CS 5600/6600: F23: Intelligent Systems
Assignment 2

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 9, 2023

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 with the feedforward and backpropagation equations we discussed in our lectures on 09/04 and 09/06 to to lay a foundation for designing and training more complex NNs with third-party packages as our dive into "neural" computation deepens. I parenthesized the adjective *neural* to emphasize the fact that although this term is used in CS, there's no quarantee that this computation has anything in common with what's actually happening in the human brain or with the findings of Brain Science.

Problem 1 (1 pt)

Forty two years after the seminal paper "A Logical Calculus of the Ideas Immanent in Nervous Activity" by McCulloch and Pitts, three researchers (David Rumelhart, Geoffrey Hinton, and Ronald Williams) published another seminal paper "Learning Representations by Back-propagating Errors," where they formalized some of the ideas originally proposed by McCulloch and Pitts. This paper was one of the first ones to describe a modern version of backpropagation. While many backprop derivatives have been proposed since then, the basic ideas are as applicable today as in the mid 1980's. I included the PDF of a paper scan in the Assignment 02 zip.

Write a one page analysis of this paper. Your analysis should include four components: 1) a brief statement of the problem addressed in the paper; 2) what you liked in the paper and why; 3) what you did not like in the paper and why; 4) any inspirations you found in the paper. Each of these points should be addressed in a separate paragraph. There's no problem if you go over one page, but don't write an opus. Save your analysis in `rumelhart_et_al.pdf`.

Problem 2: Building and Training ANNs (2 pts)

Recall from our lectures this week that the synaptic weights define how consecutive ANN layers feedfoward 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 weights, we can define the weights between the input layer and the hidden layer as a

2×3 float matrix and the weights between the hidden layer and the output layer as a 3×1 float matrix.

Analogously, 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 weights can be represented as two matrices: a 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 do some functional abstraction to represent an ANN framework. 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 ANN layers. 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, we 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 ANN machinery we'll implement handles the 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. This is not representative of many real world situations insomuch as accurate data curation does require a lot of human labor. But, let's KIS (Keep It Simple) for now. We'll follow the inspirations from A Logical Calculus of the Ideas Immanent in Nervous Activity by McCulloch and Pitts, and train our ANNs to learn the binary boolean functions `AND`, `OR`, and `XOR`. Historically, these were the functions the first ANNs were trained to learn so we're following the evolution of AI. To spare you some typing, I've defined the input `X` and the ground truth `y` arrays for you in `cs5600_6600_f23_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` (make sure that your ground truth matches the input!), and an ANN builder function `build`. The function `train_3_layer_nn()` 1) uses `build` to construct the appropriate number of wmats, 2) trains the ANN for the specified number of iterations on `X` and `y` using the feedforward and backpropagation equations (See lectures 2 and 3 on Canvas or your class notes), and returns the trained matrices. Save your implementation in `cs5600_6600_f23_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)
```

I should note, in passing, that it took me 100 iterations to train this ANN on my 5-year old Dell laptop with the Intel Core i3-7100U CPU @ 2.40GHz × 4 and 7.6 GiB RAM. I use Python 3.6.7, numpy 1.16.3, on Ubuntu 18.04 LTS (Bionic Beaver). Your number of iterations may well be different. Recall our Numerical (In)stability Theorem that we constructively proved in Python at the end of the Lecture on 09/06. Neural networks are notoriously numerically unstable (i.e., running the same NN architecture on different platforms produces different accuracies). Keep it in mind as you train your NNs.

Implement the `train_4_layer_nn(numIters, X, y, build)` that behaves analogously to `train_3_layer_nn`, except that the ANN builder function `build` builds a 4-layer ANN. Save your implementation in `cs5600_6600_f23_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 pts)

OK. We've implemented the training procedures. The next step is to implement the testing procedures. In machine learning (ML) terminology, testing is frequently called *fitting* (e.g., fitting a trained model to a dataset).

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 (wmats), a threshold float defaulting to 0.4 (you can change that if you want) and a threshold boolean flag that defaults to `False`. The function feeds `x` forward through the wmats. If the `threshold_flag` is set to `False`, the output is given as is; if it's set to `True`, the output is thresholded so that each element of the output greater than the threshold `thresh` is set to 1 and less than or equal to the threshold is set to 0. Save your implementation in `cs5600_6600_f23_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 I can safely threshold on 0.59 (on my software/hardware platform!) to distinguish between 0 and 1 in the output. Let's try 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 as a numpy array, a 3-tuple of trained wmats, a threshold float defaulting to 0.4 and a threshold boolean flag with a default to `False`. The function feeds `x` forward through the wemats. If the `threshold_flag` is `False`, the output is given as is; if it's `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_f23_hw02.py`.

Here's a test. Let's fit a 2 x 3 x 3 x 1 ANN on the XOR problem.

```
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])
```

These floats look way better than the numbers of my 2 x 3 x 1 ANN above. Of course, your floats will be different since the weights are initialized randomly. It looks like I can threshold on 0.1. Let's do it.

```
>>> 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])
```

Picture Perfect! All we need to do now is to persist the trained ANN to a file and then read it back when we need to use it again. In general, persisting trained NNs is important, especially if you train an NN for days, weeks (sometimes months). Peristing intermediate results is a great remedy against power outages. We can persist trained NNs with the Py `pickle` module. By convention, pickled files have the extension `.pck`. The function `save` below gives an example of how to persist a trained ANN to a file. The function `load` loads the persisted object into 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 an example of how it all works. I'll 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 Python, and fit it.

```
>>> 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. DO NOT CHANGE THE NAMES OF THESE FILES! When Chris Allred, the GTA for this course, and I grade and evaluate your submissions, we'll run a script that will load the above files into 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 evaluate arbitrary boolean expressions, as McCulloch and Pitts argued in their article. The formula is

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

The input to this problem is in the variable `X3` in `cs5600_6600_f23_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 for 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_f23_hw02.py`. In the comments in `cs5600_6600_f23_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 on your software/hardware platform. You may want to experiment with different architectures. Here's a question for you to consider – Is it really true that the deeper we go, the faster (few iterations) 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_f23_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_f23_hw02_uts.py`. You can use them as you work on your code. Don't run them all at once. Work on one unit test at a time until your ANN passes it, then persist it, and go on to the next one.

Some ANNs may not train the first time around, because weight initialization is random. If that's the case, you may want to re-run a given unit test several times until your trained ANN passes and persists.

You don't need a GPU for this assginment. I did all the training and fitting on my 6-year old Dell laptop. The appendix below gives my output of unit tests 03–09 in `cs5600_6600_f23_hw02_uts.py`. I ran them one at a time.

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

Happy Reading, Writing, and Hacking!

My Output for Unit Tests 03–09

```
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 2x3x1 AND ANN Thresholded at 0.4 for 500 iters
[0 0], [0.01219127] --> [0]
[1 0], [0.06038905] --> [0]
[0 1], [0.06423048] --> [0]
[1 1], [0.90758932] --> [1]
.
----------------------------------------------------------------------
```

```
Ran 1 test in 0.013s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 2x3x3x1 AND ANN Thresholded at 0.4 for 500 iters
[0 0], [0.00318224] --> [0]
[1 0], [0.04184537] --> [0]
[0 1], [0.03694972] --> [0]
[1 1], [0.93171071] --> [1]
.
----------------------------------------------------------------------
Ran 1 test in 0.019s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 2x3x1 OR ANN Thresholded at 0.4 for 500 iters
[0 0], [0.07433633] --> [0]
[1 0], [0.95615286] --> [1]
[0 1], [0.95542007] --> [1]
[1 1], [0.98296453] --> [1]
.
----------------------------------------------------------------------
Ran 1 test in 0.014s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 2x3x3x1 XOR ANN Thresholded at 0.4 for 700 iters
[0 0], [0.14046884] --> [0]
[1 0], [0.85519024] --> [1]
[0 1], [0.81278467] --> [1]
[1 1], [0.17841459] --> [0]
.
----------------------------------------------------------------------
Ran 1 test in 0.025s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 1x2x1 NOT ANN Thresholded at 0.4 for 500 iters
[0], [0.83689998] --> [1]
[1], [0.30773786] --> [0]
.
----------------------------------------------------------------------
Ran 1 test in 0.012s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 4x2x1 BOOL EXP ANN Thresholded at 0.4 for 800 iters
[1 1 1 1], [0.79582833] --> [1]
[0 1 1 1], [0.00010448] --> [0]
[1 0 1 1], [0.00010449] --> [0]
[1 1 0 1], [0.90849228] --> [1]
[1 1 1 0], [0.90848952] --> [1]
[0 0 1 1], [3.1110484e-06] --> [0]
```

```
[0 1 0 1], [0.32458603] --> [0]
[0 1 1 0], [0.32458686] --> [0]
[1 0 0 1], [0.32459043] --> [0]
[1 0 1 0], [0.32459127] --> [0]
[1 1 0 0], [0.99999859] --> [1]
[1 0 0 0], [0.99998725] --> [1]
[0 1 0 0], [0.99998732] --> [1]
[0 0 1 0], [8.73697979e-06] --> [0]
[0 0 0 1], [8.73696002e-06] --> [0]
[0 0 0 0], [0.75640773] --> [1]
.
----------------------------------------------------------------------
Ran 1 test in 0.025s

OK
>>> python cs5600_6600_f23_hw02_uts.py
Training & Testing 4x2x2x1 BOOL EXP ANN Thresholded at 0.4 for 800 iters
[1 1 1 1], [0.8508243] --> [1]
[0 1 1 1], [0.0018756] --> [0]
[1 0 1 1], [0.00187445] --> [0]
[1 1 0 1], [0.97789194] --> [1]
[1 1 1 0], [0.97789317] --> [1]
[0 0 1 1], [0.00072752] --> [0]
[0 1 0 1], [0.06919275] --> [0]
[0 1 1 0], [0.06919265] --> [0]
[1 0 0 1], [0.06918431] --> [0]
[1 0 1 0], [0.06918422] --> [0]
[1 1 0 0], [0.9999996] --> [1]
[1 0 0 0], [0.99999955] --> [1]
[0 1 0 0], [0.99999955] --> [1]
[0 0 1 0], [0.00077679] --> [0]
[0 0 0 1], [0.00077679] --> [0]
[0 0 0 0], [0.96422137] --> [1]
.
----------------------------------------------------------------------
Ran 1 test in 0.034s

OK
```