CS 615 - Deep Learning

Assignment 1 - Forward Propagation
Winter 2023

Introduction

In this first assignment we'll begin designing and implementing the modules we'll use in our deep learning architectures. In addition, we'll start getting used to importing datasets.

Allowable Libraries/Functions

Recall that you **cannot** use any ML functions to do the training or evaluation for you. Using basic statistical and linear algebra function like *mean*, *std*, *cov* etc.. is fine, but using ones like *train* are not. Using any ML-related functions, may result in a **zero** for the programming component. In general, use the "spirit of the assignment" (where we're implementing things from scratch) as your guide, but if you want clarification on if can use a particular function, DM the professor on discord.

Grading

Theory	15pts
Testing layers independently	50pts
Basic test on connected layer set	20pts
Test with medical cost dataset on connected layer set	15pts
TOTAL	100pts

Table 1: Grading Rubric

Datasets

Medical Cost Personal Dataset (mcpd_augmented.csv) This dataset consists of data for 1338 people in a CSV file. The original dataset, found at https://www.kaggle.com/mirichoi0218/insurance, contains the following information for each person:

1. age
2. sex
3. bmi
4. children
5. smoker
6. region
7. charges

For the purposes of this assignment, we have converted the *sex* and *smoker* features into binary features and the *region* into an enumerated feature $\in \{0, 1, 2, 3\}$. In addition, we omitted the *charges* information (in a subsequent assignment we will look to predict this).

1 Theory

1. Given a single input observation $x = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$ and a fully connected layer with weights of $W = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$ as biases $b = \begin{bmatrix} -1 & 2 \end{bmatrix}$, what are the output of the fully connected layer given x as its input (5pts)?

2. Given an input, $h = [10, -1]$, what would be the output(s) if this data was processed by the following activation functions/layers (10pts)?

   (a) Linear

   (b) ReLu

   (c) Sigmoid

   (d) Hyperbolic Tangent

   (e) Softmax

2 Start Implementing Your Layers

In a **single python file** called *layers.py*, place the code for the abstract base class.

```
from abc import ABC, abstractmethod
class Layer(ABC):
  def __init__(self):
    self.__prevIn = []
    self.__prevOut = []

  def setPrevIn(self,dataIn):
    self.__prevIn = dataIn

  def setPrevOut(self, out):
    self.__prevOut = out

  def getPrevIn(self):
    return self.__prevIn;

  def getPrevOut(self):
    return self.__prevOut

  @abstractmethod
  def forward(self,dataIn):
    pass

  @abstractmethod
  def gradient(self):
    pass

  @abstractmethod
  def backward(self,gradIn):
    pass
```

Now let's implement our layers, having them derive from the *Layer* class!

InputLayer

Implement a class called *InputLayer* that inherits from your abstract base class. Here's the class' public interface:

```
class InputLayer(Layer):
  #Input:  dataIn, an NxD matrix
  #Output:  None
  def __init__(self,dataIn):
    #TODO

  #Input:  dataIn, an NxD matrix
  #Output: An NxD matrix
  def forward(self,dataIn):
    #TODO

  #We'll worry about these later...
  def gradient(self):
    pass

  def backward(self,gradIn):
    pass
```

This class' constructor should take as a parameter an entire training dataset (as an $N \times D$ matrix), and initialize two attributes, *meanX* and *stdX* to be row vectors of the mean and standard deviation, respectively, of the features of your training dataset. For numeric stability, set any standard deviations that are zero to 1 (this will avoid divide-by-zero issue when zscoring).

In addition, you must implement the abstract method *forward* such that it takes a data matrix, X, as a parameter, computes the *zscored* version of this data using the *meanX* and *stdX* attributes, stores the input and output in attributes (for later use), and returns the zscored data.

Activation Layers

Next implement classes for the following activation functions:

- *LinearLayer*
- *ReLuLayer*
- *LogisticSigmoidLayer*
- *SoftmaxLayer*
- *TanhLayer* (Hyperbolic Tangent Function)

The public interface for each should be (where XXX is the name of the activation function, according to the bullets above):

```
class XXXLayer(Layer):
  #Input:  None
  #Output:  None
  def __init__(self):
    #TODO

  #Input:  dataIn, an NxK matrix
  #Output:  An NxK matrix
  def forward(self,dataIn):
    #TODO

  #We'll worry about these later...
  def gradient(self):
    pass

  def backward(self,gradIn):
    pass
```

Each class should inherit from the *Layer* abstract base class, initializing its constructor within its own constructor, and must implement the *forward* method that takes in data as a parameter, sets its parent class' previous input to that data, computes the output values, setting the parent class' previous output to that, and returns that output. **Make sure you use the correct names for your classes and methods** since we import your modules in our own automated grading scripts.

FullyConnectedLayer

Finally, let's create a class for a fully connected layer, aptly called *FullyConnectedLayer*. This too should inherit from *Layer*. Following the material in the lecture slides, this class should have two attributes, a weight matrix and a bias vector. The constructor should take in two explicit parameters, the number of features coming in, and the number of features coming out of this layer, and use these to initialize the weights (and biases, which are technically weights as well) to be random values in the range of $\pm 10^{-4}$. Its *forward* method once again takes in data X, storing it in its parent's previous input attribute, and computes the output (storing it with the parent class, and returning it) as:

$$Y = XW + b$$

where W is the weight matrix and b is the bias vector. For now, the *gradient* method may return nothing. In addition provide getter and setter methods for the weight and bias attributes. Here's the public interface:

```
class FullyConnectedLayer(Layer):
  #Input:  sizeIn, the number of features of data coming in
  #Input:  sizeOut, the number of features for the data coming out.
  #Output:  None
  def __init__(self, sizeIn, sizeOut):
    #TODO

  #Input:  None
  #Output: The sizeIn x sizeOut weight matrix.
  def getWeights(self):
    #TODO

  #Input: The sizeIn x sizeOut weight matrix.
  #Output: None
  def setWeights(self, weights):
    #TODO

  #Input:  The 1 x sizeOut bias vector
  #Output: None
  def getBiases(self):
    #TODO

  #Input:  None
  #Output: The 1 x sizeOut biase vector
  def setBiases(self, biases):
    #TODO

  #Input:  dataIn, an NxD data matrix
  #Output:  An NxK data matrix
  def forward(self,dataIn):
    #TODO

  #We'll worry about these later...
  def gradient(self):
```

```
    pass

def backward(self,gradIn):
    pass
```

3 Testing the layers

Let's test each layer using the following input data (this can be considered two observations, with four features per observation):

$$X = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}$$

We'll create this as a numpy array:

$$X = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])$$

Write a script that does the following:

1. Imports your *layers* module.
2. Creates the data shown above.
3. Instantiates each your layers. For the fully-connected layer, we'll have its output size be 2.
4. Sets the fully-connected layer's weights to $W = \begin{bmatrix} 1 & 2 \\ 2 & 0 \\ 1 & 1 \\ -1 & 4 \end{bmatrix}$, and $b = \begin{bmatrix} 1 & 2 \end{bmatrix}$
5. Forward propagates the input data X through each layer *independently* and outputs their return values.

In your report provide:

- Output of input layer
- Output of fully connected layer (output is of size two)
- Output of reLu activation layer
- Output of logistic sigmoid activation layer
- Output of softmax activation layer
- Output of tanh activation layer

4 Connecting Layers and Forward Propagate

Now let's assemble a simple network and forward propagate data through it!

Our architecture will be:

Input→FC (2 outputs)→Logistic Sigmoid

We'll once again use the data X from our prior problem as the input to this pipeline, and set the weights and biases as previously mentioned. From an implementation standpoint, you'll likely want to create instances of your classes and organize them in some sort of an ordered structure such that the output of one layer is the input of the next. See the example code at the end of the *Building Blocks* slides.

In your report provide the output from each layer.

5 Testing on full dataset

To test your implementation on a real dataset, we'll use the augmented medical cost dataset mentioned earlier in the assignment. Read in the dataset as your input data X and pass it through the architecture from the previous problem (two outputs from FC layer). This time **no not** manually set the weights and biases of the fully connected layer. Instead let them keep their random values assigned during the layer's instantiation.

In your report, just provide the output of the last layer pertaining to the **first observation**.

Submission

For your submission, upload to Blackboard a single zip file containing:

1. PDF Writeup
2. Source Code
3. readme.txt file

The readme.txt file should contain information on how to run your code to reproduce results for each part of the assignment.

The PDF document should contain the following:

1. Part 1:
   (a) Your solutions to the theory question
2. Parts 2: Nothing
3. Part 3:
   (a) The output for each layer when using the provided X as its input.
4. Part 4:
   (a) The output of the each layer, when given input X as the input to the first layer.
5. Part 5:
   (a) The output pertaining to the first observation from the final layer, when given the augemented medical cost dataset as its input.