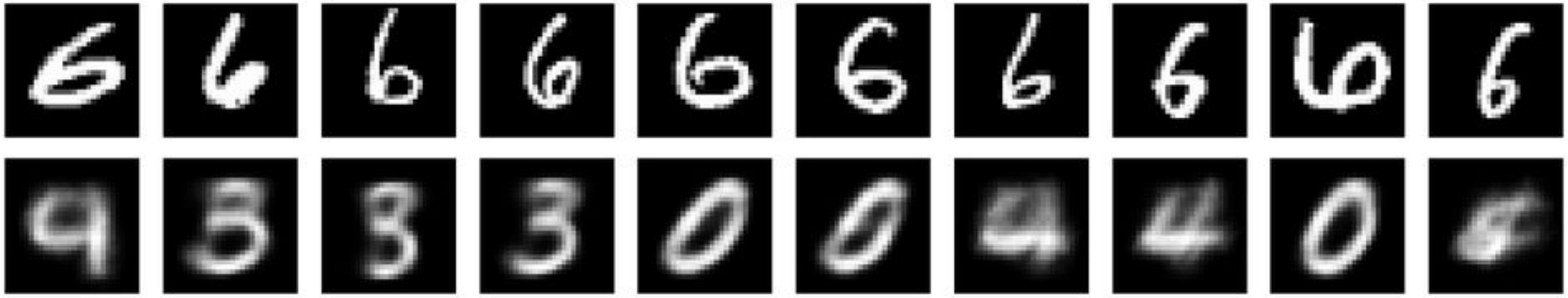

Before fine-tuning
After fine-tuning
θ
φ
π
2π/3
π/3
0
−π/3
−2π/3
−π
π/6
π/2
5π/6

What would digit "6" look like if we had never seen it before?

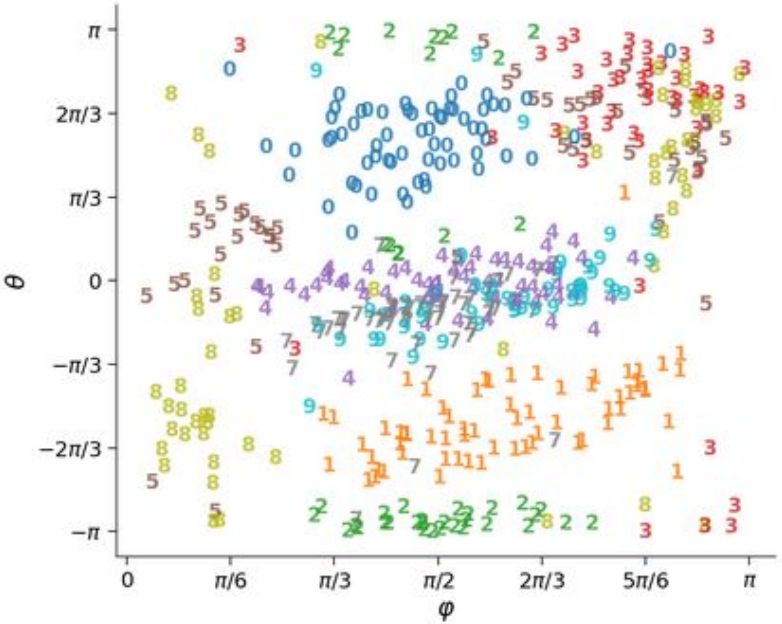
Train the autoencoder from scratch without digit class 6 and visualize reconstructions from digit 6.

Epoch	Loss train	Loss test
1/3	0.1901	0.1888
2/3	0.1808	0.1800
3/3	0.1802	0.1795

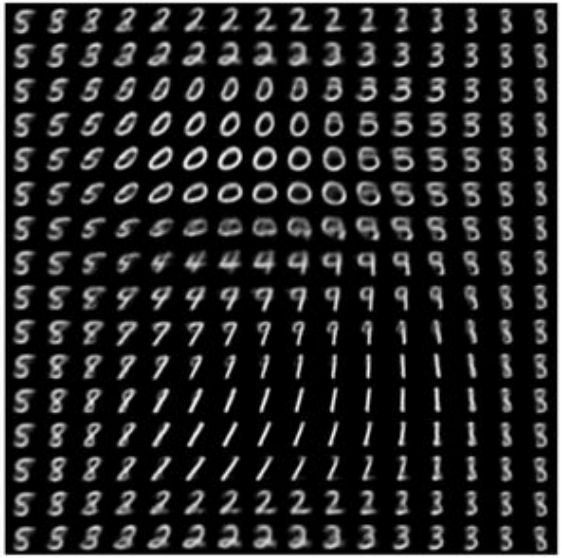




Encoder map



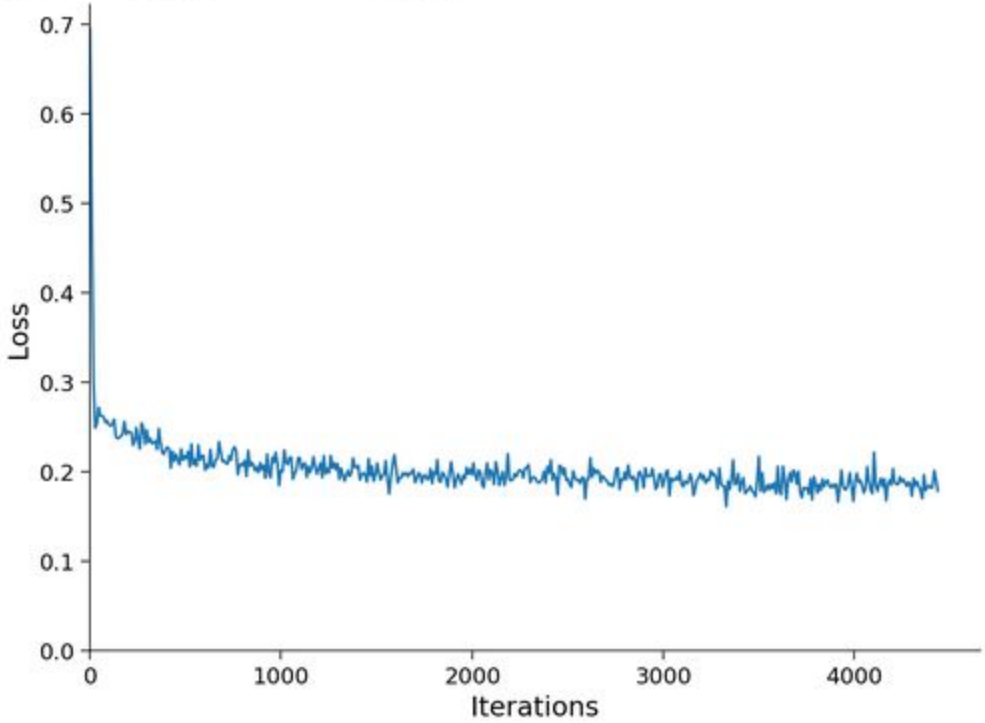
Decoder grid

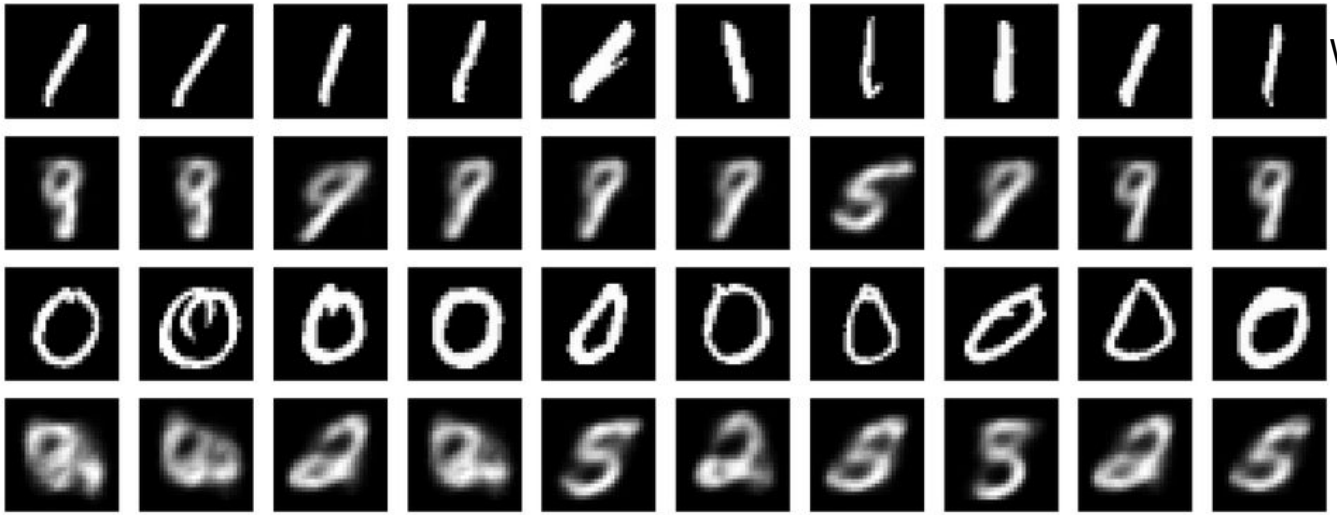


Removing the most dominant digit classes

Digit classes 0 and 1 are dominant in the sense that these occupy large areas of the decoder grid, compared to other digit classes that occupy very little generative space. How will latent space change when removing the two most dominant digit classes? Will latent space re-distribute evenly among remaining classes or choose another two dominant classes?

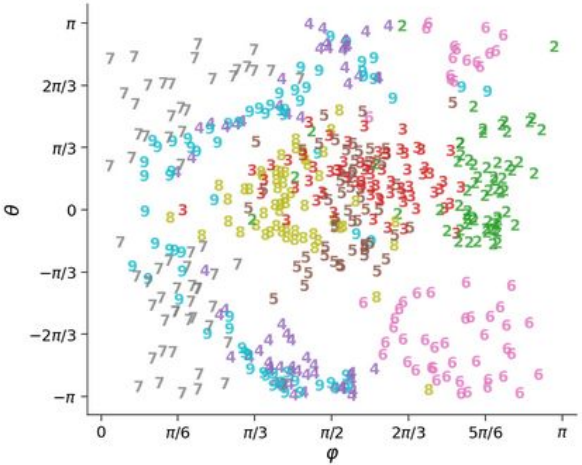
Epoch	Loss train	Loss test
1/3	0.1992	0.1992
2/3	0.1910	0.1914
3/3	0.1849	0.1853



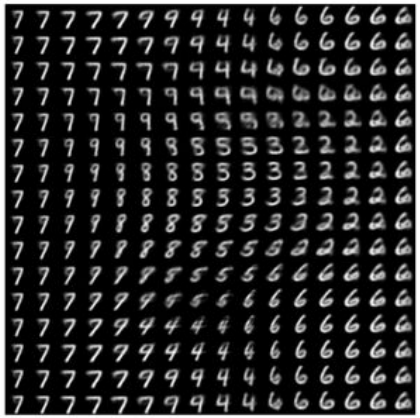


W3D5_pod31

Encoder map



Decoder grid

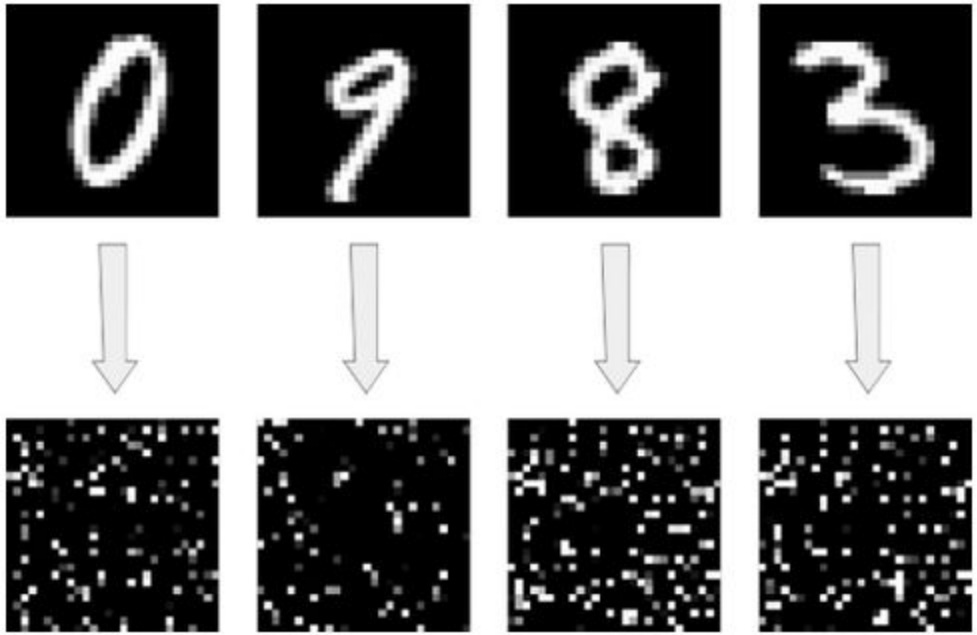


ANNs? Same but different!

"Same same but different" is an expression used to express differences between supposedly similar subjects. Investigate a fundamental difference in how fully-connected ANNs process visual information compared to human vision.

However, there is a crucial aspect of ANN processing already encoded in the vectorization of images. This network architecture completely ignores the relative position of pixels. To illustrate this, we show that learning proceeds just as well with shuffled pixel locations.

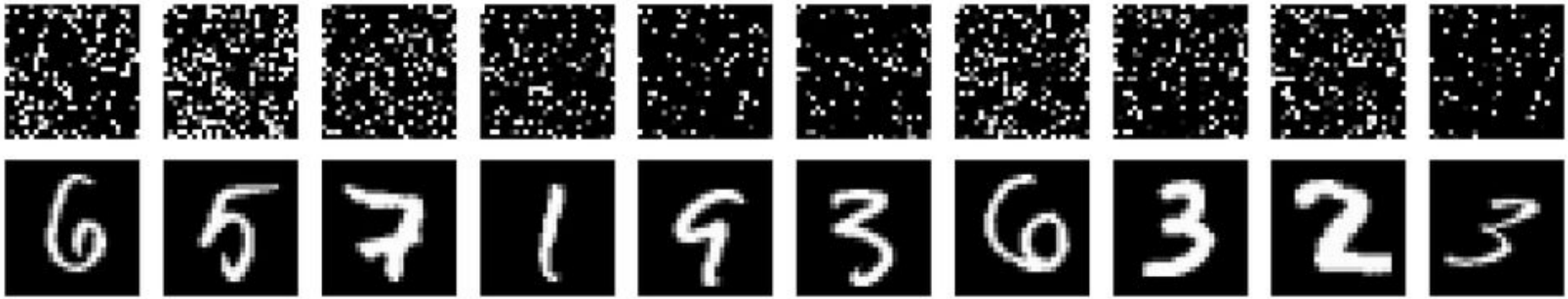
First, we obtain a reversible shuffle map stored in shuffle_image_idx used to shuffle image pixels randomly.



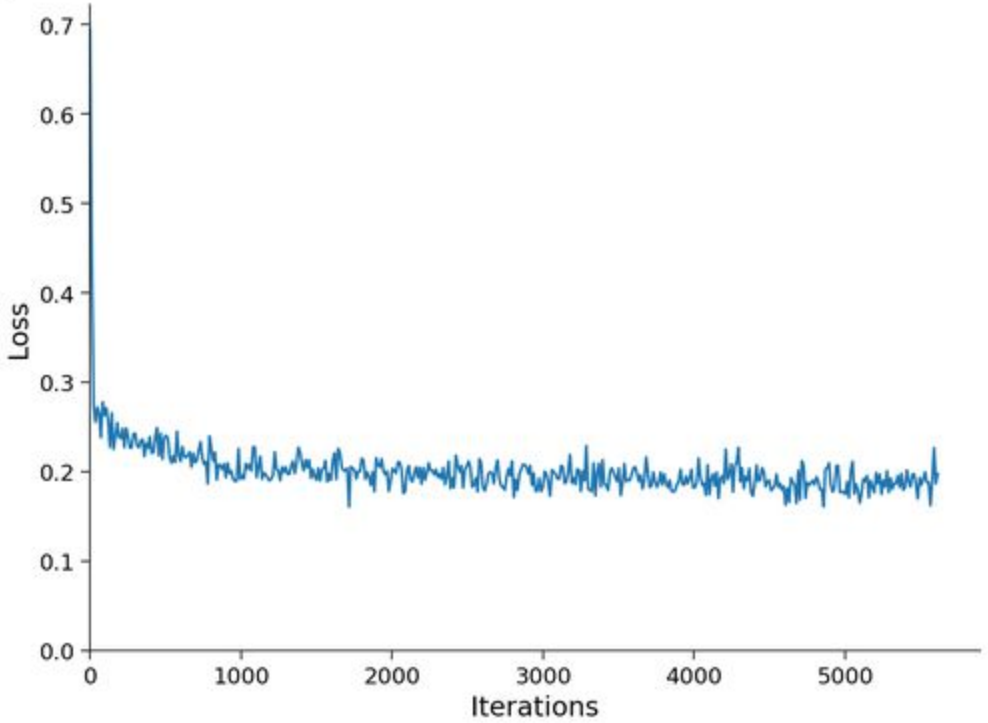
The unshuffled image set input_shuffle is recovered as follows:

input_shuffle[:, shuffle_rev_image_idx]]

First, we set up the reversible shuffle map and visualize a few images with shuffled and unshuffled pixels, followed by their noisy versions.



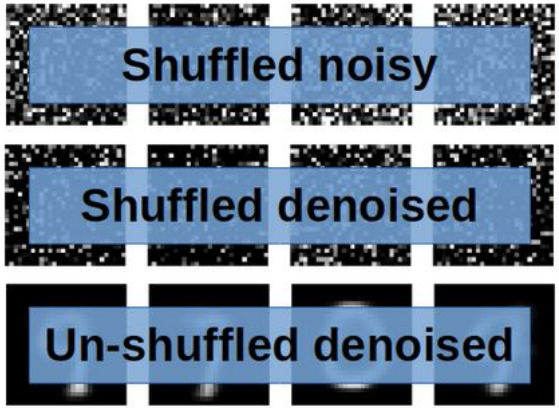
Epoch	Loss train	Loss test
1/3	0.1965	0.1968
2/3	0.1904	0.1915
3/3	0.1868	0.1875



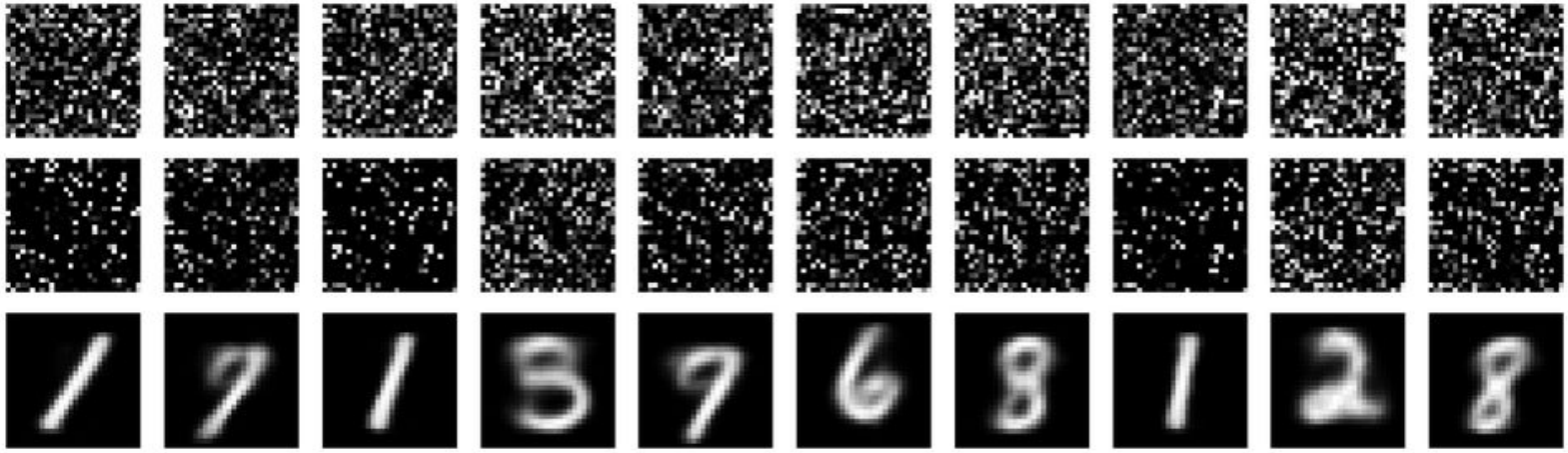
Finally, visualize reconstructions and latent space representation with the trained model.

We visualize reconstructions by plotting three rows:

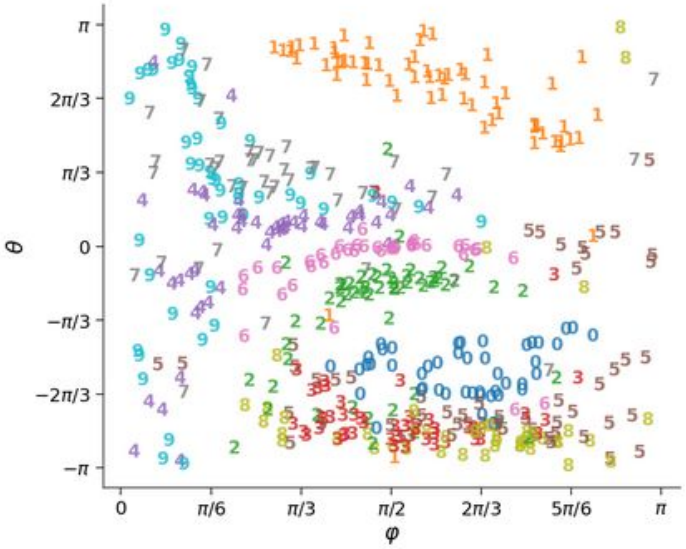
- *Top row with shuffled noisy images*
- *Middle row with reconstructions of shuffled denoised images*
- *Bottom row with unshuffled reconstructions of denoised images*



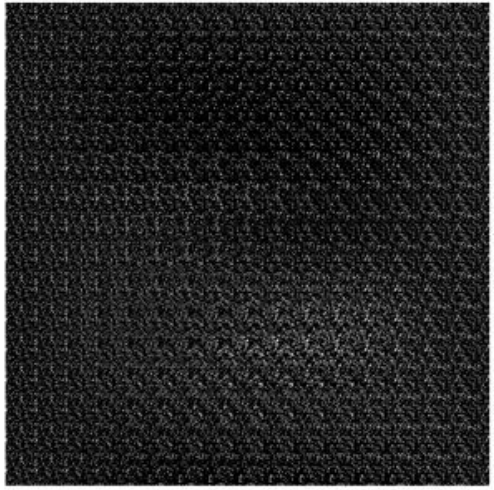
We obtain the same organization in the encoder map as before. Sharing similar internal representations confirms the network to ignore the relative position of pixels. The decoder grid is different than before since it generates shuffled images.



Encoder map



Decoder grid



Summary

Used to model certain aspects of cognition or even extend them to biologically plausible architectures - autoencoders of spiking neurons.

Autoencoders trained in learning by doing tasks such as compression/decompression, removing noise, etc. can uncover rich lower-dimensional structure embedded in structured images and other cognitively relevant data.

The data domain seen during training imprints a "cognitive bias". Such bias is related to the concept (What you see is all there is) coined by Daniel Kahneman in psychology.

Additional applications of autoencoders to neuroscience, also see here how to replicate the input-output relationship of real networks of neurons with autoencoders.