L12 Auto-Encoders

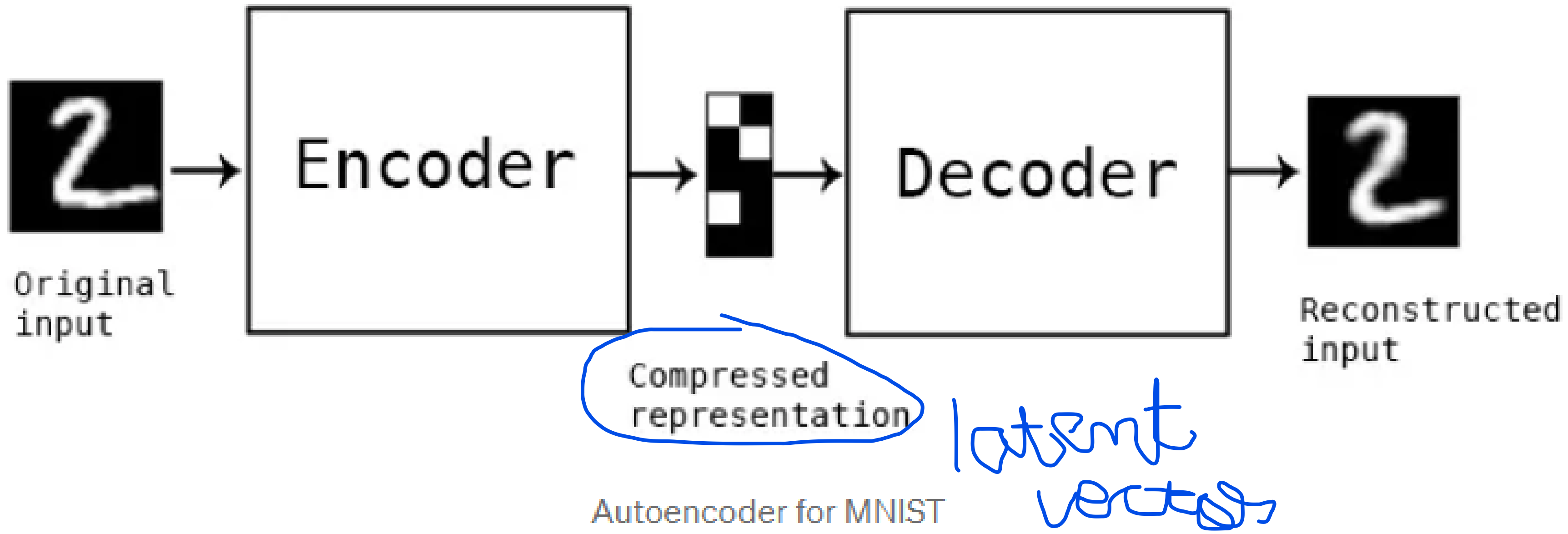
Auto-Encoders

- With advances in AI, we see that GPUs, TPUs, and other faster storage and processing mechanisms have successfully processed vast amounts of data in the shortest possible time
- We also need to consider how to compress and store data.
- Autoencoders is a technique that allows to compress available information in less space.
- Autoencoders are the self-supervised models that can learn to compress the input data efficiently.
- Self-supervised models: These models are trained as supervised machine learning models and during inference, they work as unsupervised models So they are called self-supervised models.

Auto-Encoders

- Autoencoders are a specific type of feedforward neural networks where the input is the same as the output.
- They compress the input into a lower-dimensional code and then reconstruct the output from this representation.
- The code is a compact “summary” or “compression” of the input, also called the latent-space representation or bottleneck or code layer
- Autoencoders are trained the same way as ANNs via backpropagation.
- Note: We can think of the latent vector as a code for an image, which is where the terms encode/decode come from.
- When we use autoencoders, we actually do not care about the output itself, but rather we care about the vector constructed in the middle (the latent vector).
- This vector is important because it is a representation of the input image and with this representation, we could potentially do several tasks like reconstructing the original image.

Auto-Encoder for MNIST



Autoencoder for MNIST

Example of the input/output image from the MNIST dataset to an autoencoder
Encoder maps the input data to a lower-dimensional representation
Decoder maps the lower-dimensional representation back to the original dimensionality.

Auto-Encoders - Components

An autoencoder consists of 3 components: encoder, code and decoder. The encoder compresses the input and produces the code, the decoder then reconstructs the input only using this code.

1. Encoder: It works as a compression unit that compresses the input data into a latent-space representation. It encodes the input data as a compressed representation in a reduced dimension.
2. Decoder: It aims to reconstruct the input data from the latent space representation.
3. Latent representation: layer that contains the code, represents the compressed knowledge of the input data.

The key idea is that the autoencoder learns to ignore the noise and capture the most salient features of the data.

The code, as a compressed representation of the input, prevents the neural network from memorizing the input and overfitting the data.

The smaller the code, the lower the risk of overfitting.

Auto-Encoders

Autoencoders are mainly a dimensionality reduction (or compression) algorithm with following properties:

1. Data-specific: Autoencoders are only able to meaningfully compress data similar to what they have been trained on.

- Since they learn features specific for the given training data, they are different than a standard data compression algorithm like gzip.
- So we cannot expect an autoencoder trained on handwritten digits to compress landscape photos.

2. Lossy: The output of the autoencoder will not be exactly the same as the input, it will be a close but degraded representation.

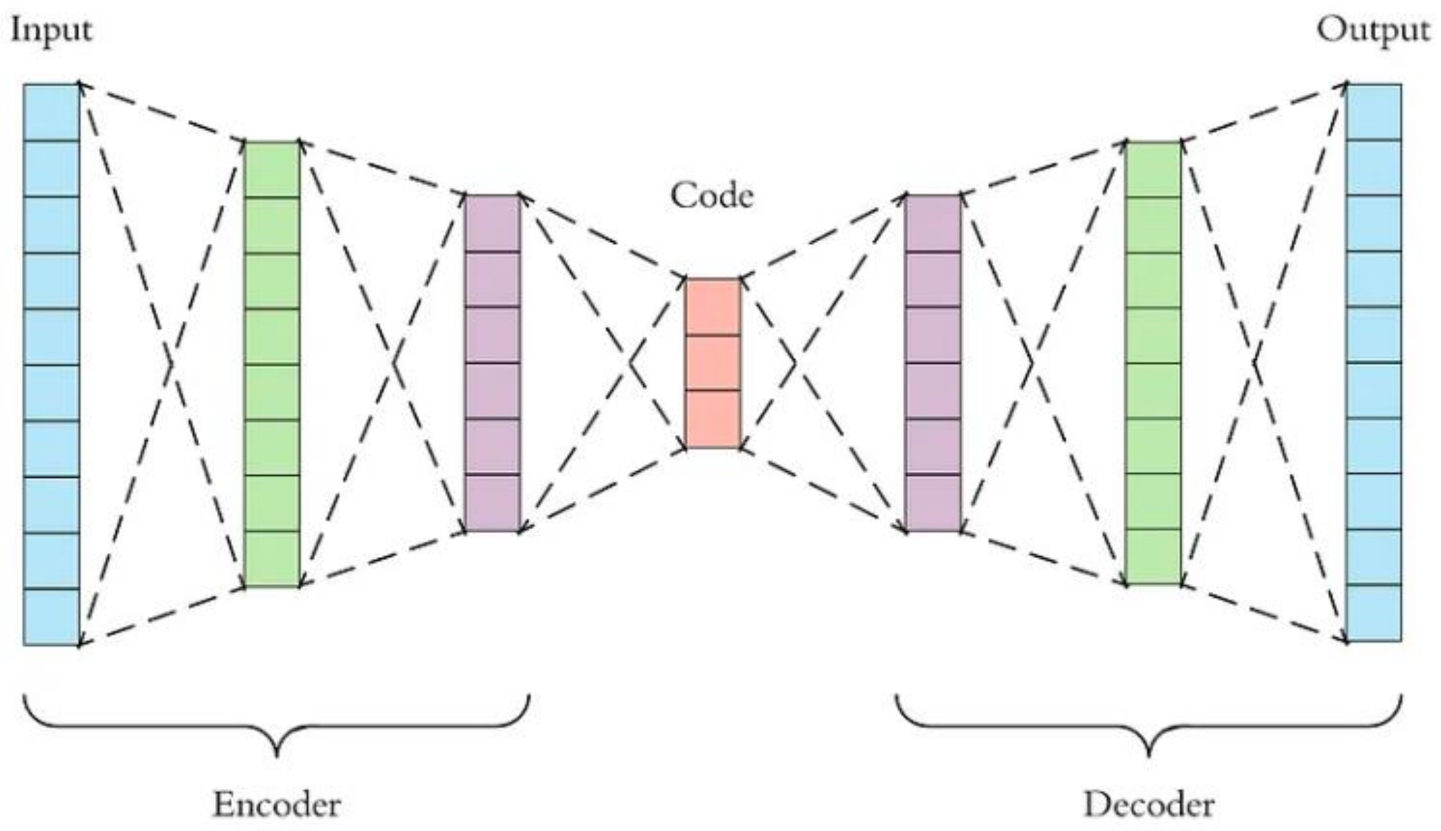
- They cannot give lossless compression

3. Unsupervised: To train an autoencoder we only need to feed the raw input data

- Autoencoders are considered an unsupervised learning technique since they do not need explicit labels to train on.
- To be more precise they are self-supervised because they generate their own labels from the training data.

Autoencoder Architecture

The architecture of an autoencoder is designed to be symmetrical with a latent representation known as code in the middle.



Autoencoder Architecture

- In an Autoencoder both Encoder and Decoder are made up of a
  - Combination of NN (fully-connected feedforward neural networks or ANNs) layers
  - In the case of CNN Autoencoder, these layers are CNN layers (Convolutional, Max Pool, Flattening, etc.)
  - In the case of RNN/LSTM their respective layers are used.
- Code is a single layer of an ANN with the dimensionality of our choice.
- The number of nodes in the code layer (code size) is a hyperparameter that we set before training the autoencoder.

Autoencoder Architecture

- There are 4 hyperparameters that we need to set before training an autoencoder:

- Code size: number of nodes in the middle layer. Smaller size results in more compression.
- Number of layers: the autoencoder can be as deep as our choice. In the figure above we have 2 layers in both the encoder and decoder, without considering the input and output.
- Number of nodes per layer: the autoencoder architecture we are working on is called a stacked autoencoder since the layers are stacked one after another.
- The number of nodes per layer decreases with each subsequent layer of the encoder, and increases back in the decoder.
- Also the decoder is symmetric to the encoder in terms of layer structure (this is not necessary and we have choice over these parameters).
- Loss function: we either use mean squared error (MSE) or binary crossentropy. If the input values are in the range [0, 1] then we typically use crossentropy, otherwise we use the mean squared error.

Autoencoder Architecture

- First the input passes through the encoder, which is a fully-connected ANN, to produce the code.
- The decoder, which has the similar ANN structure, then produces the output only using the code.
- The goal is to get an output identical with the input.
- Note that the decoder architecture is the mirror image of the encoder.
- This is not a requirement but it is typically the case.
- The only requirement is the dimensionality of the input and output needs to be the same.

Auto-encoder using MNIST dataset - Grayscale images of handwritten single digits between 0 and 9

Auto-encoders using MNIST dataset

- Using Autoencoders we can compress image files which are useful for sharing and saving in a faster and memory efficient way.
- These compressed images contain the key information same as in original images but in a compressed format that can be used further for other reconstructions and transformations.
- Create an Autoencoder that can compress the image file for the MNIST dataset
- Input dimension: 28*28 = 784
- Encoder : 784 ==> 128 ==> 64 ==> 36 ==> 18 ==> 9
- Decoder: 9 ==> 18 ==> 36 ==> 64 ==> 128 ==> 784 ==> 28*28 = 784

Auto-encoders using MNIST dataset

- Create an Autoencoder that can compress the image file for the MNIST dataset
- The encoder reduces the dimensionality of the data sequentially as given by:
- 28*28 = 784 ==> 128 ==> 64 ==> 36 ==> 18 ==> 9
- Here the number of input nodes is 784 that are coded into 9 nodes in the latent space.
- The decoder increases the dimensionality of the data to the original input size, in order to reconstruct the input.
- 9 ==> 18 ==> 36 ==> 64 ==> 128 ==> 784 ==> 28*28 = 784
- Here the input is the 9-node latent space representation and the output is the 28*28 reconstructed input.

Auto-encoders using MNIST dataset

Importing necessary libraries

```
import torch
from torchvision import datasets
from torchvision import transforms
import matplotlib.pyplot as plt
```

Auto-encoders using MNIST dataset

Step 1: Loading the Dataset

- Load the MNIST dataset into loader using DataLoader module.
- The dataset is downloaded and transformed into image tensors.
- Using the DataLoader module, the tensors are loaded
- The dataset is loaded with Shuffling enabled and a batch size of 32.

```
# Transforms images to a PyTorch Tensor
tensor_transform = transforms.ToTensor()
# Download the MNIST Dataset
dataset = datasets.MNIST(root = "./data", train = True, download = True, transform =
tensor_transform)
# DataLoader is used to load the dataset or training
loader = torch.utils.data.DataLoader(dataset = dataset, batch_size = 32, shuffle = True)
```

Auto-encoders using MNIST dataset

Step 2: Configure encoder and decoder dimensions

- The encoder reduces the dimensionality of the data sequentially as given by:
- 28*28 = 784 ==> 128 ==> 64 ==> 36 ==> 18 ==> 9
- Here the number of input nodes is 784 that are coded into 9 nodes in the latent space.
- The decoder increases the dimensionality of the data to the original input size, in order to reconstruct the input.
- 9 ==> 18 ==> 36 ==> 64 ==> 128 ==> 784 ==> 28*28 = 784
- Here the input is the 9-node latent space representation and the output is the 28*28 reconstructed input.
- The encoder starts with 28*28 nodes in a Linear layer followed by a ReLU layer, and it goes on until the dimensionality is reduced to 9 nodes.
- The decoder uses these 9-node latent representation to bring back the original image by using the inverse of the encoder architecture.
- The decoder architecture uses a Sigmoid Layer to range the values between 0 and 1 only.

Auto-encoders using MNIST dataset

Step 3: Create Autoencoder Class: 28*28 ==> 9 ==> 28*28

```
class AE(torch.nn.Module):
        def __init__(self):
                super().__init__()
                 #Building an linear encoder with Linear
                # layer followed by Relu activation function

                self.encoder = torch.nn.Sequential(
                        torch.nn.Linear(28 * 28, 128),
                        torch.nn.ReLU(),
                        torch.nn.Linear(128, 64),
                        torch.nn.ReLU(),
                        torch.nn.Linear(64, 36),
                        torch.nn.ReLU(),
                        torch.nn.Linear(36, 18),
                        torch.nn.ReLU(),
                        torch.nn.Linear(18, 9)
                )
```

```
self.decoder = torch.nn.Sequential(
```

Write decoder architecture

```
                )
        def forward(self, x):
                encoded = self.encoder(x)
                decoded = self.decoder(encoded)
                return decoded
```

Auto-encoders using MNIST dataset

Step 3: Create Autoencoder Class: 28*28 ==> 9 ==> 28*28

```
class AE(torch.nn.Module):
        def __init__(self):
                super().__init__()
                 #Building an linear encoder with Linear
                # layer followed by Relu activation function

                self.encoder = torch.nn.Sequential(
                        torch.nn.Linear(28 * 28, 128),
                        torch.nn.ReLU(),
                        torch.nn.Linear(128, 64),
                        torch.nn.ReLU(),
                        torch.nn.Linear(64, 36),
                        torch.nn.ReLU(),
                        torch.nn.Linear(36, 18),
                        torch.nn.ReLU(),
                        torch.nn.Linear(18, 9)
                )
                self.decoder = torch.nn.Sequential(
                        torch.nn.Linear(9, 18),
                        torch.nn.ReLU(),
                        torch.nn.Linear(18, 36),
                        torch.nn.ReLU(),
                        torch.nn.Linear(36, 64),
                        torch.nn.ReLU(),
                        torch.nn.Linear(64, 128),
                        torch.nn.ReLU(),
                        torch.nn.Linear(128, 28 * 28),
                        torch.nn.Sigmoid()
                )
        def forward(self, x):
                encoded = self.encoder(x)
                decoded = self.decoder(encoded)
                return decoded
```

Auto-encoders using MNIST dataset

Step 4: Model Initialization

```
model = AE()

# Validation using MSE Loss function
#loss_function = torch.nn.L1Loss()
loss_function = torch.nn.MSELoss()
# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-1, weight_decay =
1e-8)
epochs = 20
outputs = []
losses = []
```

Auto-encoders using MNIST dataset

Step 5: Training Loop

```
for epoch in range(epochs):
    for (image, _) in loader:
            # Reshaping the image to (-1, 784)
            image = image.reshape(-1, 28*28)
            # Output of Autoencoder
            reconstructed = model(image)
            loss = loss_function(reconstructed, image)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            # Storing the losses in a list for plotting
            losses.append(loss)
            outputs.append((epochs, image, reconstructed))
```

Auto-encoders using MNIST dataset

Step 6: Output generation

```
# Defining the Plot Style
plt.style.use('fivethirtyeight')
plt.xlabel('Iterations')
plt.ylabel('Loss')


# Plotting the last 100 values
plt.plot([loss.item() for loss in losses[-100:]])
```

Auto-encoders using MNIST dataset

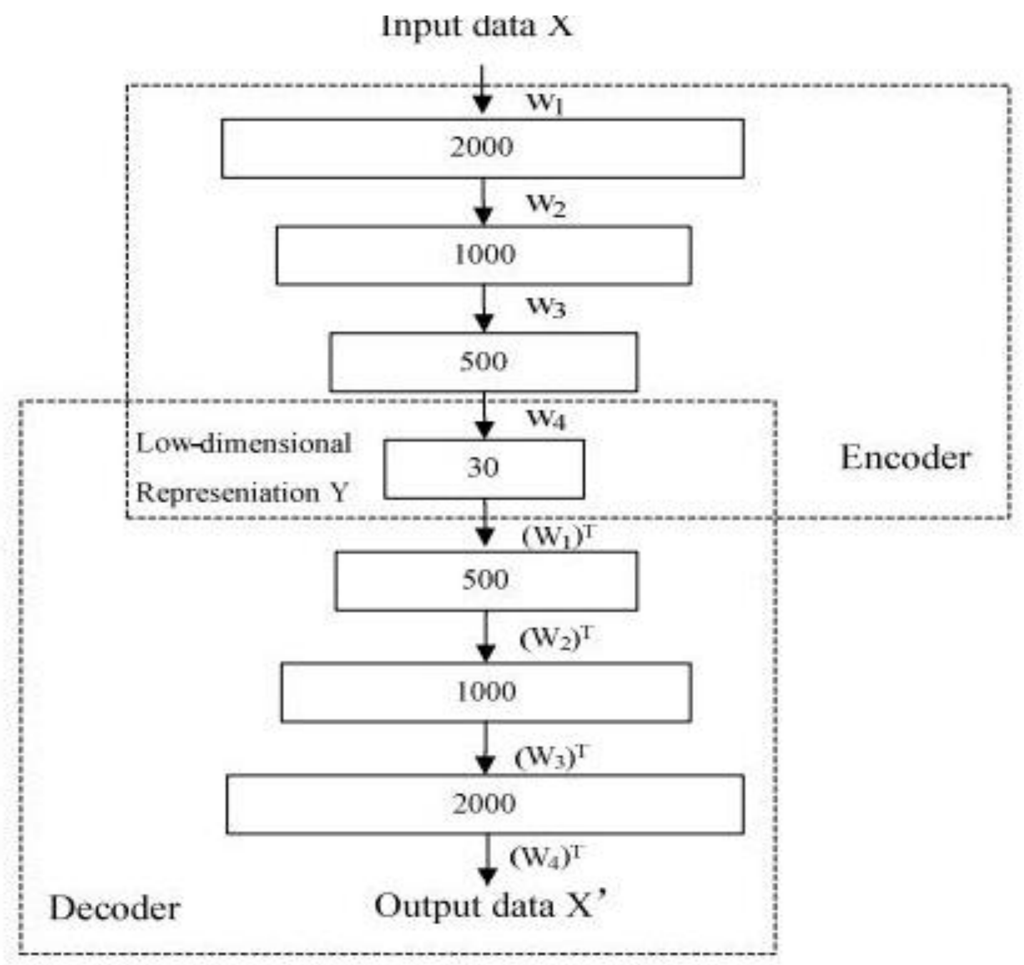
Step 7: Input/Reconstructed Input to/from Autoencoder

```
# Plotting original images
plt.figure(figsize=(20, 4))
num_images_to_plot = min(len(image), 20)
for i in range(num_images_to_plot):
    plt.subplot(2, 10, i+1)
    plt.imshow(image[i].detach().reshape(28, 28), cmap='gray')
    plt.axis('off')

# Plotting reconstructed images
plt.figure(figsize=(20, 4))
for i in range(num_images_to_plot):
    plt.subplot(2, 10, i+1)
    plt.imshow(reconstructed[i].detach().reshape(28, 28), cmap='gray')
    plt.axis('off')

plt.show()
```

Autoencoder using CNN

- Repeat the autoencoder construction using CNN instead of FFNW

Schematic of a typical Autoencoder



Applications of Autoencoders

1. File Compression: Primary use of Autoencoders is that they can reduce the dimensionality of input data which is referred to as file compression.

Autoencoders works with all kinds of data like Images, Videos, and Audio, this helps in sharing and viewing data faster than we could do with its original file size.

2. Image De-noising: Autoencoders are also used as noise removal techniques - Image De-noising

It is the best choice for De-noising because it does not require any human interaction

once trained on any kind of data it can reproduce that data with less noise than the original image.

3. Image Transformation: Autoencoders are also used for image transformations, which is typically classified under GAN(Generative Adversarial Networks) models.

Using these we can transform Black and White images to colored one and vice versa, we can up-sample and down-sample the input data, etc.

Auto-encoders - Summary

- Recently, the autoencoder concept has become more widely used for learning generative models of data.