DL_HW4_AmirPourmand

March 12, 2022

```
<font face="XB Zar" size=5>
    <div align=center>
        <font face="IranNastaliq" size=30>
            <p></p>
            <p></p>


        </font>
        <font color=#FF7500>

          -

        <font color=blue>


    <hr/>
    <div align=center>
        <font size=6>
            <br />
```

- Autoencoders - Attention Models :

```
</font>
```

```
<font face="XB Zar" size=5>
    <div align=center>
            <p></p>
            <p></p>
            :
            <br/>
                :
    </div>
    <br />
    <hr />
    <style type="text/css" scoped>
    p{
    border: 1px solid #a2a9b1;background-color: #f8f9fa;display: inline-block;
    };
    </style>
</font>
```

1 Autoencoders

Just Complete the ToDo Parts

[ ]:
```python
import torch
import torch.nn as nn
from torchvision import datasets
from torchvision import transforms
import matplotlib.pyplot as plt
```

[ ]:
```python
# 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
# for training
loader = torch.utils.data.DataLoader(dataset = dataset,
                                     batch_size = 32,
                                     shuffle = True)
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./data/MNIST/raw/train-images-idx3-ubyte.gz

  0%|          | 0/9912422 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./data/MNIST/raw/train-labels-idx1-ubyte.gz

  0%|          | 0/28881 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./data/MNIST/raw/t10k-images-idx3-ubyte.gz

  0%|          | 0/1648877 [00:00<?, ?it/s]

Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
```

```
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./data/MNIST/raw/t10k-labels-idx1-ubyte.gz
```

```
  0%|          | 0/4542 [00:00<?, ?it/s]
```

```
Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

[ ]:
```
# Creating a PyTorch class
# 28*28 ==> 9 ==> 28*28
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        ''' Todo: Build a linear encoder with Linear
         layer followed by Relu activation function
         784 ==> 9 '''
        self.encoder = nn.Sequential(
                nn.Linear(28*28, 14*14),
                nn.ReLU(),
                nn.Linear(14*14,7*7),
                nn.ReLU(),
                nn.Linear(7*7, 9),
                nn.ReLU(),
        )

        ''' Todo: Build a linear decoder with Linear
         layer followed by Relu activation function
         The Sigmoid activation function
         outputs the value between 0 and 1
         9 ==> 784 '''
        self.decoder = nn.Sequential(
                nn.Linear(9,7*7),
                nn.ReLU(),
                nn.Linear(7*7,14*14),
                nn.ReLU(),
                nn.Linear(14*14,28*28),
                nn.Sigmoid()
        )

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

[ ]:
```
''' Todo: Initialaize model '''
model = AE()
```

```
''' Todo: Validation using MSE Loss function '''
loss_function = nn.MSELoss()

''' Todo: Use an Adam Optimizer with lr = 0.1 '''
optimizer = torch.optim.Adam(model.parameters(),lr=0.1)
```

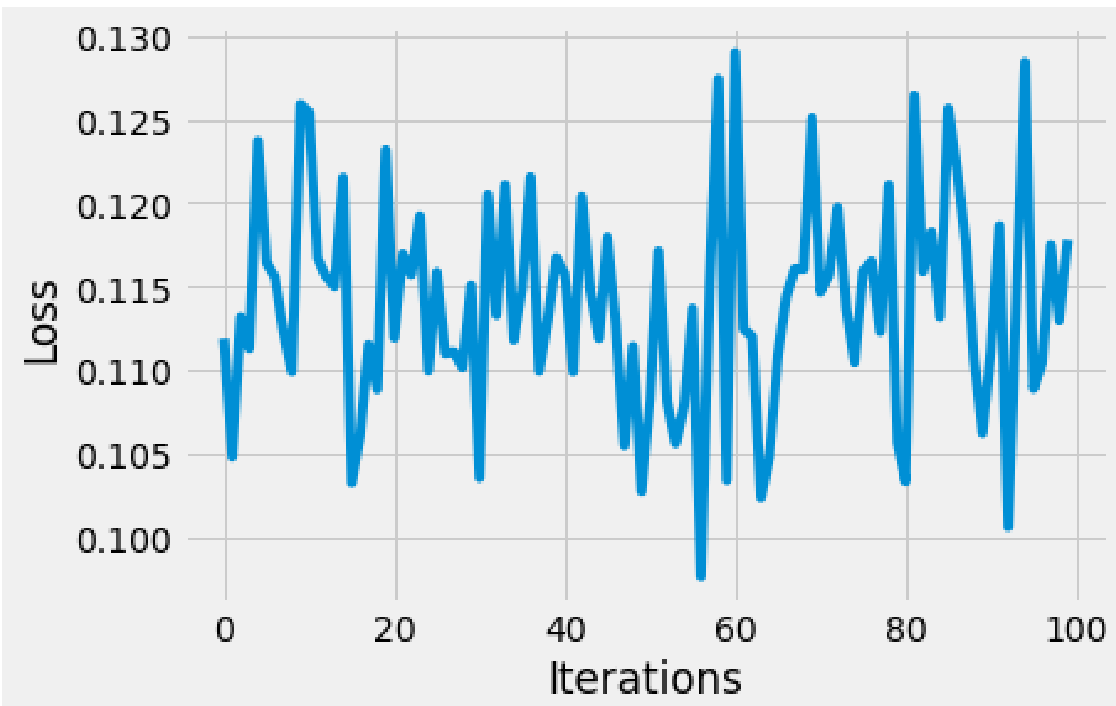
[ ]:
```
from tqdm import tqdm

epochs = 20
outputs = []
losses = []
for epoch in range(epochs):
    for (image, _) in tqdm(loader):

      ''' Todo: Reshaping the image to (-1, 784) '''
      image = torch.reshape(image,(-1,784))

      # Output of Autoencoder
      reconstructed = model(image)

      ''' Todo: Calculate the loss function '''
      loss = loss_function(image,reconstructed)

      # The gradients are set to zero,
      # the the gradient is computed and stored.
      # .step() performs parameter update
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      # Storing the losses in a list for plotting
      losses.append(loss)
    outputs.append((epochs, image, reconstructed))

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

''' Todo: Plot the last 100 values '''
plt.plot(range(100),losses[-100:])
```

```
100%|      | 1875/1875 [00:30<00:00, 62.43it/s]
100%|      | 1875/1875 [00:29<00:00, 63.38it/s]
100%|      | 1875/1875 [00:29<00:00, 62.60it/s]
100%|      | 1875/1875 [00:29<00:00, 63.24it/s]
100%|      | 1875/1875 [00:31<00:00, 59.72it/s]
```

```
100%|          | 1875/1875 [00:30<00:00, 61.65it/s]
100%|          | 1875/1875 [00:30<00:00, 60.92it/s]
100%|          | 1875/1875 [00:31<00:00, 59.38it/s]
100%|          | 1875/1875 [00:31<00:00, 59.76it/s]
100%|          | 1875/1875 [00:33<00:00, 56.50it/s]
100%|          | 1875/1875 [00:33<00:00, 56.42it/s]
100%|          | 1875/1875 [00:34<00:00, 54.20it/s]
100%|          | 1875/1875 [00:33<00:00, 55.73it/s]
100%|          | 1875/1875 [00:36<00:00, 51.37it/s]
100%|          | 1875/1875 [00:36<00:00, 51.04it/s]
100%|          | 1875/1875 [00:35<00:00, 52.87it/s]
100%|          | 1875/1875 [00:35<00:00, 52.79it/s]
100%|          | 1875/1875 [00:36<00:00, 51.21it/s]
100%|          | 1875/1875 [00:37<00:00, 50.36it/s]
100%|          | 1875/1875 [00:38<00:00, 48.53it/s]
```

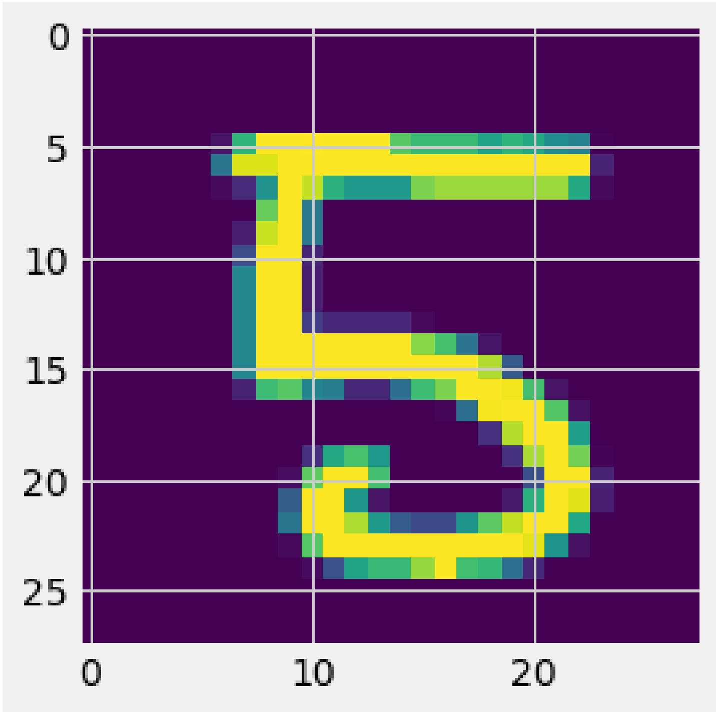
[ ]:

```
[<matplotlib.lines.Line2D at 0x7fc05bf67510>]
```

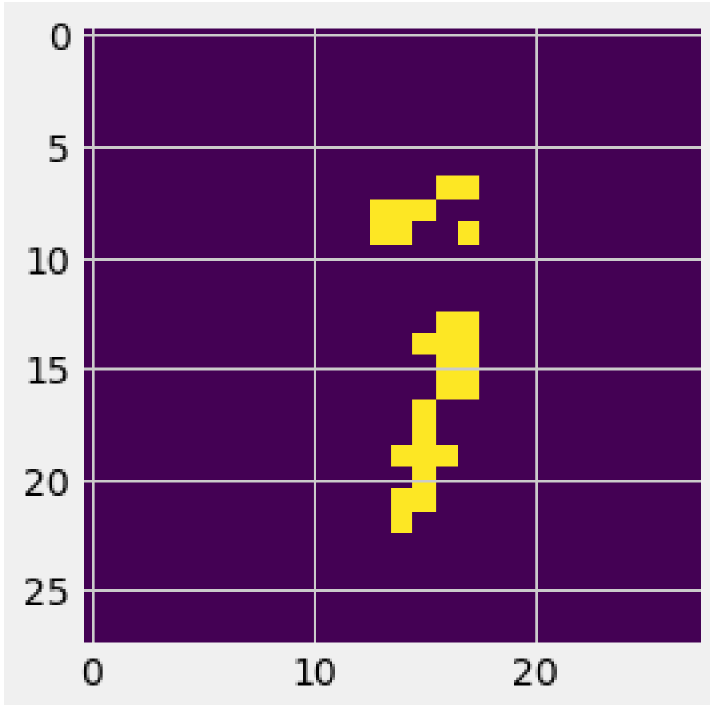


[ ]:

```python
# Plot the first input image array
for i, item in enumerate(image):

  # Reshape the array for plotting
  item = item.reshape(-1, 28, 28)
  plt.imshow(item[0])
```




```
[ ]: ''' Todo: Plot the first reconstructed input image array '''
for i, item in enumerate(reconstructed):

  # Reshape the array for plotting
  item = item.reshape(-1, 28, 28)
  plt.imshow(item[0].detach().numpy())
```



2 Attention Models

Just Complete the ToDo Parts

[ ]:
```
# Some imports that we require to write the network.
import torch
import torch.nn as nn
from torch import optim
import torch.nn.functional as F
from torch.autograd import Variable
```

[ ]:
```
# Encoder for the attention network that is similar to the vanilla encoders
class Encoder(nn.Module):
  # Store the parameters
  def __init__(self, input_size, hidden_size, bidirectional = True):
    super(Encoder, self).__init__()
    self.hidden_size = hidden_size
    self.input_size = input_size
    self.bidirectional = bidirectional

    ''' ToDo : Create an LSTM layer '''
    self.lstm = nn.LSTM(input_size, hidden_size, bidirectional = bidirectional)
```

```
  # The Forward function
  def forward(self, inputs, hidden):

    ''' Todo : Pass the input through the LSTM with the provided hidden state␣
↪'''
    output, hidden = self.lstm(inputs.view(1, 1, self.input_size), hidden)
    return output, hidden

  # This function has to be called before passing sentence through the LSTM to␣
↪initialize the hidden state.
  def init_hidden(self):
    return (torch.zeros(1 + int(self.bidirectional), 1, self.hidden_size),
      torch.zeros(1 + int(self.bidirectional), 1, self.hidden_size))
```

[ ]:

```
# This class is the attention based decoder
class AttentionDecoder(nn.Module):

  def __init__(self, hidden_size, output_size, vocab_size):
    super(AttentionDecoder, self).__init__()
    self.hidden_size = hidden_size
    self.output_size = output_size

    # This layer calculates the importance of the word, by using the previous␣
↪decoder hidden state and the hidden state of the encoder at that particular␣
↪time step
    self.attn = nn.Linear(hidden_size + output_size, 1)
    ''' Todo: The 'lstm' layer takes in concatenation of vector obtained by␣
↪having a weighted sum according to attention weights and the previous word␣
↪outputted '''
    self.lstm = nn.LSTM(hidden_size + vocab_size, output_size)
    ''' Todo: Map the output feature space into the size of vocabulary '''
    self.final = nn.Linear(output_size, vocab_size)

  # The 'init_hidden' function is used in the same way as in the encoder.
  def init_hidden(self):
    return (torch.zeros(1, 1, self.output_size),
      torch.zeros(1, 1, self.output_size))

  # The forward function of the decoder
  def forward(self, decoder_hidden, encoder_outputs, input):

    # 'weights' list is used to store the attention weights
    weights = []
    for i in range(len(encoder_outputs)):
      print(decoder_hidden[0][0].shape)
      print(encoder_outputs[0].shape)
```

```
            # Pass each encoder output through the 'attn' layer along with
            # decoder's previous hidden state by concatenating them and store
            # them in the 'weights' list
            weights.append(self.attn(torch.cat((decoder_hidden[0][0],
                                                encoder_outputs[i]), dim = 1)))

        ''' Todo : scale weights in range (0,1) by applying softmax activation '''
        normalized_weights = F.softmax(torch.cat(weights, 1), 1)

        # To calculate the weighted sum, we use batch matrix multiplication
        attn_applied = torch.bmm(normalized_weights.unsqueeze(1),
                                 encoder_outputs.view(1, -1, self.hidden_size))

        input_lstm = torch.cat((attn_applied[0], input[0]), dim = 1) #if we are␣
↪using embedding, use embedding of input here instead

        output, hidden = self.lstm(input_lstm.unsqueeze(0), decoder_hidden)

        output = self.final(output[0])

        return output, hidden, normalized_weights
```

[ ]:
```
# Testing the code
bidirectional = True
c = Encoder(10, 20, bidirectional)
a, b = c.forward(torch.randn(10), c.init_hidden())
print(a.shape)
print(b[0].shape)
print(b[1].shape)

x = AttentionDecoder(20 * (1 + bidirectional), 25, 30)
y, z, w = x.forward(x.init_hidden(), torch.cat((a,a)), torch.zeros(1,1, 30))
print(y.shape)
print(z[0].shape)
print(z[1].shape)
print(w)
```

```
torch.Size([1, 1, 40])
torch.Size([2, 1, 20])
torch.Size([2, 1, 20])
torch.Size([1, 25])
torch.Size([1, 40])
torch.Size([1, 25])
torch.Size([1, 40])
torch.Size([1, 30])
torch.Size([1, 1, 25])
torch.Size([1, 1, 25])
```

```
tensor([[0.5000, 0.5000]], grad_fn=<SoftmaxBackward0>)
```

```
[ ]:
```