Project 3

CMSC 421, Fall 2019

Andreas Verdelis

Last update: November 11, 2019

- Due date: December 2
- Late date: December 4

What you'll need to do

- Implement and train a basic neural network with backpropagation
  - We'll give you part of the code, you need to fill in the details

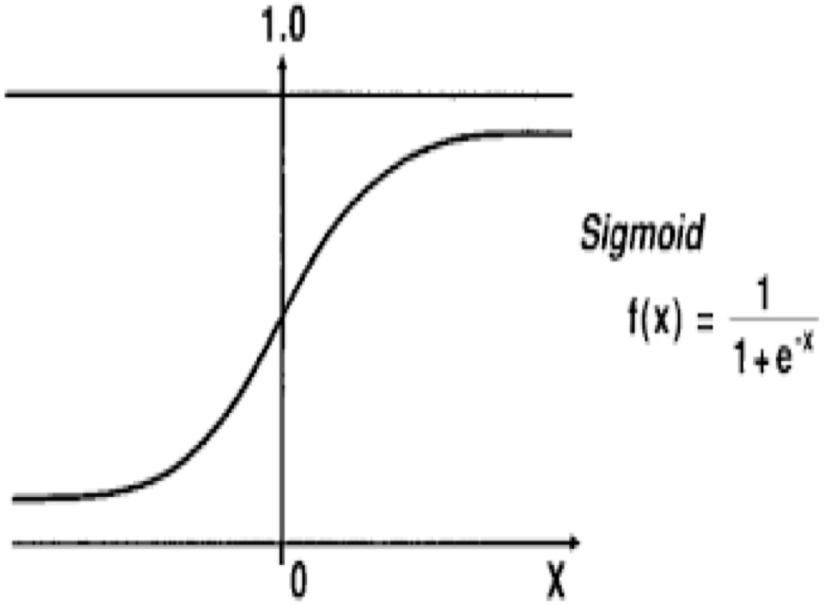
- We'll give you a ZIP archive containing two files:
  - studNet.py: the class and method outlines
  - studenttest.py: code to train and test your neural network
- studNet.py contains some, but not all, of the necessary methods
  - You'll need to write the others yourself

Methods/Classes provided in studNet.py

activation(z):

- This is the activation function used in the feed-forward compution of the network.
- The included function is the sigmoid, but you can change this as you see fit.

Here is a graph showing visually how the sigmoid function works:



Methods/Classes provided in studNet.py (continued)

sigderiv(z):

This is the derivative of the sigmoid function

- You'll need to use it to update the values when training your neural network
- If you use an activation function other than the sigmoid, then you'll need to use a different derivative than the one returned by sigderiv

Methods/Classes provided in studNet.py (continued)

Class: neuralnetwork:

__init__(self, size, seed):

- This function will initialize the weights and biases for a neural network of the size specified by the ‘size’ parameter.
- The ‘size’ parameter is a list of the form
  - inputsize, hidden layer 1 size, … , hidden layer n size, output size]
- Example on next page

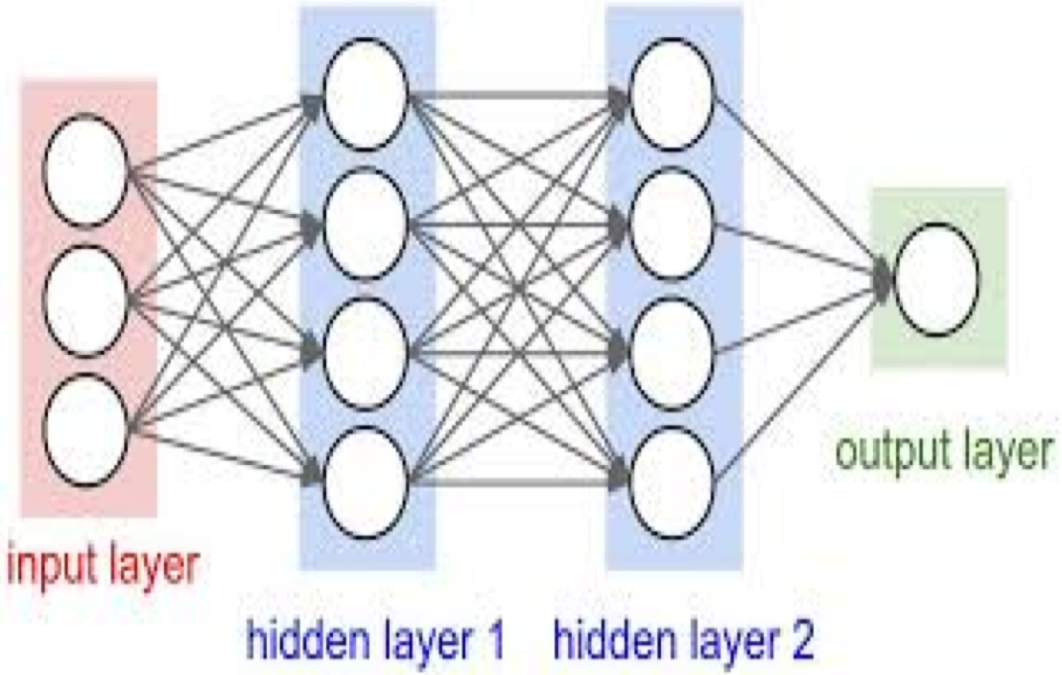
Example of __init__

Suppose size = [3, 4, 4, 1]

- then __init__ creates the network shown at right
- It initializes a variable

  self.weights = [[v_{11} v_{12} v_{13}] [v_{21} v_{22} v_{23}] [v_{31} v_{32} v_{33}] [v_{41} v_{42} v_{43}] …]

- each v_{ij} is the weight on the connection to unit i in the first hidden layer from unit j in the input layer



Methods/Classes provided in studNet.py (continued)

forward(self, input):

Given a vector of input parameters of form:

[[parameter 11, p12, ..., p1n] [p21, p22, …, p2n] …]

This method will return a 3-tuple:

- the output value(s) of each example (the variable ‘a’ in the source code)
- The values before activation was applied after the input was weighted (the variable ‘pre_activations’)
- The values after activation for all layers (the variable ‘activations’)

- The reason it returns ‘pre_activations’ and ‘activations’ is because you’ll need them for updating the network

What you have to do:

- Implement a method called train(…)
  - test_train calls it to train the neural network on the data set
  - To do this training, you will need to perform back propagation and calculate deltas

- We've provided method headers for train(), backpropagate(), and calcDeltas()
  - If you want to use them, you'll need to write the method bodies
  - But you don't have to use them if you want to implement your network differently

- The only things that must stay the same for testing:
  - you **MUST** have the **test_train** method and **predict** as provided
  - test_train must return an instance of your neural network that can be used to call predict(a).

Guidelines

What you will need to have in a minimally functioning neural network:

- You must implement some form of back propagation
- You must compute deltas (error) in some way in order to update your network
- You must choose a cost function to use in updating your network

Guidelines Cont.

Some Variables You Might Want to Think About Optimizing Are:

- alpha (the learning rate)
- the number of iterations your training algorithm takes (while there are no hard limits on time, if your training does too many iterations it will overfit the data and perform badly)

Suggestions:

Here are some things you will want to think about implementing to maximize the effectiveness of your network:

- An early stopping feature
- Regularization
- A good cost function
- A validation set in training to tune the hyperparameters

How you can train and test your network

We’ve included a testing class: studenttest.py

- Trains and tests your network using the sklearn ‘make moons’ dataset.
- To test your network across a wider variety of examples, you can change the values of samples and noise in .make_moons()
  - add more noise to make the problem harder
  - change the number of data samples

How we'll train and test your network

- studenttest.py isn't guaranteed to be the exact dataset we use for grading
  - But the one we use for grading will be similar
  - Good performance on studenttest.py will translate to good scores on our tests

- Any data we use in testing will be in the same format that studenttest.py uses for X and y data
  - In our tests, y is guaranteed to be 0 or 1
  - But X might have more parameters than it does in studenttest.py

Evaluation criteria

- 35% correctness: – whether your network works correctly, whether your submission follows the instructions
- 15% programming style
- 15% documentation
  - Docstrings at start of the file and in each function; comments elsewhere
- 35% on performance
  - Does your program crash?
  - How long to train?
  - How well does it classify our test data?
- We'll consider giving extra credit to the top n performers (n ≈ 3 to 5)