CS 5600/6600: Intelligent Systems
Assignment 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 7, 2019

Learning Objectives

1. Backpropagation
2. Implementing and Training simple ANNs

Problem 1 (1 point)

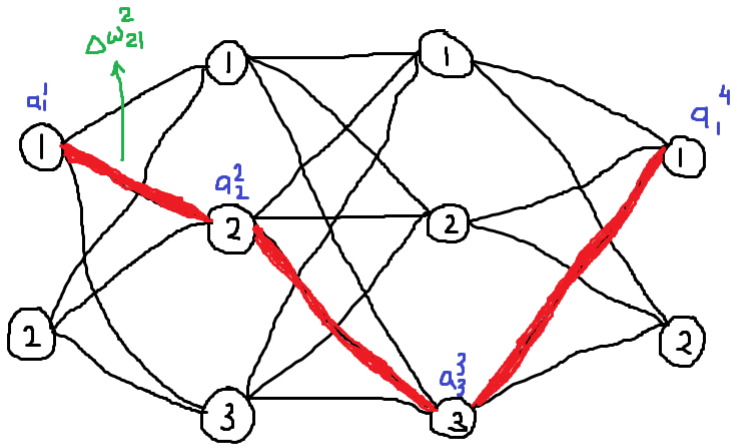

Figure 1: An activation path in a 2 x 3 x 3 x 2 ANN.

Consider a 2 x 3 x 3 x 2 ANN in Figure 1. Let's assume that our cost function C is defined as $C = (y - a)^2$, where y is the ground truth and a is the ANN's output. Suppose that we modify the weight w_{21}^2 by Δw_{21}^2. Give the formula for estimating ΔC along the activation path $a_2^{(2)}, a_3^{(3)}, a_1^{(4)}$ specified by the three red synapses. Note that when taking partial derivatives, you need to focus only on values along the specified path. For example, your cost function along the path is $C = (y - a_1^{(4)})^2$.

Type your solution to this problem in a math-friendly editor and save it in `hw02_f19_prob01.pdf`.

Problem 2 (2 points)

In this problem, you will implement and train several 3- and 4-layer ANNs that implement the feedforward and backpropagation equations discussed in lecture 2. This problem will lay a solid foundation for implementing and training more complex NNs with third-party packages as our journey into neural computation progresses and deepens.

Recall that, as we discussed in class, the synapse weights that define how consecutive ANN layers feedforward can be represented as 2D matrices. Suppose that we have an ANN with 2 neurons in the input layer, 3 neurons in the hidden layer, and 1 neuron in the output layer. If we use matrices to define synapse weights, we define the synapse weights between the input layer and the hidden layer as a 2×3 matrix and the synapse weights between the hidden layer and the output layer as a 3×1 matrix. If we have an ANN with 10 neurons in the input layer, 5 neurons in the hidden layer, and 100 neurons in the output layer, the ANN's synapse weights can be represented as two matrices: an 10×5 matrix from the input layer to the hidden layer and a 5×100 matrix from the hidden layer to the output layer.

Let's abstract into functions some useful ANN machinery. To begin with, 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)
```

Once we have this function, you can use it to define functions to build ANNs of various complexities. Here is how we can build 4 x 2 x 2 x 2 ANNs.

```
def build_4222_nn():
    return build_nn_wmats((4, 2, 2, 2))
```

The next piece of machinery we'll implement is our inputs and ground truths. In this problem, we will focus on 3- and 4-layer ANNs that take arrays 0's and 1's. For example, if we want to train ANNs for the binary `AND`, `OR`, and `XOR` boolean functions, then our input `X` and our ground truths `y` can be defined as follows.

```
# This is the input.
X1 = np.array([[0, 0],
               [1, 0],
               [0, 1],
               [1, 1]])

# This is the ground truth for the and function.
y_and = np.array([[0],
                  [0],
                  [0],
                  [1]])

# This is the ground truth for the or function.
y_or = np.array([[0],
                 [1],
                 [1],
                 [1]])

# This is the ground truth for the xor function.
y_xor = np.array([[0],
                  [1],
                  [1],
                  [0]])

# This the ground truth for the not function.
y_not = np.array([[1],
                  [0]])
```

Now implement the function `train_3_layer_nn(numIters, X, y, build)`. This function takes the number of iterations, `numIters`, the input `X`, the ground truth `y` for `X`, and an ANN builder function. This function uses `build` to build the appropriate number of weight matrices, and trains the ANN for the specified number of iterations on `X` and `y` using the feedforward and backpropagation equations discussed in class in lecture 2, and returns the trained matrices.

Here is how you can train a 2 x 2 x 1 ANN and a 2 x 3 x 1 ANN on the `XOR` problem for 100 iterations.

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

```
def build_221_nn():
    return build_nn_wmats((2, 2, 1))

>>> xor_wmats_231 = train_3_layer_nn(100, X1, y_xor, build_231_nn)
>>> xor_wmats_221 = train_3_layer_nn(100, X1, y_xor, build_221_nn)
```

Now implement the `train_4_layer_nn(numIters, X, y, build)` that behaves like the function `train_3_layer_nn`, except the ANN builder function which it takes as the 4-th argument builds a 4-layer ANN. Here is how you can train a 2 x 3 x 3 x 1 ANN to solve the `XOR` problem.

```
def build_2331_nn():
    return build_nn_wmats((2, 3, 3, 1))

>>> xor_wmats_2331 = train_4_layer_nn(100, X1, y_xor, build_2331_nn)
```

So far so good. We've now implemented the training procedures. The next step is to implement the testing procedures. In ANN terminology (and, more generally speaking, in machine learning), testing is referred to as *fitting*. Implement the function `fit_3_layer_nn(x, wmats, thresh=0.4, thresh_flag=False)` This function takes a sample input x given as a numpy array, a 2-tuple of trained weight matrices, a threshold float defaulting to 0.4 and a threshold boolean flag that defaults to `False`. The function feeds `x` forward through the weight matrices. If the `threshold_flag` is set to `False`, the output is given as is. If it is set to `True`, the output is thresholded so that each element of the output greater than the threshold is set to 1 and less than or equal to the threshold is set to 0.

Here is how we can train a 2 x 3 x 1 ANN for 300 iterations on the XOR problem and then test it without thresholding.

```
>>> X1
array([[0, 0],
       [1, 0],
       [0, 1],
       [1, 1]])
>>> xor_wmats_231 = train_3_layer_nn(300, X1, y_xor, build_231_nn)
>>> fit_3_layer_nn(X1[1], xor_wmats_231)
array([ 0.69526769])
>>> fit_3_layer_nn(X1[3], xor_wmats_231)
array([ 0.58161528])
>>> fit_3_layer_nn(X1[0], xor_wmats_231)
array([ 0.16727339])
>>> fit_3_layer_nn(X1[2], xor_wmats_231)
array([ 0.75587798])
```

It looks like we can safely threshold on 0.59. Let's do it.

```
>>> fit_3_layer_nn(X1[0], xor_wmats_231, thresh=0.59, thresh_flag=True)
array([0])
>>> fit_3_layer_nn(X1[1], xor_wmats_231, thresh=0.59, thresh_flag=True)
array([1])
>>> fit_3_layer_nn(X1[2], xor_wmats_231, thresh=0.59, thresh_flag=True)
array([1])
>>> fit_3_layer_nn(X1[3], xor_wmats_231, thresh=0.59, thresh_flag=True)
array([0])
```

Implement the function `fit_4_layer_nn(x, wmats, thresh=0.4, thresh_flag=False)`. This function takes a sample input x given as a numpy array, a 3-tuple of trained weight matrices, a threshold float defaulting to 0.4 and a threshold boolean flag that defaults to `False`. The function feeds `x` forward through the weight matrices. If the `threshold_flag` is set to `False`, the output is given as is. If it is set to `True`, the output is thresholded so that each element of the output greater than the threshold is set to 1 and less than or equal to the threshold is set to 0.

Let's train a 2 x 3 x 3 x 1 ANN on the XOR problem and fit it.

```
def build_2331_nn():
    return build_nn_wmats((2, 3, 3, 1))

>>> X1
array([[0, 0],
       [1, 0],
       [0, 1],
       [1, 1]])
>>> xor_wmats_2331 = train_4_layer_nn(500, X1, y_xor, build_2331_nn)
>>> fit_4_layer_nn(X1[0], xor_wmats_2331)
array([ 0.08663132])
>>> fit_4_layer_nn(X1[1], xor_wmats_2331)
array([ 0.91359799])
>>> fit_4_layer_nn(X1[2], xor_wmats_2331)
array([ 0.90882091])
>>> fit_4_layer_nn(X1[3], xor_wmats_2331)
array([ 0.06888355])
```

This looks way better than the results of my 2 x 3 x 1 ANN above! Of course, your floats may be different since the weights are initialized randomly. I can threshold on 0.1.

```
>>> fit_4_layer_nn(X1[0], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([0])
>>> fit_4_layer_nn(X1[1], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([1])
>>> fit_4_layer_nn(X1[2], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([1])
>>> fit_4_layer_nn(X1[3], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([0])
```

All we need to do now is to persist the trained ANN to a file and then read it back. We can do this with `pickle`. The function `save` persists the trained ANN to a file. The function `load` loads the persisted object into a running Python.

```
import pickle

def save(ann, file_name):
    with open(file_name, 'wb') as fp:
        pickle.dump(ann, fp)

def load(file_name):
    with open(file_name, 'rb') as fp:
        nn = pickle.load(fp)
    return nn
```

Here is how it works. I train a 2 x 3 x 3 x 1 ANN for 500 iterations on the `XOR` problem, persist it to a file, load it back into a running Python, and fit it again.

```
>>> xor_wmats_2331 = train_4_layer_nn(500, X1, y_xor, build_2331_nn)
>>> save(xor_wmats_2331, '/home/vladimir/AI/xor_wmats_2331.pck')
>>> loaded_wmats = load('/home/vladimir/AI/xor_wmats_2331.pck')
>>> fit_4_layer_nn(X1[0], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([0])
>>> fit_4_layer_nn(X1[1], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([1])
>>> fit_4_layer_nn(X1[2], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([1])
>>> fit_4_layer_nn(X1[3], xor_wmats_2331, thresh=0.1, thresh_flag=True)
array([0])
```

Train a 3-layer ANN and a 4-layer ANN for each of the following problems: binary `AND`, binary `OR`, binary `NOT`, and binary `XOR`. Use the the `save` function defined above to persist your trained ANNs in the following files.

1. `and_3_layer_ann.pck`, `and_4_layer_ann.pck`;
2. `or_3_layer_ann.pck`, `or_4_layer_ann.pck`;
3. `not_3_layer_ann.pck`, `not_4_layer_ann.pck`;
4. `xor_3_layer_ann.pck`, `xor_4_layer_ann.pck`;

Please do not change the names of these files. When we grade your assignments, we will run a script file that will load the above files into a running Python and run a few tests with `fit_3_layer_nn` and `fit_4_layer_nn`.

Finally, let's train a 3-layer ANN and a 4-layer ANN to evaluate a more complex boolean formula. The formula is

$$(x_1 \wedge x_2) \vee (\neg x_3 \wedge \neg x_4).$$

Create an input for this problem and save it in a numpy array in the variable `X4` in `hw02_f19_data.py`. Create the ground truth for this problem and save it in the variable `bool_exp` as an appropriate numpy array. Build and train a 3-layer ANN and a 4-layer ANN to solve this problem. Use the function `save` defined above to persist your ANNs in the files `bool_3_layer_ann.pck` and `bool_4_layer_ann.pck`.

Again, please do not change the names of these files. When we grade your assignments, we will run a script file that will load the above files into a running Python and run a few tests with `fit_3_layer_nn` and `fit_4_layer_nn`.

Save your Python code in `hw02_f19.py` and `hw02_f19_data.py`. In the comments in `hw02_f19.py`, state the structure of each of your ANNs (e.g., 2 x 3 x 1 or 2 x 4 x 4 x 1) and how many iterations it took you to train it. Place all your files in `hw02.zip` and submit it in Canvas. Your zip should contain the following files.

1. `hw02_f19_prob01.pdf`
2. `hw02_f19.py`, `hw02_f19_data.py`.

3. `and_3_layer_ann.pck`, `and_4_layer_ann.pck`;
4. `or_3_layer_ann.pck`, `or_4_layer_ann.pck`;
5. `not_3_layer_ann.pck`, `not_4_layer_ann.pck`;
6. `xor_3_layer_ann.pck`, `xor_4_layer_ann.pck`;
7. `bool_3_layer_ann.pck`, `bool_4_layer_ann.pck`.

Happy Hacking!