PyTorch Implementation of Autoencoders

A Short Overview of Autoencoder Implementations in PyTorch

Scope: This document is a summary of the "Module 4: PyTorch Implementation of Autoencoders" notebook from the course "Deep Learning: Mastering Neural Networks" by MIT xPRO. This summary offers a quick overview of the main aspects included in the notebook.

Table of Contents

1. Simple Autoencoder with 0,1,2 MNIST Digits

In this case, we will use the MNIST dataset from the Module 3 assignment. To start with, the dataset will also be pruned to only contain digits 0, 1, and 2. An important data preparation step is to normalize the intensity of the images; we then do the usual train, validation, test data split.

```
# These transforms will be performed on every datapoint - in this example we want to transform
# every datapoint to a Tensor datatype, and perform normalization
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize([0.5], [0.5])])
mnist_train = torchvision.datasets.MNIST('', train=True, transform=transform, download=True)
# Here we will only keep the digits 0, 1, and 2
train_indices = [idx for idx, target in enumerate(mnist_train.targets) if target in [0, 1, 2]]
mnist_train = Subset(mnist_train, train_indices)

# We will split out train dataset into train and validation!
mnist_train, mnist_val = torch.utils.data.random_split(mnist_train,
               [int(np.floor(len(mnist_train)*0.75)), int(np.ceil(len(mnist_train)*0.25))])
# Keep only the digits 0, 1, and 2 for the test as well
mnist_test = torchvision.datasets.MNIST('', train=False, transform=transform, download=True)
test_indices = [idx for idx, target in enumerate(mnist_test.targets) if target in [0, 1, 2]]
mnist_test = Subset(mnist_test, test_indices)

# We will create DataLoaders just like before with a batch size of 100
batch_size = 100
dataloaders = {'train': DataLoader(mnist_train, batch_size=batch_size),
               'val': DataLoader(mnist_val, batch_size=batch_size),
               'test': DataLoader(mnist_test, batch_size=batch_size)}
dataset_sizes = {'train': len(mnist_train),
                 'val': len(mnist_val),
                 'test': len(mnist_test)}
```

a. Model Definition and Hyperparameters

!Remember: In this case, we have separated the model into two distinct sections: an encoder and decoder. This allows us to easily encode a specific datapoint after the model is trained.

```
class SimpleAutoEncoder(nn.Module):
    def __init__(self, input_size):
        super(SimpleAutoEncoder, self).__init__()
        # Split the Encoder and Decoder
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 2)
        )
        self.decoder = nn.Sequential(
            nn.Linear(2, 64),
            nn.ReLU(),
            nn.Linear(64, 128),
            nn.ReLU(),
            nn.Linear(128, input_size)
        )

    def forward(self, x):
        return self.decoder(self.encoder(x))

    def encode(self, x):
        return self.encoder(x)
```

```
# hyperparameters
input_size = 784 # flattening our dataset 28*28 = 784
learning_rate = 0.001
num_epochs = 5

model = SimpleAutoEncoder(input_size).to(device)
```

b. Training the Model

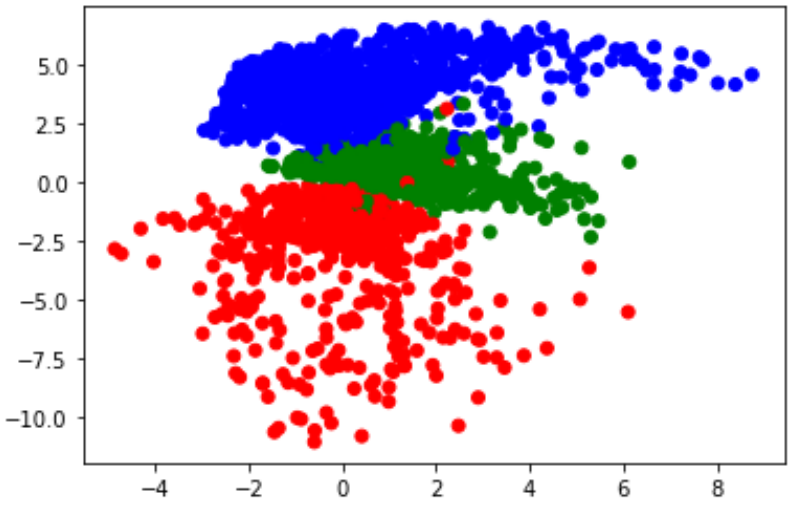
This training function for our autoencoder, “train_autoencoder”, as defined in the notebook is very similar to the ones mentioned in earlier notebooks. Keep in mind that an autoencoder is similar to a regression model, so there aren’t specific class labels/accuracy, only loss related to mean square error between our autoencoder output, and the true image.

```
# loss and optimizer
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer, gamma=0.95)

model, training_curves = train_autoencoder(model, dataloaders, dataset_sizes, criterion,
                                           optimizer, scheduler, num_epochs=num_epochs)
```

c. Visualizing the Latent Space Encoding

Now, let's visualize the 2D latent space encoding for our different digits. Here we use the "plot_dataset" function as defined in the notebook, which just does a scatter plot of each type of digit against the two dimensions of our latent space (the two hidden layer nodes in the middle of the autoencoder). Below, red indicates digit "0", blue is digit "1", and green is digit "2".



As you can see, our encoding does a reasonably good job of separating the different digit classes. Each cluster represents the predetermined digits 0, 1, and 2. So, we might expect that the latent representation could be an excellent starting point for a classification neural network.

2. Autoencoder on Entire MNIST Dataset

After visualizing what an autoencoder can do in latent space, we build a deeper network and expand to the entire MNIST dataset.

a. Dataset Preparation

In the notebook, dataset preparation is very similar to the earlier 0, 1, 2 case, except we keep all of the MNIST classes.

b. Model Hyperparameters and Definition

Now we are using a latent space with dimension 32, which will allow for more information to be encoded, leading to more accurate decoding.

```
class AutoEncoder(nn.Module):
    def __init__(self, input_size):
        super(AutoEncoder, self).__init__()
        ## Split the Encoder and Decoder
        self.encoder = nn.Sequential(
            nn.Linear(input_size, 256),
            nn.ReLU(),
            nn.Linear(256,32)
        )
```

```
        self.decoder = nn.Sequential(
            nn.Linear(32, 256),
            nn.ReLU(),
            nn.Linear(256, input_size)
        )

    def forward(self, x):
        return self.decoder(self.encoder(x))

    def encode(self, x):
        return self.encoder(x)
```

```
# hyperparameters
input_size = 784 # flattening our dataset 28*28 = 784
learning_rate = 0.001
num_epochs = 10

model = AutoEncoder(input_size).to(device)
```

c. Training the Model

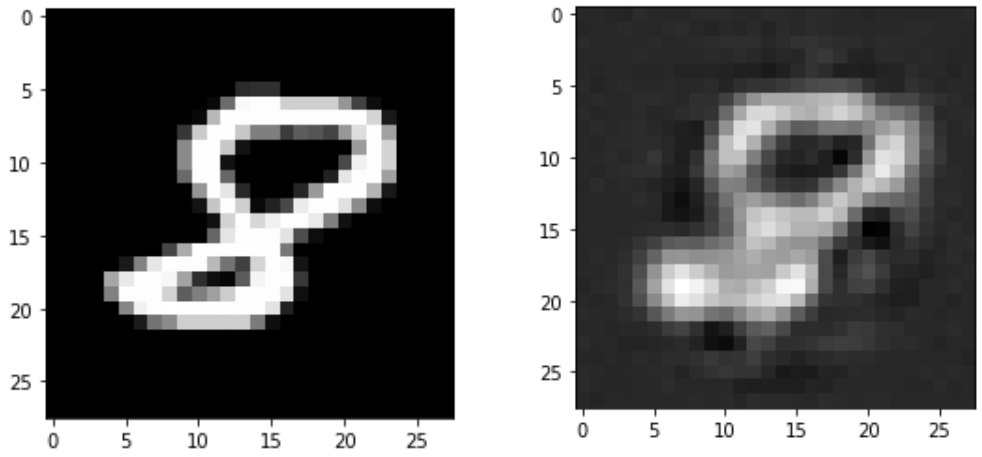
Since we are still training an autoencoder, we will use the exact same training function used in section 1.

d. Visualizing the Results

Lastly, we are going to visualize the results of our autoencoder. The notebook provides a "plot_digit" function that displays an example input image for that digit type, and then the reconstructed output of the autoencoder for that input. This will give us a feel for how well the autoencoder is able to regenerate a complicated digit image, given a latent representation of only 32 numbers.

Below is an example, where we see that the basic structure of the input image is relatively well reconstructed:

```
plot_digit(8)
```



The notebook generalizes this plot function to show some "count" number of digit examples, which you will be able to experiment with in your notebook.