Coding Problem

Background: This problem will explore the concept of Artificial neural networks and parameter optimization. Read:

- Artificial Neural Networks:
  `https://en.wikipedia.org/wiki/Artificial_neural_network`
- Optimization:
  `http://scipy-lectures.github.io/advanced/mathematical_optimization`
- Approximation:
  `https://en.wikipedia.org/wiki/Universal_approximation_theorem`

Problem: First using Python and NumPy, and then using Tensorflow, implement a three-layer neural network and optimize its parameters to fit nonlinear functions. Investigate and characterize the dependence of the fitting ability on the number of network parameters. Specifically:

1. Write a function that implements the neural network, as a function of input stimulus x and network parameters b_0, w_1, b_1, w_2, and b_2, where the b_i are the biases at each layer and the w_is are the weights. Specifically, your function should implement:

   $$F(x) = (A((x + b_0) * w_1) + b_1) * w_2 + b_2.$$

   In this equation: A is a scalar activation function (typically the hyperbolic tangent $\tanh$), $*$ means matrix multiplication, x is a vector of length N, b_0 is a vector of length N, w_1 is a matrix of shape (N, I), b_1 is a vector of length I, w_2 is a matrix of shape (I, O) and b_2 is a vector of shape O. (These symbols are chosen of 'N' = 'input', 'I' = 'intermediate', and 'O' = 'output'.)

   Your function operate on NumPy array objects and use efficient NumPy array operations.

2. Define a cost function `Cost` that assesses how well a given setting of the parameters $(b_0, w_1, b_1, w_2, b_2)$ does at allowing several non-linear functions to be approximated by the neural network embodied by the function F. Specifically, you should define a *regularized least squares* cost function of the form:

   $$Cost((b_0, w_1, b_1, w_2, b_2), X, Y) = mean[(F(X) - Y)^2] + C \cdot ||w_2||^2$$

   where C is a positive scalar and $||\cdot||^2$ denotes square-vector norm. This function produced a large value when $F(X)$ differs from Y, but it penalizes large values in the coefficients of the intermediate layer, e.g. the cost increases when w_2 has large norm.

   Again your function should be implemented via NumPy and it should be efficient.

3. Define a function `dCost` that evaluates the *derivative* of the cost function with respect to the variables b_0, w_2, b_1, w_2 and b_2. In other words, your function should return as a NumPy array the partial derivatives $d\text{Cost}/dv$ for each variable v. You might find the package `PyAutoDiff` helpful here and the next problem section, as well.

4. Using the optimization tool "L-BFGS-B" provided in the the SciPy optimization package, optimize the parameters of the neural network to approximate the following nonlinear functions:

   - $h_1(x) = x^2$
   - $h_2(x) = x^3 - 10x^2 + x - 1$
   - $h_3(x) = x^{3/2} - 20x^{0.5} + 2 * x + 2$
   - $h_4(x) = 3x^{5/2} - 20x^{0.3} - 10x + 5$
   - $h_5(x) = \sin(\pi x)$
   - $h_6(x) = \cos(\pi x)$
   - $h_7(x) = \sin(2 * \pi x)$
   - $h_8(x) = \tan(\pi * (x + 0.5))$

   You should perform this approximation on the interval between 0 and 10, and evaluate your results at intervals of 0.1 units. Your network should have $N = 100$ input nodes. Since there are 8 nonlinear functions above, your network should have $O = 8$ outputs. The number of intermediate nodes is up to you. Set regularization constant $C = 1$.

   Specifically, you want to run an optimization like:

   ```
   result = scipy.optimize.fmin_l_bfgs_b(Cost,
                   x0 = (b0_0, w1_0, b1_0, w2_0, b2_0),
                   fprime = dCost, args = (X, Y))
   ```

   where X is the array `np.tile(np.arange(0, 10, .1), (N, 1))`, and Y is the data array `[`h_i`(x) for x in np.arange(0, 10, .1)]`, and `b0`$_0$`, w1`$_0$`, b1`$_0$`, w2`$_0$ and `b2`$_0$ are initial random guesses for your parameter values.

5. Graph the final error as a function of number of input nodes (N) as well as number of intermediate nodes (I). What have you learned? What happens if you remove the regularization by setting $C = 0$? Can you explain the relationship between what you've learned and the Universal Approximation Theorem (UAT) of Neural Networks?

6. Perform the same optimization with a tensorflow implementation of the above using tensorflow's built-in backpropagation via stochastic gradient descent.

How to present your solution: Please submit your solution in any way that allows us to run it ourselves. For example, you could sunmit:

- A python script that can be run by the interface `python scriptname.py`. To submit graphs you can either have the program create them (using e.g. matplotlib) or just submit graphed files with your code, or

- An `ipython` notebook containing all the code and graphs, or
- A public github repo with the code in it that we can pull.