CS 5600/6600: Intelligent Systems
Assignment 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 12, 2020

Learning Objectives

1. Backpropagation
2. Implementing and Training Simple ANNs

Introduction

In this assignment, we'll implement and train several 3- and 4-layer ANNs that realize the feedforward and backpropagation equations discussed in lectures 2 and 3. No reading assignment for this week. We'll resume them next week. I'd like you to focus on coding and experimenting with these simple ANNs. Your work will lay a solid foundation for designing and training more complex NNs with third-party packages as our dive into neural computation progresses and deepens.

Problem 1: Building and Training ANNs ($2\frac{1}{2}$ points)

Recall from lectures 2 and 3 that the synapse weights define how consecutive ANN layers do the feedforwarding and 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 the synapse weights, we define the weights between the input layer and the hidden layer as a 2×3 matrix and the 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. By the way, `wmats` abbreviates "weight matrices." Here are a few test runs. Of course, your weights will be different, because they're randomly generated.

```
>>> 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 topologies. Here's 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 assignment, we'll focus on training 3- and 4-layer ANNs on problems that take arrays of 0's and 1's as inputs and require no data labeling. In other words, the ground truth is known. Specifically, we'll to train ANNs for the binary `AND`, `OR`, and `XOR` boolean functions. The input `X` and the ground truth `y` can be defined as follows. To spare you some typing, I've defined these for you in `cs5600_6600_f20_hw02_data.py`.

```
# 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]])
```

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 wmats, trains the ANN for the specified number of iterations on `X` and `y` using the feedforward and backpropagation equations, as discussed in lectures 2 and 3, and returns the trained matrices. Save your implementation in `cs5600_6600_f20_hw02.py`.

Here's how we 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)
```

Implement the `train_4_layer_nn(numIters, X, y, build)` that behaves exactly 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. Save your implementation in `cs5600_6600_f20_hw02.py`. Here's how we 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)
```

Problem 2: Fitting ANNs ($2\frac{1}{2}$ points)

OK. We've implemented the training procedures. The next step is to implement the testing procedures. In machine learning terminology, testing is frequently 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. Save your implementation in `cs5600_6600_f20_hw02.py`.

Here's 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 in this case we can safely threshold on 0.59 to distinguish between 0 and 1 on the output. 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. Save your implementation in `cs5600_6600_f20_hw02.py`.

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 will be different since the weights are initialized randomly. But in this case, we 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])
```

Perfect! All we need to do now is to persist the trained ANN to a file and then read it back. In general, persisting trained NNs is an important step. Sometimes you've been training an NN for days or weeks (sometimes months), and it's a shame if your work is wasted. We can persist trained NNs with the Py `pickle` module. The function `save` below 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's how it works. I'll train a 2 x 3 x 3 x 1 ANN for 500 iterations on the `XOR` problem, and 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 `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`;

Let me raise my text size at you for a second. DO NOT CHANGE THE NAMES OF THESE FILES! When the TA and I grade and evaluate your submissions, we'll write and run a Py script 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`. If you change the names of the persisted nets, the script will choke.

Finally, train a 3-layer ANN and a 4-layer ANN to evaluate a slightly more complex boolean formula to convince yourself that ANNs can be trained to compute arbitrary boolean expressions. The formula is

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

The input for this problem is in the variable `X3` in `cs5600_6600_f20_hw02_data.py`. The ground truth is in the variable `bool_exp` in the same file. 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`.

Save your Py code in `cs5600_6600_f20_hw02.py`. In the comments in `cs5600_6600_f20_hw02.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. Experiment with different architectures. Is it really true that the deeper we go, the faster it trains? Place all your files in `hw02.zip` and submit the zip in Canvas. To recap, your zip should contain the following files.

1. `cs5600_6600_f20_hw02.py`;
2. `and_3_layer_ann.pck`, `and_4_layer_ann.pck`;
3. `or_3_layer_ann.pck`, `or_4_layer_ann.pck`;
4. `not_3_layer_ann.pck`, `not_4_layer_ann.pck`;
5. `xor_3_layer_ann.pck`, `xor_4_layer_ann.pck`;
6. `bool_3_layer_ann.pck`, `bool_4_layer_ann.pck`.

A Few Suggestions

I've written several unit tests in `cs5600_6600_f20_hw02_uts.py`. You can use them when you work on your code. Don't run them all at once. Work them one at a time. Comment all of them out initially, uncomment one, and work on it until it passes. Then go on to the next one.

Some ANNs may not train the first time around, because weight initialization is random. If that is the case, you may want re-run a given unit test several times until it trains and then persist the net.

You don't need a GPU for this assginment. I did all the training and fitting on my 3-year old Dell

laptop with Intel i3-7100U CPU and 7.6GB of RAM running the 64-bit Ubuntu 18.04.5 LTS and Python 3.6.

Don't re-invent any wheels. Use numpy, which rests on the blood, sweat, and tears of at least 2 generations of C/C++ programmers. It's a great scientific computing tool, really!

Happy Hacking!