PURDUE UNIVERSITY
ECE 595 MACHINE LEARNING II

Project 2: Autoencoders

Due Date: October 4, 2019

1 INTRODUCTION

An autoencoder is simply a neural network which is trained to reproduce an approximation of its input at its output. Typically, the input is mapped to a reduced dimension representation in the hidden layer, referred to as the "bottleneck" layer, and then the "bottleneck" layer representation is used to calculate the reconstruction of the input sample. The *encoder* of the autoencoder is the portion of the neural network that maps the input to the hidden layer representation, and the *decoder* maps the hidden layer representation to an approximate reconstruction of the input at the output. Figure 1.1 below shows the architecture of an autoencoder. Initially, an autoencoder may not seem useful since the objective is to obtain an input reconstruction, which will be slightly different than the input. However, once an autoencoder is trained, its bottleneck layer is of more interest than its reconstruction. Of course, during training, the reconstruction should have a minimum error w.r.t. the input, but, by training a neural network in this way, the bottleneck layer is forced to learn a reduced dimensional representation of the input. In other words, a trained autoencdoer can be used for dimensionality reduction. Using an autoencoder for dimensionality reduction is much different than other methods (such as PCA) because the network is learning a non-linear mapping (due to the activation functions) of the input to a reduced dimensional space. The new representation may capture aspects of the input that linear dimensionality reduction algorithms may fail to represent.

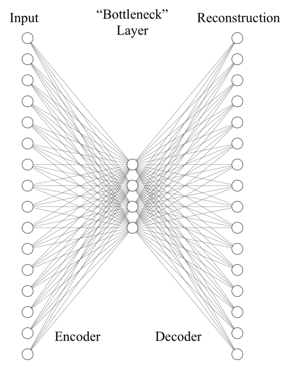

Figure 1.1: The architecture of an autoencoder.

In this project, we will implement various types of autoencoder neural networks using the MNIST Database of Hand Written Digits. This is a dataset of 70,000 images (each a 28×28 grayscale image of a hand written digit as shown below in Figure 1.2), which we will divide into a 60000/10000 train/test set. The objective of the autoencoders will be to reconstruct each image at the output with the smallest possible error.

For this project, the MNIST data, which can be loaded using Keras, is done for you in the available code. You are expected to fill in the Jupyter Notebook provided to you as well as generate all deliverables, as outlined in the following sections, in the Notebook. **Submit your final**

Figure 1.2: Example images from the MNIST dataset.

code as a single .ipynb file. Ensure that the deliverables are clearly shown for each part to earn full credit. Name your submission *lastname_firstname_autoencoders.ipynb*.

2 Example Autoencoders using Keras in Python

Below is an example of how to create and train an autoencoder consisting of fully connected layers using Keras in Python with an input dimension of 100, a hidden layer dimension of 25, and an output dimension of 100 (the output will always be the same dimension as the input since our goal is to reconstruct the input at the output). Beginning on line 70, see how to implement an autoencoder consisting of convolutional layers as well.

Listing 1: Autoencoders in Keras

```
 1  #ECE 595 Machine Learning II
 2  #Project 1: Autoencoders - Example Code
 3  #Written by: Rajeev Sahay and Aly El Gamal
 4
 5  from keras.models import Sequential
 6  from keras.layers import Dense
 7  from keras import backend as K
 8  import matplotlib.pyplot as plt
 9
10  # Assume data_train is a matrix with shape (num_samples, 100)
11  # Assume data_test is a matrix with shape (num_samples, 100)
12
13  # Create a fully connected autoencoder architecture
14  def autoencoder():
15      model = Sequential()
16
17      # Encoder
18      model.add(Dense(25,
19                      activation="choose_activation",
20                      use_bias=True,
21                      kernel_initializer="uniform",
22                      input_dim=100))
23
24      # Decoder
25      model.add(Dense(100,
26                      activation="choose_activation",
27                      kernel_initializer="uniform"))
28
29      return model
30
31  # Create an autoencoder graph
32  ae_model = autoencoder()
33
34  # Compile the graph using a particular loss function and optimizer algorithm
35  ae_model.compile(loss="choose_loss_fcn",
36                   optimizer="choose_optimizer_alg")
37  # Training parameters
38  num_epochs = 100
39  batch_size = 1024
40
41  # Train the model using the inputs as the target outputs
42  ae_model_history = ae_model.fit(data_train, data_train,
                                        validation_data=(data_test, data_test),
                                        epochs=num_epochs,
                                        batch_size=batch_size,
                                        shuffle=True)

# Generate predicted reconstructions using the trained model
reconstructions = ae_model.predict(data_test)

# Obtain the hidden layer representation of the input data:
get_hl = K.function([ae_model.layers[0].input], [ae_model.layers[0].output])
data_test_hidden_layer_rep = get_hl([data_test])[0]

# Plot training history of loss versus epochs
plt.plot(ae_model_history.history['loss'])
plt.plot(ae_model_history.history['val_loss'])

# Plot first three data samples in data_test as graysclae image
n = 3  # how many digits we will display
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.imshow(data_test[i].reshape(10, 10))
    plt.gray()
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)


# One can create a convolutional autoencoder according to the
# following architecture, which can be trained identically to
# the fully connected model above

# Assume data_train is a matrix with shape (num_samples, 10, 10, 1) here
# Assume data_test is a matrix with shape (num_samples, 10, 10, 1) here

# Use UpSampling 2D for upsampling layer
from keras.layers import Conv2D, MaxPooling2D, UpSampling2D

def convolutional_autoencoder():
    model = Sequential()

    #Encoder
    model.add(Conv2D(64,
                     kernel_size=4,
                     activation="choose_activation",
                     padding="same",
                     input_shape=(10, 10, 1)))
    model.add(MaxPooling2D(pool_size=(2, 2),
                           padding="same"))

    #Decoder
    model.add(Conv2D(64,
                     kernel_size=4,
                     activation="choose_activation",
                     padding='same'))
    model.add(UpSampling2D((2, 2)))

    #Reconstruction
    model.add(Conv2D(1,
                     (3, 3),
                     activation='choose_activation',
                     padding='same'))

    return model
```

3 Part 1: Fully Connected Autoencoder

Implement a fully connected autoencoder according to the following specifications:

- Input layer: 784 perceptrons (reshape each image so that it is a flattened 784-dimensional vector)
- First hidden layer: 400 perceptrons
- Second hidden layer: 200 perceptrons
- Third hidden layer ("bottleneck" layer): 100 perceptrons
- Fourth hidden layer: 200 perceptrons
- Fifth hidden layer: 400 perceptrons
- Output layer (reconstruction): 784 perceptrons

You will have to make decisions about which activation functions to use at each layer, whether or not to use a bias for each layer, which loss function to use for training, which optimizer to use for updating the parameter values during training, and the batch size. **Train your model using 150 epochs.**

Deliverables for Part 1:

1. Show a plot of the model loss versus the epochs for both the training and validation data (the validation data here can be the testing data). Include a legend, appropriate axes labels, and a title.
2. Show images of the first ten samples in the testing set
3. Show images of the "bottleneck" layer representation of the first ten images in the testing set (reshape the images to be 10 x 10)
4. Show images of the reconstructed samples of the first ten images in the testing set

looks at samples in between the layersof activatic

4 Part 2: Convolutional Autoencoder

Implement a convolutional autoencoder according to the following specifications:

- Input layer that accepts a 28 × 28 grayscale image
- First hidden layer: A convolutional layer with 16 feature maps and a 3 × 3 filter size
- Second hidden layer: A 2 × 2 maxpooling layer
- Third hidden layer: A convolutional layer with 8 feature maps and a 3 × 3 filter size
- Fourth hidden layer (“bottleneck” layer): A 2 × 2 maxpooling layer
- Fifth hidden layer: A convolutional layer with 8 feature maps and a 3 × 3 filter size
- Sixth hidden layer: A 2 × 2 upsampling layer
- Seventh hidden layer: A convolutional layer with 16 feature maps and a 3 × 3 filter size
- Eighth hidden layer: A 2 × 2 upsampling layer
- Output layer (reconstruction): A convolutional layer with a single feature map and a 3 × 3 filter size

You will have to make decisions about which activation functions to use at each layer, whether or not to use a bias for each layer, which loss function to use for training, which optimizer to use for updating the parameter values during training, and the batch size. **Train your model using 150 epochs.**

Deliverables for Part 2:

1. Show a plot of the model loss versus the epochs for both the training and validation data (the validation data here can be the testing data). Include a legend, appropriate axes labels, and a title.
2. Show images of the first ten samples in the testing set
3. Show images of the "bottleneck" layer representation of the first ten images in the testing set (reshape the images to be 28 × 14)
4. Show images of the reconstructed samples of the first ten images in the testing set

5 Part 3: Denoising Autoencoder

An autoencoder can be used to filter noise (denoise) from data samples. The architecture of a denoising autoencoder is no different than that of a regular autoencoder. In fact, the only difference between a normal autoencoder and a denoising autoencoder is the training data. A denoising autoencoder is trained to filter noise from the input and produce a denoised version of the input as the reconstructed output. This is achieved during training by using the noisy sample as the input and the original noise-free sample as the target reconstruction. In Part 3, you are going to implement a denoising autoencoder that can filter Gaussian noise injected into the MNIST dataset. The noisy data is created by generating 784 values, per sample, from a zero mean unit variance Gaussian distribution, scaling them by 0.25 (to make the additional noise more intense), and adding them to each point in the original image. Each image is then clipped to stay in the valid pixel range. The autoencoder will then be trained to eliminate this added noise.

Begin by selecting an architecture (you may use your architecture from Part 1 or Part 2 or create a new architecture) for your denoising autoencoder. Then, compile and train the model appropriately using the noisy data as inputs and the noise-free data as the target labels. Finally, show ten samples of original images from the testing set, their corresponding noisy counterparts, and their denoised DAE outputs.

Deliverables for Part 3:

1. Show images of the first ten samples in the training set
2. Show images of the first ten samples in the training set after injecting Gaussian noise
3. Show a plot of the model loss versus the epochs for both the training and validation data (the validation data here can be the testing data) of the DAE. Make sure to include a legend, appropriate axes labels, and a title.
4. Show images of the first ten samples in the testing set
5. Show images of the first ten samples in the testing set after adding noise
6. Show images of the first ten samples in the testing set after "denoising"