CS 3430: SciComp with Py
Assignment 7
Training and Testing Artificial Neural Networks

Vladimir Kulyukin
Department of Computer Science
Utah State University

February 24, 2018

1 Learning Objectives

1. Artificial Neural Networks
2. Linear Algebra
3. Numpy

2 Introduction

In this assignment, we will design, train, and evaluate an artificial neural network (ANN) to recognize odd and even numbers. ANNs are widely used in various areas of AI and machine learning. If you understand how they work and the types of problems where they can be used, not only will you be a better computer scientist but also a better data scientist.

3 Building Synapse Weight Matrices

Recall from the two lectures on ANNs that an ANN is a stack of neuron layers that starts at the input layer and ends at the output layer with some number of hidden layers in between. The synapse weights that define how the consecutive layers communicate or feedfoward can be defined as 2D matrices. For example, if we have an ANN that has 2 neurons in the input layer, 3 neurons in the hidden layer, and 1 neuron in the output layer, then the synapse weights between the input layer and the hidden layer are a 2×3 matrix and the synapse weights between the hidden layer and the output layer are a 3×1 matrix. If we have an ANN with 8 neurons in the input layer, 3 neurons in the hidden layer, and 8 neurons in the output layer, the ANN's synapse weights can be represented as two matrices: an 8×3 matrix from the input layer to the hidden layer and a 3×8 matrix from the hidden layer to the output layer.

Implement the function `build_nn_wmats(layer_dims)` that takes a n-tuple of layer dimensions, i.e., the numbers of neurons in each layer of ANN and returns a $(n-1)$-tuple of weight matrices initialized with random floats with a mean of 0 and a standard deviation of 1 for the corresponding n-layer ANN. Here are a few test runs.

```
>>> wmats = build_nn_wmats((2, 3, 1))
>>> wmats[0]
array([[-0.66476894,  0.54290862, -0.04445949],
       [-0.51803961, -0.87631211,  0.2820124 ]])
>>> wmats[1]
```

```
array([[-0.16116445],
       [-0.55181583],
       [-0.56616483]])
>>> len(wmats)
2
>>> wmats = build_nn_wmats((8, 3, 8))
>>> len(wmats)
2
>>> wmats[0]
array([[-0.38380596, -1.22059231, -0.26049966],
       [-1.32474024,  0.14011499,  0.86672211],
       [ 3.41899775, -1.52939008,  0.36952701],
       [-0.38335483,  0.40123533,  1.23863721],
       [ 0.31817877, -1.38816843,  0.10774014],
       [ 0.02857123, -0.26562244, -1.0397514 ],
       [-0.19636436, -0.97511094, -0.98953965],
       [-0.46425178,  0.75145605,  0.04730575]])
>>> wmats[1]
array([[ 1.34137241, -1.34226443, -1.09963163, -0.29983641, -0.84395309,
        -2.25919743, -0.11766274, -0.88921309],
       [-0.69884047, -0.88099456,  0.57212951,  0.38200215, -0.79697418,
         0.78602093,  0.51487098,  0.30219318],
       [-0.50060092,  1.02075046, -0.34423742,  0.05115683, -0.26345156,
        -1.8147592 ,  1.98869102,  0.5423938 ]])
>>> wmats[0].shape
(8, 3)
>>> wmats[1].shape
(3, 8)
```

4 Logical Gates and Identity Functions

Let us recapture what we did in the two lectures on ANNs to build a foundation for a small ANN project described in the next section. Recall that we discussed training two $2 \times 3 \times 1$ ANNs that function as binary `AND`- and `OR`-gates.

To train an ANN we need the training data. In general, the training data consist of the inputs to the ANN and the true outputs, i.e., the outputs that the ANN should be able to produce after it is trained. In the case of the binary gate networks, the input is a 2-element binary array and the output is a 1-element array where 0 represents `False` and 1 represents `True`. The numpy array `X_GATE` is the input data. The array `y_and` and `y_or` represent the ground truths for the `AND`-gate and the `OR`-gate, respectively.

```
X_GATE = np.array([[0, 0],
                   [1, 0],
                   [0, 1],
                   [1, 1]])

y_and = np.array([[0],
                  [0],
                  [0],
                  [1]])

y_or = np.array([[0],
```

```
                [1],
                [1],
                [1]])
```

Let us define a function that returns a 2-tuple of weight matrices for a $2 \times 3 \times 1$ ANN and train the two networks with the `train_3_layer_nn(numIters, X, y, build)` functions discussed in class. Here is our network builder.

```
def build_231_nn():
    return build_nn_wmats((2, 3, 1))
```

The network builder function is passed as the last argument to `train_3_layer_nn` to build two nets.

```
>>> and_wmats = train_3_layer_nn(70000, X_GATE, y_and, build_231_nn)
>>> or_wmats = train_3_layer_nn(70000, X_GATE, y_or, build_231_nn)
>>> len(and_wmats)
2
>>> len(or_wmats)
2
>>> and_wmats[0]
array([[-3.50801105,  3.87732455, -6.22570247],
       [ 7.90848533,  2.3213691 ,  1.41739827]])
>>> and_wmats[1]
array([[ 11.10855803],
       [ -5.75516316],
       [-15.48881278]])
>>> or_wmats[0]
array([[-2.35262981,  3.66325681, -3.98302908],
       [-2.57531226,  3.30858754, -3.87543413]])
>>> or_wmats[1]
array([[ -5.52059659],
       [  6.70742807],
       [-11.85836956]])
```

We can now in position to test our networks. Toward that end, modify the function `fit_3_layer_nn` discussed in the second lecture on ANNs. The modified version should accept a 2-tuple of weight matrices `W1` and `W2` instead of using them as two seperate arguments. Recall that `W1` is a weight matrix from the input layer to the hidden layer and `W2` is a weight matrix from the hidden layer to the output layer. Below are two tests. The three columns are the input to the net, the output, and the ground truth.

```
>>> for i in xrange(len(X_GATE)):
        print X_GATE[i], fit_3_layer_nn(X_GATE[i], and_wmats), y_and[i]

[0 0] [0] [0]
[1 0] [0] [0]
[0 1] [0] [0]
[1 1] [1] [1]
>>> for i in xrange(len(X_GATE)):
        print X_GATE[i], fit_3_layer_nn(X_GATE[i], or_wmats), y_or[i]

[0 0] [0] [0]
```

```
[1 0] [1] [1]
[0 1] [1] [1]
[1 1] [1] [1]
```

Let us train an $8 \times 3 \times 8$ network to map an 8-element bit array with exactly one 1 to itself. The training data are as follows.

```
X8_ID = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                  [0, 1, 0, 0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 0, 0, 0],
                  [0, 0, 0, 0, 1, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0],
                  [0, 0, 0, 0, 0, 0, 1, 0],
                  [0, 0, 0, 0, 0, 0, 0, 1]])

y8_id = np.array([[1, 0, 0, 0, 0, 0, 0, 0],
                  [0, 1, 0, 0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 0, 0, 0],
                  [0, 0, 0, 0, 1, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0],
                  [0, 0, 0, 0, 0, 0, 1, 0],
                  [0, 0, 0, 0, 0, 0, 0, 1]])
```

Let us define a network builder and train it with `train_3_layer_nn`.

```
def build_838_nn():
    return build_nn_wmats((8, 3, 8))
```

```
>>> wmats = train_3_layer_nn(700000, X8_ID, y8_id, build_838_nn)
>>> for i in xrange(len(X8_ID)):
        print X8_ID[i], fit_3_layer_nn(X8_ID[i], wmats)
```

```
[1 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0] [0 1 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0] [0 0 1 0 0 0 0 0]
[0 0 0 1 0 0 0 0] [0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0] [0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0] [0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0] [0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1] [0 0 0 0 0 0 0 1]
```

Build and train a $9 \times 4 \times 9$ network that maps an 9-bit string with exactly one 1 to itself on the following data.

```
X9_ID = np.array([[1, 0, 0, 0, 0, 0, 0, 0, 0],
                  [0, 1, 0, 0, 0, 0, 0, 0, 0],
                  [0, 0, 1, 0, 0, 0, 0, 0, 0],
                  [0, 0, 0, 1, 0, 0, 0, 0, 0],
                  [0, 0, 0, 0, 1, 0, 0, 0, 0],
                  [0, 0, 0, 0, 0, 1, 0, 0, 0],
                  [0, 0, 0, 0, 0, 0, 1, 0, 0],
```

```
                    [0, 0, 0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 1]])

y9_id = np.array([[1, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 1, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 1, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 1, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 1, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 1, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 1, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 1, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 1]])
```

Here is how we can train and test this network.

```
>>> wmats = train_3_layer_nn(700000, X9_ID, y9_id, build_949_nn)
>>> for i in xrange(len(X9_ID)):
        print X9_ID[i], fit_3_layer_nn(X9_ID[i], wmats)
```

```
[1 0 0 0 0 0 0 0 0] [1 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0] [0 1 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0] [0 0 1 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0] [0 0 0 1 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0] [0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 1 0 0 0] [0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 0] [0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 1 0] [0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 1] [0 0 0 0 0 0 0 0 1]
```

5 Training a 4-Layer ANN to Recognize Even and Odd Binary Numbers

On to a small ANN project. Let us put our knowledge of ANNs to actual use and build a 4-layer ANN to recognize whether the binary representation of an integer between 0 and 128 is even or odd. Specifically, design and train an ANN to accomplish this task. There is only one requirement that your ANN must satisfy: it must have 4 layers so that the input layer is an 8-element binary array and the output of your ANN is a 2-element binary array. If the first bit in the output array is 1, then input is classified as even; if the second bit in the output array is 1, then the input is classified as odd. Obviously, if both bits in the output array of your ANN are 1's or 0's, then your ANN does not classify the corresponding output correctly.

There is no requirement on how many neurons the two hidden layers should have. They can have as few as a single neurons and as many as several million. This is your design decision. You can experiment with several numbers. Another number you will have to experiment is the number of iterations over which your ANN will be trained.

Implement your design in the function `build_even_odd_nn()` that returns a 3-tuple of weight matrices for your ANN: `W1`, `W2`, and `W3`, where `W1` is the weight matrix from the input layer to the first hidden layer, `W2` is the weight matrix from the first hidden layer to the second hidden layer, and `W3` is the weight matrix from the second hidden layer to the output layer. The weights in all three matrices should be initialized to random floats with a mean of 0 and a standard deviation of 1. Here is a sample call.

```
>>> wmats = build_even_odd_nn()
```

```
>>> len(wmats)
3
>>> type(wmats[0])
<type 'numpy.ndarray'>
>>> type(wmats[1])
<type 'numpy.ndarray'>
>>> type(wmats[2])
<type 'numpy.ndarray'>
```

We have to create the data. Toward that end, implement the function `create_nn_data()` that returns a 2-tuple of numpy arrays. The first is an array of 129 binary numpy arrays that contain the binary codes of the integers from 0 upto 128. The second is the ground truth array of 2-element binary numpy arrays that classify each array in the first array as an even or odd integer. Here is an example where we first generate the two data arrays and then display the first 10 element of each array.

```
>>> X, y = create_nn_data()
>>> len(X)
129
>>> len(y)
129
>>> X[:10]
array([[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 1],
       [0, 0, 0, 0, 0, 0, 1, 0],
       [0, 0, 0, 0, 0, 0, 1, 1],
       [0, 0, 0, 0, 0, 1, 0, 0],
       [0, 0, 0, 0, 0, 1, 0, 1],
       [0, 0, 0, 0, 0, 1, 1, 0],
       [0, 0, 0, 0, 0, 1, 1, 1],
       [0, 0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 1, 0, 0, 1]])
>>> y[:10]
array([[1, 0],
       [0, 1],
       [1, 0],
       [0, 1],
       [1, 0],
       [0, 1],
       [1, 0],
       [0, 1],
       [1, 0],
       [0, 1]])
```

Note the ground truth classifications in `y`. The binary encoding of 0, i.e., `[0, 0, 0, 0, 0, 0, 0, 0]`, is classified as `[1, 0]`, i.e., even, and the binary encoding of 3, i.e., `[0, 0, 0, 0, 0, 0, 1, 1]`, is classified as `[0, 1]`, i.e., odd.

After you have taken care of the data, implement the function `train_4_layer_nn(numIters, X, y, build)` that trains the 4-layer ANN built by the `build` function for the number of iterations `numIters` on the data `X` and the ground truth `y`. This function should return a 3-tuple of weight matrices for the trained network: `W1`, `W2`, and `W3`, where `W1` is the weight matrix from the input layer to the first

hidden layer, `W2` is the weight matrix from the first hidden layer to the second hidden layer, and `W3` is the weight matrix from the second hidden layer to the output layer.

Here is an example of how you can create the data and then use it to train your ANN. In the code below, `numIters` should be replaced with a positiv integer that you will have to discover by experiment. This is the number of iterations that gives optimal performance for your ANN.

```
>>> X, y = create_nn_data()
>>> even_odd_wmats = train_4_layer_nn(numIters, X, y, build_even_odd_nn)
>>> len(even_odd_wmats)
3
```

To test your ANN, write the function `fit_4_layer_nn(x, wmats, thresh=0.4, thresh_flag=False)` that takes an input `x` to a trained 4-layer ANN specified by the 3-tuple of weight matrices `wmats`, feedforwards this input throught the network and outputs the classification. The classification is output as a float array if `thresh_flag=False` and as a binary array is `thresh_flag=True`. If the classification is output as a binary array, then the thresholding is done so that each element in the output is thresholded to 1 if it is greater than `thresh` and 0 otherwise. Here are a few test runs.

```
>>> fit_4_layer_nn(X[0], even_odd_wmats)
array([  9.99070844e-01,   7.27296943e-04])
>>> fit_4_layer_nn(X[0], even_odd_wmats, thresh_flag=True)
array([1, 0])
>>> fit_4_layer_nn(X[3], even_odd_wmats)
array([  3.28728872e-04,   9.99673649e-01])
>>> fit_4_layer_nn(X[3], even_odd_wmats, thresh_flag=True)
array([0, 1])
```

Implement the function `is_even_nn(n, even_odd_wmats)` that takes a integer `n` in `[0, 128]` and a 3-tuple of trained weight matrices for your ANN and returns `True` if your ANN classifies it as even and `False` if it classifies it as odd. Below are several test runs.

```
>>> X, y = create_nn_data()
>>> even_odd_wmats = train_4_layer_nn(numIters, X, y, build_even_odd_nn)
>>> is_even_nn(0, even_odd_wmats)
True
>>> is_even_nn(1, even_odd_wmats)
False
>>> is_even_nn(2, even_odd_wmats)
True
```

Finally, implement the function `eval_even_odd_nn(even_odd_wmats)` that takes a 3-tuple of weight matrices of your trained 4-layer ANN, runs your network on all integers from 0 upto 128 and outputs the classification accuracy, i.e., the number of times your network classifies as an odd integer as odd and an even integer as even by comparing the output of `is_even_nn(n, even_odd_wmats)` with `n % 2 == 0` and `n % 2 != 0`.

```
>>> X, y = create_nn_data()
>>> even_odd_wmats = train_4_layer_nn(70000, X, y, build_even_odd_nn)
>>> eval_even_odd_nn(even_odd_wmats)
1.0
```

Appreciate what we have done! Although our ANN has been confined to `[0, 128]`, it has been trained to recognize the divisibility by 2.

6 What to Submit

Write your solutions in `cs3430_s18_hw07.py` and submit it via Canvas. State in your comments what is your best accuracy and for how many iterations your ANN should be trained.

Happy Hacking!