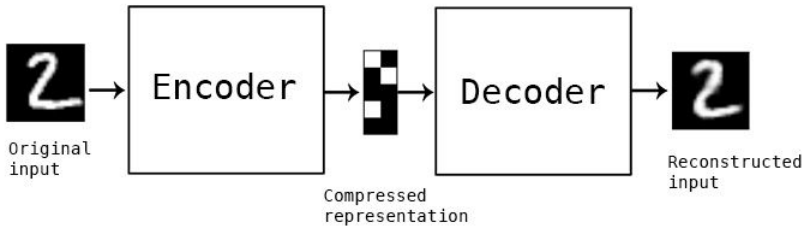
Autoencoder

CSCI 6379 Intro to DL

Autoencoder

- An autoencoder is a neural network used for unsupervised learning of efficient codings.
- The aim of an autoencoder is to learn a representation (encoding) for a set of data, typically for the purpose of dimensionality reduction.
- An autoencoder learns to compress the data from the input layer into a short code, and then uncompress that code into something that closely matches the original data.



Encoder and Decoder

Autoencoders consist of two main parts:

Encoder: This part of the network compresses the input into a latent-space representation. It defines a mapping from the input data to the representation space. This part can be thought of as a function h(x), where x is the input data.

Decoder: This part aims to reconstruct the input from the latent space representation. It defines a mapping from the representation space back to the original input space. This can be thought of as a function g(h(x)), where the goal is to make g(h(x)) as close as possible to x.

Input: x

Latent vector: z = h(x)

Output: y = g(z) = g(h(x))

Loss functions

Common loss functions used in the context of autoencoders include:

Mean Squared Error (MSE) Loss: For real-valued input data, the MSE loss can be used to penalize the squared difference between each element in the input and the output. It is defined as:

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2$$

Where N is the number of elements in x (and y), and x_i and y_i a i-th elements of x and y, respectively.

Binary Cross-Entropy Loss: For binary input data (e.g., black and white images), the binary cross-entropy loss can be appropriate. It measures the difference between two probability distributions for each element in the input and output, defined as:

$$\text{BCE} = -\frac{1}{N}\sum_{i=1}^{N}[x_i \log(y_i) + (1 - x_i)\log(1 - y_i)]$$

Where x_i is a binary element in the input, and y_i is the predicte probability of that element being 1.

Encoder

```
class Encoder(nn.Module):
    def __init__(self, encoded_space_dim,fc2_input_dim):
        super().__init__()

        self.encoder_cnn = nn.Sequential(
            nn.Conv2d(1, 8, 3, stride=2, padding=1),
            nn.ReLU(True),
            nn.Conv2d(8, 16, 3, stride=2, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.Conv2d(16, 32, 3, stride=2, padding=0),
            nn.ReLU(True)
        )

        self.flatten = nn.Flatten(start_dim=1)
        self.encoder_lin = nn.Sequential(
            nn.Linear(3 * 3 * 32, 128),
            nn.ReLU(True),
            nn.Linear(128, encoded_space_dim)
        )

    def forward(self, x):
        x = self.encoder_cnn(x)
        x = self.flatten(x)
        x = self.encoder_lin(x)
        return x
```

Decoder

```
class Decoder(nn.Module):
    def __init__(self, encoded_space_dim,fc2_input_dim):
        super().__init__()
        self.decoder_lin = nn.Sequential(
            nn.Linear(encoded_space_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, 3 * 3 * 32),
            nn.ReLU(True)
        )
        self.unflatten = nn.Unflatten(dim=1,unflattened_size=(32, 3, 3))
        self.decoder_conv = nn.Sequential(
            nn.ConvTranspose2d(32, 16, 3,
            stride=2, output_padding=0),
            nn.BatchNorm2d(16),
            nn.ReLU(True),
            nn.ConvTranspose2d(16, 8, 3, stride=2,
            padding=1, output_padding=1),
            nn.BatchNorm2d(8),
            nn.ReLU(True),
            nn.ConvTranspose2d(8, 1, 3, stride=2,
            padding=1, output_padding=1)
        )
    def forward(self, x):
        x = self.decoder_lin(x)
        x = self.unflatten(x)
        x = self.decoder_conv(x)
        x = torch.sigmoid(x)
        return x
```

Models and Parameters

```
encoder = Encoder(encoded_space_dim=4,fc2_input_dim=128)
decoder = Decoder(encoded_space_dim=4,fc2_input_dim=128)
encoder.to(device)
decoder.to(device)

params_to_optimize = [
    {'params': encoder.parameters()},
    {'params': decoder.parameters()}
]
optim = torch.optim.Adam(params_to_optimize, lr=0.001, weight_decay=1e-05)
loss_fn = torch.nn.MSELoss()
```

train function

```
def train_epoch(encoder, decoder, device, dataloader, loss_fn, optimizer,noise_factor=0.3):
    encoder.train()
    decoder.train()
    train_loss = []
    for image_batch, _ in dataloader:
        image_noisy = add_noise(image_batch,noise_factor)
        image_noisy = image_noisy.to(device)
        encoded_data = encoder(image_noisy)
        decoded_data = decoder(encoded_data)
        loss = loss_fn(decoded_data,image_noisy)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        train_loss.append(loss.detach().cpu().numpy())
    return np.mean(train_loss)
```

test function

```
def test_epoch(encoder, decoder, device, dataloader, loss_fn,noise_factor=0.3):
    encoder.eval()
    decoder.eval()
    with torch.no_grad():
        conc_out = []
        conc_label = []
        for image_batch, _ in dataloader:
            image_batch = image_batch.to(device)
            encoded_data = encoder(image_batch)
            decoded_data = decoder(encoded_data)
            conc_out.append(decoded_data.cpu())
            conc_label.append(image_batch.cpu())
        conc_out = torch.cat(conc_out)
        conc_label = torch.cat(conc_label)
        val_loss = loss_fn(conc_out, conc_label)
    return val_loss.data
```

training

```
loss_fn = torch.nn.MSELoss()

for epoch in range(num_epochs):
   train_loss=train_epoch(
        encoder=encoder,
        decoder=decoder,
        device=device,
        dataloader=train_loader,
        loss_fn=loss_fn,
        optimizer=optim, noise_factor=0.3)
   val_loss = test_epoch(
        encoder=encoder,
        decoder=decoder,
        device=device,
        dataloader=test_loader,
        loss_fn=loss_fn, noise_factor=0.3)
```

Results (10th epoch)