F. Wermelinger
Office: Pierce 211

Pair-Programming 4

Python closure, Fully connected neural networks

Issued:	September 23, 2022
Due:	October 7, 2022 11:59pm

In this pair-programming session you will consider a simple implementation of a fully connected neural network and perform a forward pass of random input data.

You should work on the exercises in groups of 3 to 4 students via a `tmate` session. Your team members can submit the same file. Please indicate your names in a header in the files. See the tutorials on the class website for an example pair-programming workflow.[1] Do not forget to commit and push your work when you are done. Ensure that you are on your *default branch* for this and not, possibly, on your homework branch.

Exercise 1: Python Closure for Neural Network Layer

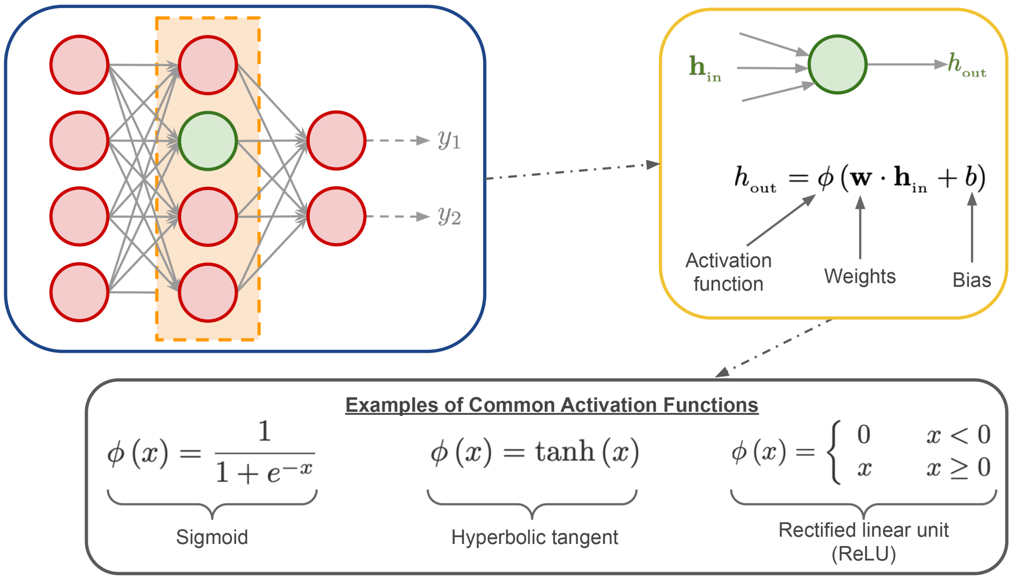
Deliverables:

1. `exercise_1.py`

The goal in this exercise is to build a hidden layer object using Python closures for simple, fully connected neural networks. For those unfamiliar with neural networks, you can think of them as functions with many fitting parameters. You want this function to represent some data. In its basic form, you pass your data through the neural network and get a prediction. Then you compare this prediction against the true data (ground truth) using a *loss function*. The loss function tells you how good your function is at approximating the data. Finally, you update the parameters of the neural network such that the prediction of the neural network results in a smaller loss function for the following iteration. Figure 1 illustrates some of the key components behind a basic neural network.

Your task will be to create a Python closure that models a layer of the neural network, see figure 1. The user will instantiate a layer by passing in the size of the layer (number of neural units) as well as the associated *activation function*. Later on, the layer can be used by calling the instantiated object. When the layer is used, the user need to pass in the inputs to the layer as well as the weights and biases for each unit in the layer. The result of calling an instantiated layer object will be an activated output of the appropriate size. Create a Python closure for a fully connected neural network layer with the following API:

[1] https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-pp



y_1
y_2
$\mathbf{h}_{\text{in}}$
h_{out}
$h_{\text{out}} = \phi\left(\mathbf{w} \cdot \mathbf{h}_{\text{in}} + b\right)$
Activation function
Weights
Bias
Examples of Common Activation Functions
$\phi(x) = \frac{1}{1+e^{-x}}$
Sigmoid
$\phi(x) = \tanh(x)$
Hyperbolic tangent
$\phi(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
Rectified linear unit (ReLU)

FIGURE 1: Key components for a fully connected neural network. A hidden layer is indicated by the orange rectangle in the top left.

- The outer function should return a layer function object. The outer function should take two arguments:

  shape: a list-like object of two numbers, where the first number corresponds to the number of inputs to the layer and the second number corresponds to the number of neural units in the layer.

  activation: an activation function (recall that functions are first class objects in Python).

- The returned layer function should return the layer outputs. The layer outputs are simply the outputs of each neural unit in the layer. The layer function accepts three arguments:

  inputs: the inputs to the layer. This should be a NumPy array.[2]

  weights: the weights for the layer. This should be a matrix of size `shape`.

  bias: the bias for each neural unit in the layer. This should be a vector of size `shape[1]` (the number of neural units in the layer).

- Moreover, the layer function should assert that the passed arguments have the correct size and dimension. Raise an exception if there is a dimension mismatch for the passed arguments.

The following is an example how a user is expected to interact with this object:

```
# Define network layers
layer1 = make_layer(shape1, np.tanh)
layer2 = make_layer(shape2, np.tanh)

# Initialize weights and biases
```

[2]https://numpy.org/

```
w1 = ...
w2 = ...
b1 = ...
b2 = ...

# Run through the network
inp = np.random.uniform(0.0, 1.0, 100).reshape(1, -1) # input to the network
out = layer2(layer1(inp, w1, b1), w2, b2)
```

Write a similar application code in your `exercise_1.py` file including some test code to catch exceptions for invalid input dimensions.

This is of course a very simplified implementation of a neural network. Here is an example of how it relates to a real application:

- The input could be an image. The first thing to do is *reshape* that image into a one dimensional vector.
- Now run that image through the network like we did above. This process can be automated more than what is shown in the example.
- The last layer outputs a prediction. In the example above, it was not defined what the output should be. In a real application, it may be be a vector representing *image classes*. For example, in the MNIST[3] database the output layer would have 10 neural units for the 10 possible digits.
- The prediction is fed into a loss function and compared with actual labeled data. Notice that loss functions are not taken into account in this exercise.
- Take the derivative of the loss function with respect to the weights and biases in order to update the weights and biases for the next iteration when the neural network is *trained*. Again, this is not part of this exercise but *computing this derivative is the motivation behind the class project*.
- Continue at the top for the next iteration until the loss function output has reached the desired threshold. At this point the network has been trained (its weights and biases) and can be used in another application for image classification as an example.

The closure defines the characteristic of each layer, i. e., its size and the activation function of the neurons in the layer. Once the layer(s) have been *instantiated*, they can be reused in each iteration of the training process using updated weights and biases as input.

[3] https://en.wikipedia.org/wiki/MNIST_database