AUNNL Documentation

Version 1.0 beta
Documentation Edition 1.0.0

Github: https://github.com/botperson/aunnl/

Part 1: An Introduction

This was simply an attempt to write neural networks from scratch and nothing more. When the network was required to do multiple different tasks, it evolved into a library, which is AUNNL. While AUNNL is incomparable to proper, full-fledged deep learning and machine learning libraries, it does do one thing. Writing code with AUNNL is very easy, simply because you don't have to do much. This is useful for people who just want to make small, simple neural networks to play with for some time. By no means is it actually meant to be a proper library to be used for more than a little while. It's just a very primitive toy that is easy to use and intuitive (hopefully).

With that being said, it doesn't take much time to understand what AUNNL does and how it works. Continue on through the documentation to see how to use AUNNL.

I'd suggest you also check out the example code, which is available on the AUNNL Github repository: https://github.com/botperson/aunnl/

If you'd like to see the inefficient jungle that is hidden when AUNNL is used, check out the code of aunnl.py. I've done my best to comment it to make it easy to understand. It will also allow me to remember what my code is doing if I ever need to use AUNNL again.

Part 2: Functions

This documentation is designed to be simple. From here, you essentially need to know *how a neural network can feed forward information* and what *weights* and *biases* are. I also expect you to have some basic knowledge of python syntax - just functions, arrays, and variables.

<u>It is highly recommended that you read these docs while you play with the simple examples in the examples folder to understand what these functions do.</u>

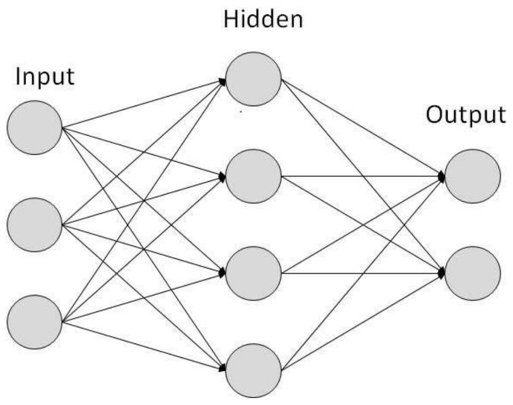
aunnl.createNetwork()

`aunnl.createNetwork(hl, i, d)`

The `createNetwork()` function creates a local instance of the neural network, i.e, it stores it in your RAM. It sets all the weights and biases randomly.
This function has 3 parameters:

- *hl* - The number of hidden layers in your neural network.
- *i* - The number of inputs of your neural network.
- *d* - An array of integers specifying the number of neurons in the hidden layers and output layers. The values in the array must be put in the order in which the layers would be fed information in a feed-forward case.

Example 1: `aunnl.createNetwork(1, 3, [4, 2])` creates this network:



The ‘1’ indicates that the network has 1 hidden layer, the ‘3’ indicates that the network has 3 inputs, or 3 input neurons, the [4, 2] represents the 4 neurons in the hidden layer and the 2 neurons in the output layer.

Example 2: `aunnl.createNetwork(2, 4, [8, 7, 3])` will create a neural network with four input neurons, two hidden layers with 8 neurons and 7 neurons for the first and second layers respectively, and three output neurons.

aunnl.feedForward()

`aunnl.feedforward(`*`input)`*

This function feeds an input forwards through the neural network and produces an output. The only parameter is *input*, which is an array that has the values for the inputs for the network.
Example: `aunnl.feedForward([0, 0, 1, 0])` feeds forward the values 0, 0, 1, 0 through the network and produce an output.
The output of `aunnl.feedForward()` is stored the array `output`, which can be used in your program as `aunnl.output`. The output array contains the integer values of each of the outputs or output neurons.

aunnl.backPropagate()

`aunnl.backPropagate(`*`labels`*`,` *`yc`*`,` *`loop`*`,` *`data`*`,` *`start`*`,` *`boo`*`)`

This function calculates how much the weights and biases should change. When we create the network with `aunnl.createNetwork()`, the weights and biases are assigned randomly. If we feed forward an input, the network will give us a garbage output - a result of those random weights and biases. The weights and biases are what generate the result, and if we want the neural network to do something, then we need to correct the weights and biases. Gradients tell us how to correct these. To do this, we need to find out how much the weights and biases need to change. The procedure for finding how much the weights and biases have to change, i.e., the gradients, is called backpropagation.

Backpropagation requires a set of training data and training labels to learn from.
(AUNNL currently only supports supervised learning)

The AUNNL backprop function requires that you provide an array of your expected outputs (eg: `[0, 1]`) for backprop. For this, you must first perform forward propagation to see how bad the network performs with an input that is associated with the output array you use in the backprop function.

To make it easy to use multiple training examples and to make code shorter and simpler, the second parameter is a boolean which is set by default to false. When this is set to true, everything becomes simpler. Unlike the case in the above paragraph, if *yc* is set to true, you do not have to manually call the `feedForward()` function - the `backPropagate()` function will call if automatically. If *yc* is set to true, an average ne

If *yc* is set to true, more parameters have to be specified. These are:

- *loop* - The number of times you want to cycle through training examples, which is usually the number of training data and examples you want to use
- *data* - This is an array that stores the training data
- *start* - This is an integer that tells AUNNL on which training example backprop should start. By default, this is set to 0, and should be set to 0 unless you plan to use stochastic gradient decent.

If *yc* is set to true, the *data* and *input* parameters, which are arrays, should be structured as:
`data[training example number]` = training data array (like inputs `[0, 1, 1, 0]` for example)
`labels[training example number]` = training labels array (like outputs `[0, 1]` for example)

In an actual application they could look like this:

```
data = [ [0, 1, 1, 0],
       [1, 1, 0, 0],
       [0, 1, 0, 1] ]
labels = [ [0, 1],
          [1, 0],
          [0, 1] ]
```

The above example is for training with three examples. It indicates that the input `[0, 1, 1, 0]` should produce the output of `[0,1]`, the input `[1, 1, 0, 0]` should produce the output `[1, 0]`, and so on.

If *yc* is set to false, then you needn't use a nested array for `data` and `labels`, as you are not using multiple training examples to calculate your gradients (which is not reccomended, and will incentivize the network to be biased towards the output of your given training example, and not change). In this case, if your input is `[1, 0, 0, 0]` and output is `[0, 1]`, data and labels should be in this format:

```
output = [1, 0, 0, 0]
labels = [0, 1]
```

As you can see, you needn't use a nested array if *yc* is set to false because you will only use one training example.

The final or sixth parameter is *boo*, which is a boolean. By default it is set to false. The *boo* parameter, when set to true, will automatically convert a boolean `labels` array to an integers `labels` array to be used by the network. If `labels` is not a boolean array in your case, then this should either be left alone (it is set to false by default) or should be marked as true.

aunnl.decendGradient()

`aunnl.decendGradient(`*`lr`*`)`

This function changes the weights and biases based on the (average) gradient calculated by the `aunnl.backPropagate()` function. The only parameter for this function is *lr*, or leaning rate. This is a hyperparameter and should be changed with trial and error to find an optimal value that makes the network most accurate. It internally uses the `learnr8()`function, so you do not have to manually call it if you use the `decendGradient()` function.
This function becomes redundant if you use `aunnl.simpleGradientDecent()`

aunnl.cleanGradients()

This function simply resets all the gradients stored in the `gradient` array to zero to be used for the next backprop routine.
This function becomes redundant if you use `aunnl.simpleGradientDecent()`

aunnl.learnr8()

`aunnl.learnr8(`*`lr`*`)`

This function is used to make the gradients change because of the learning rate. Internally, it is simply multiplying all the gradients with the learning rate, and these new gradinet values will be used in the other steps of gradient decent.
This function becomes redundant if you use `aunnl.decendGradient()` or `aunnl.simpleGradientDecent()`

aunnl.simpleGradientDecent()

`aunnl.simpleGradientDecent(`*`lr`*`, `*`c`*`)`

This function eliminates the need to manually call the `aunnl.decendGradient()`, `aunnl.cleanGradients()` and `aunnl.learnr8()` functions, as it internally calls these automatically. It primarily makes it significantly easier to perform gradient decent when you have used multiple training examples with `aunnl.backPropagate()` by setting its *yc* parameter to true. It allows you to take a more accurate average gradient decent step with the influence of a diverse set of training examples.

The aunnl.simpleGradientDecent() function has two parameters. The *lr* parameter is the learning rate. This is a hyperparameter and should be changed with trial and error to find an optimal value that makes the network most accurate. The *c* parameter is an integer which averages the gradient. In almost all cases, *c* should be the number of training examples you use with `aunnl.backPropagate()`. If you've used 100 training examples with `aunnl.backPropagate()`, *c* should be equivalent to 100.

aunnl.printProgress()

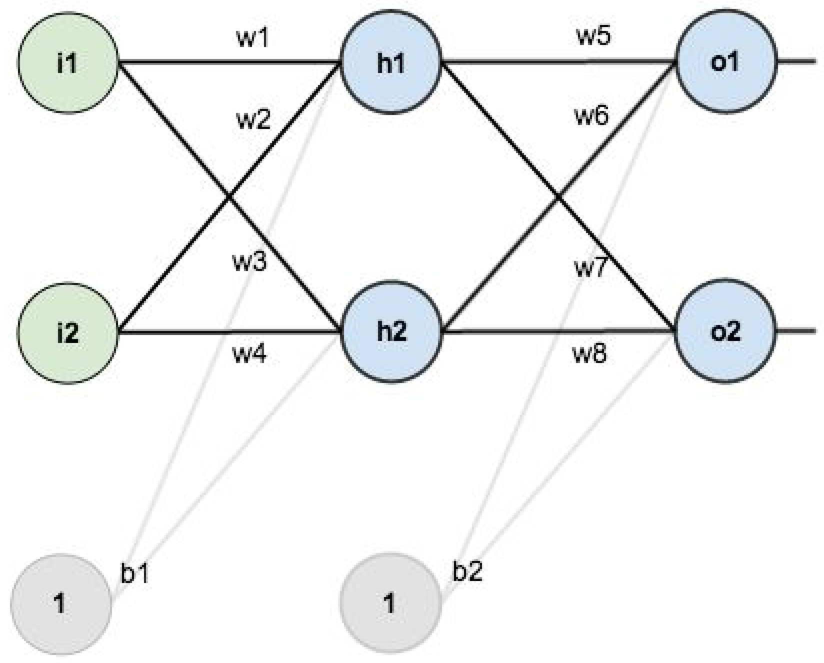
`aunnl.printProgress(`*`c`*`, `*`f`*`)`

This function is purely aesthetic, and just prints a progress bar. When you're training a large network with a ton of training examples, it comes in handy. It has two parameters. The *c* parameter is an integer specifying the magnitude of completion. The *f* parameter specifies the final magnitude of the completed task. If 1% of a task is completed, `printProgress(`*`1`*`, `*`100`*`)` will print a progress bar with 1/100 completion, i.e., 1% completion.

Part 3: Internal Network Structuring

This section is only meant for people who would like to see what AUNNL is doing internally. If you only want to use AUNNL, then you should skip this section entirely. If you are trying to understand aunnl.py, then you should use the comments in the code as your guide and this as additional information that you'll need. This section will be referencing the code often.

Let's take this simple neural network as an example:



In the program, weights are associated with the layer they feed into. For example, w1 and w2 are associated with h1, not i1 & i2. Similarily, w6, w6 are associated with o1, and so on for all the neurons.

This is important, as weights are stored in a nested array, and so are their gradients. If we wish to know the value of w5, it is stored in `weight[1][0][0]`. Let me explain this better. The first array index specifies the layer. The weight w5 is associated with the neuron o1, which is in the output layer. The output layer is the second layer excluding the input layer. As indexing starts at 0, the first array index is 1. So, our weight is stored somewhere in `weight[1]`. The neuron o1 is the first neuron in the output layer. As indexing starts at zero, its index is 0. Thus w5 should be stored in `weight[1][0]`. The neuron o1 is connected to the hidden layer via w5 and w6. As w5 is the connection to the first neuron in the hidden/previous layer, it is the first connection to o1. As indexing starts at 0, its index is 0. Thus, the integer value of the weight w5 is stored in `weight[1][0][0]`.

The values of biases require only two indicies, as they are not connected to anything, they are only associated with a neuron. The image uses a single bias for an entire layer, but AUNNL internally uses a bias for each neuron. To get the bias, we simply need to get the indicies for the layer and neuron. By following the same format for the `weight` array, the integer value for bias for the o2 neuron is `bias[1][0]`.

To get the integer values of gradients, simply follow the above formats for weights an biases, but with an extra index at the start. The first index specifies if the gradient is for a weight or for a bias. A 0 indicates a weight and a 1 indicates a bias. Thus, the integer gradient value for the neuron w5 is `gradient[0][1][0][0]` and the integer gradient value for the bias of the o2 neuron is `gradient[1][1][0]`. As you can see, the indicies after the first index are the same as the ones that were used to find the weight and bias values. Only the first index is added.

A similar format is followed for other variables and arrays in aunnl.py.