Lab 8

From 6.034 Wiki

Contents

- Code available on following links:
- Use Git on Athena: `git clone /mit/6.034/www/labs/lab6`
- Download it as a ZIP file: http://web.mit.edu/6.034/www/labs/lab6/lab6.zip
- View the files individually: http://web.mit.edu/6.034/www/labs/lab6/

All of your answers belong in the main file `lab6.py`.

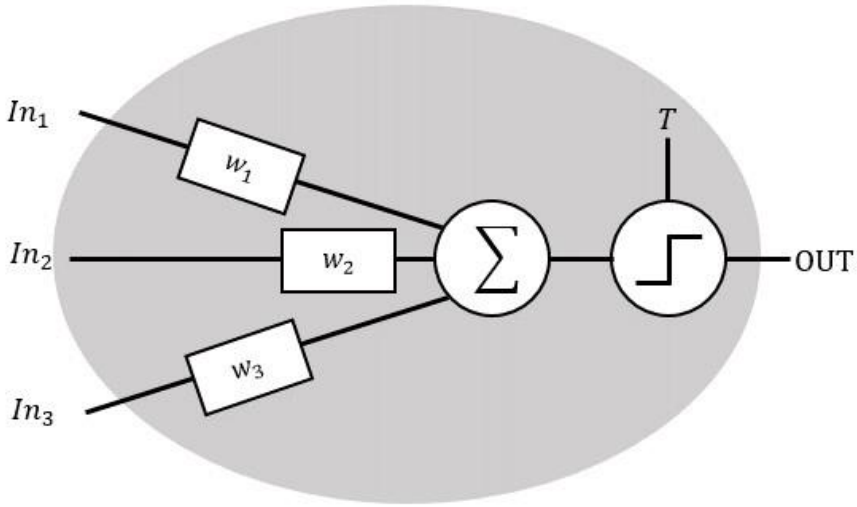
Neural Networks are Biologically Inspired

An artificial neural network (ANN), commonly referred to as a *neural net*, is a biologically-inspired machine-learning classification model. It is a network of atomic *neurons*, each of which takes in one or many inputs, computes a value based on them, and outputs another value that may be consumed by more neurons downstream.

This design attempts to mimic our understanding of the human brain, which comprises nerve cells (neurons) connected to each other in massive networks via axons and dendrites. Our current understanding of the brain is that a neuron *activates* when sufficient electric impulses reach the neuron from neighboring, connected neurons; upon activation, the neuron sends an electric impulse down its axon to be received by other neurons.

The Anatomy of an Artificial Neural Network

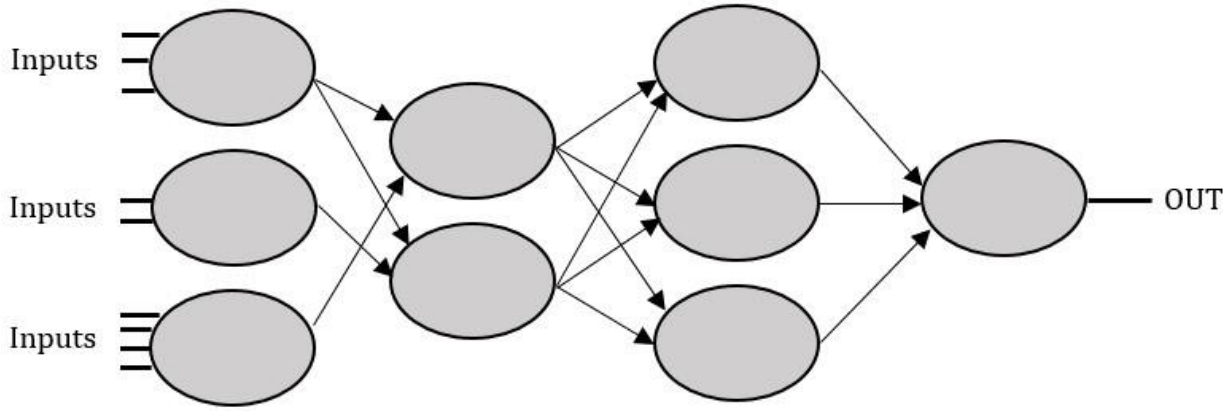
A very simple artificial neural network (with just one neuron) might look like this:



Each input (In_i) is multiplied by a constant *weight* (w_i); these products are then summed together ($In_1*w_1 + ... + In_n*w_n$)and put through a *threshold function* (T) which evaluates the result (which typically ranges from 0 to 1) based on the value of the input.

Indeed, every connection between a neuron and another neuron (or between an input and a neuron) is called a *wire*, and each wire has a weight associated with it. In practice, artificial neural networks can comprise hundreds or thousands of neurons, with possibly millions of wires and weights! However, in this class, we typically stick to much smaller neural nets.

Here is an example of a more complicated neural net, which is formed by chaining multiple neurons together:



Part 1: Wiring a Neural Net

Consider a neural net with two inputs *x* and *y*, both connecting to a neuron **P** with weights *a* and *b* respectively. This input-layer neuron **P** draws a line in a 2-dimensional space and shades one side of it, satisfying the inequality `a x + b y >= T` for some constant threshold *T*.

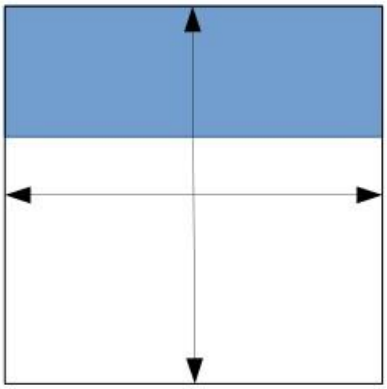
In fact, *every* input-layer neuron that takes in two inputs *x* and *y* will draw a line and shade one side of it in 2-space.

The remaining neurons in the later layers of the neural net perform logic functions on the shadings.

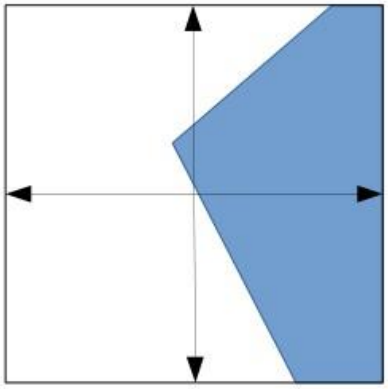
Each of the following pictures can be produced by a neural net with two inputs *x* and *y*. For each one, you need to determine the minimum number of neurons necessary to produce the picture. Express your answer as a list indicating the number of nodes per layer. Put your answers in `lab6.py` as assignments to the variables `nn_half, ...`.

As an example, the neural net shown in the previous section would be represented by the list `[3, 2, 3, 1]`.

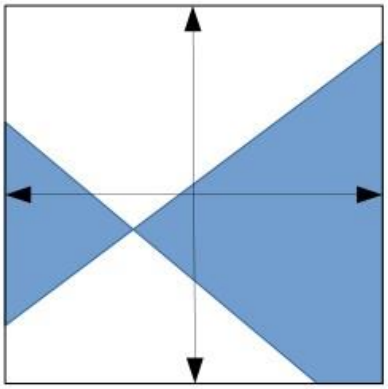
`nn_half = []`
shaded half-plane above a horizontal line



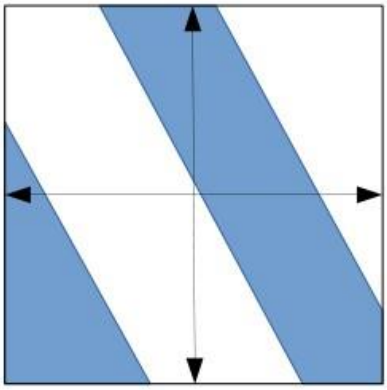
`nn_angle = []`
“pie slice” of the plane, defined by two lines



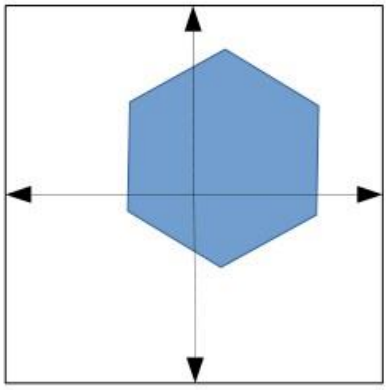
`nn_cross = []`



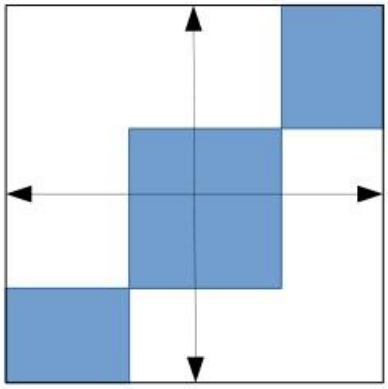
`nn_stripe = []`



`nn_hexagon = []`



`nn_grid = []`
three sections of a 3x3 grid



If you are still confused about some of the solutions, you're welcome to view the detailed explanations here (http://web.mit.edu/6.034/www/neural-net-visualization.pdf) .

Part 2: Coding Warmup

Exploring different threshold functions

First, you'll code some threshold functions for the neural nets. The `stairstep`, `sigmoid`, and `ReLU` functions are threshold functions; each neuron in a neural net uses a threshold function to determine whether its input stimulation is large enough for it to emit a non-zero output. The mathematical definitions of these three threshold functions are as follows:

$$\begin{aligned} \text{stairstep}_T(x) &= \begin{cases} 1, \text{ if } x \geq T \\ 0, \text{ if } x < T \end{cases} \\ \text{sigmoid}_{S,M}(x) &= \frac{1}{1 + e^{-S(x-M)}} \\ \text{ReLU}(x) &= \max(0, x) \end{aligned}$$

Please implement each of the functions below. `stairstep`: Computes the output of the

stairstep function using the given threshold (T).

```
def stairstep(x, threshold=0):
```

`sigmoid`: Computes the output of the sigmoid function using the given steepness (S) and midpoint (M). For your convenience, the constant *e* is defined in `lab6.py`.

```
def sigmoid(x, steepness=1, midpoint=0):
```

`ReLU`: Computes the output of the ReLU (rectified linear unit) function.

```
def ReLU(x):
```

Measuring performance with the accuracy function

The accuracy function is used when training the neural net with back propagation. It measures the performance of the neural net as a function of its desired output and its actual output (given some set of inputs). Note that the accuracy function is symmetric -- that is, it doesn't care which argument is the desired output and which is the actual output.

We define our accuracy function, along with its derivative:

$$\begin{aligned} \text{Performance} = \text{Accuracy}(out^*, out) &= -\frac{1}{2}(out^* - out)^2 \\ \frac{\text{dAccuracy}}{\text{d}out} &= out^* - out \end{aligned}$$

where *out* is the output of the neural net, and out* is the *desired* output of the neural net.

Implement `accuracy(desired_output, actual_output)`, which computes accuracy using `desired_output` and `actual_output`. Fun fact: If the neurons in the network are using the `stairstep` threshold function, the accuracy can only be `-0.5` or `0`.

```
def accuracy(desired_output, actual_output):
```

Part 3: Forward propagation

Important Notice

Before starting this section, please read the entirety of the API section so that you know what functions and features are available to you!

Next, you'll code forward propagation, which takes in a dictionary of inputs and computes the output of every neuron in a neural net.

As part of coding forward propagation, you should understand how a single neuron computes its output as a function of its input: each input into the neuron is multiplied by the weight on the wire, the weighted inputs are summed together, and the sum is passed through a specified threshold function to produce the output.

You should iterate over each neuron in the network in order, starting from the input neurons and working toward the output neuron. (Hint: The function `topological_sort()` may be useful.) The algorithm is called forward propagation because the outputs you calculate for earlier neurons will be propagated forward through the network and used to calculate outputs for later neurons.

To assist you, we've provided a helper function `node_value`, which takes in a node (which can either be an **input** or a **neuron**), a dictionary mapping input names (e.g. 'x') to their values, and a dictionary mapping neuron names to their outputs, and returns the output value of the node.

For example:

```
>>> input_values = {'x': 3, 'y': 7}
>>> neuron_outputs = {'Neuron1': 0}
>>> node_value('Neuron1', input_values, neuron_outputs)
0
>>> node_value('y', input_values, neuron_outputs)
7
>>> node_value(-1, input_values, neuron_outputs)
-1
>>>
```

Note that `node_value` does not *compute* the outputs of any nodes. It simply looks at the input node and returns the value for that node if it's a constant or if it's an entry in one of the two input dictionaries. Depending on your implementation, you may or may not want to use this function.

Implement the method `forward_prop`:

```
def forward_prop(net, input_values, threshold_fn=stairstep):
```

Here, `net` is a neural network, `input_values` is a dictionary mapping input variables to their values, and `threshold_fn` is a function* that each neuron will use to decide what value to output. This function should return a tuple containing two elements:

1. The overall output value of the network, i.e. the output value associated with the output neuron of the network.
2. A dictionary mapping neurons to their immediate outputs.

The dictionary of outputs is permitted to contain extra keys (for example, the input values). The function should *not* modify the neural net in any way.

* The `threshold_fn` argument will be one of the threshold functions you implemented at the beginning of the lab. The astute reader will recognize that each of the three threshold functions take a different number of (and different types of) arguments, so any algorithm leveraging this `threshold_fn` input can't technically be implementation-agnostic with respect to the threshold function type. However, for this lab, we have decided to simplify your lives by only requiring that you call `threshold_fn` with the first argument, `x`.

Part 4: Backward propagation

Backward propagation is the process of training a neural network using a particular training point to modify the weights of the network, with the goal of improving the network's performance. In the big picture, the goal is to perform gradient ascent on an n-dimensional surface defined by the n weights in the neural net. We compute the update for some weight w_i using the partial derivative of the accuracy with respect to w_i.

In lecture, we derived several useful equations for computing weight updates during back-propagation. As a mathematical shortcut, we introduced the variable δ that lets us reuse computation during back-propagation, instead of having to repeatedly take derivatives.

Below,

- δ_B indicates the delta calculation for neuron B.
- $W_{A \to B}$ represents the weight of the wire between neurons A and B.
- r is the learning rate or step size: it's an indication of how quickly you allow your weights to change.

$$
\begin{aligned}
w_{A \to B,\ \text{new}} &= w_{A \to B,\ \text{old}} + \Delta w_{A \to B} \\
\Delta w_{A \to B} &= r \cdot out_A \cdot \delta_B \\
\delta_B &= \begin{cases} out_B \cdot (1 - out_B) \cdot (out^* - out_B), \text{ if neuron } B \text{ is in final layer} \\ out_B \cdot (1 - out_B) \cdot \sum\limits_{\text{outgoing } C_i} w_{B \to C_i}\, \delta_{C_i}, \text{ if neuron } B \text{ is not in final layer} \end{cases}
\end{aligned}
$$

When implementing code in this section, don't forget about the functions you've written above! They may be helpful.

Gradient ascent

Conceptually, the idea of *the network's performance* is abstracted away inside a huge and complex accuracy function, and gradient ascent (or descent, depending on your point of view) is simply a form of hill-climbing used to find the best output possible from the function.

When training a neural net, gradient ascent is usually performed after forward propagation. Forward propagation produces some output from the network. Gradient ascent then looks at this output and decides how the parameters of the neural net should change in order to get an output from the net that is closer to the desired output. After gradient ascent is performed and the parameters of the neural net are updated accordingly, forward propagation can be run again to get the new output from the neural net, which should be closer to the desired output. You can run this process repeatedly until the output of the net matches (or is close to) the desired output.

To get a feel for this concept, we will first ask you to implement a very simplified gradient ascent algorithm, to perform a single step of (pseudo-)gradient ascent. `gradient_ascent_step` takes in three arguments:

1. `func`: a function *of three arguments*
2. `inputs`: a list of three values representing the three current numerical inputs into `func`
3. `step_size`: how much to perturb each variable

This function should perturb each of the inputs by either `+step_size`, `-step_size`, or `0`, in every combination possible (a total of $3^3 = 27$ combinations), and evaluate the function with each possible set of inputs. Find the assignments that *maximize* the output of `func`, then return a tuple containing

1. the function output at the highest point found, and
2. the list of variable assignments (input values) that yielded the highest function output.

```
def gradient_ascent_step(func, inputs, step_size):
```

For example, if the highest point is `func(3, 9, 4) = 92`, you would return `(92, [3, 9, 4])`.

Note: We have defined the constant `INF` in `lab6.py` if you need to represent infinity.

Back prop dependencies

Recall from class (or from the equations above) that in back propagation, calculating a particular weight's update coefficient has dependencies on certain neuron outputs, inputs, and other weights. In particular, updating the weight between nodes A and B requires the output from node A, the current weight on the wire from A to B, the output of node B, and all neurons and weights downstream to the final layer.

Implement a function that takes in a neural net and a `Wire` object, then returns a `set` containing all `Wire`s, inputs, and neurons that are necessary to compute the update coefficient for `wire`'s weight. You may assume that the output of each neuron has already been calculated via forward propagation.

```
def get_back_prop_dependencies(net, wire)
```

Hint: One way to do this is to maintain a `set` of already-recorded dependencies and build it up as you move through the network. Start by adding the dependencies of the current `wire` and move forward, recursively (or iteratively) adding dependencies of each wire down the network.

If you're still not sure how to approach this function, you can skip ahead to the next section, and/or look at an example in 2015 Quiz 3, Problem 1, Part B (http://courses.csail.mit.edu/6.034f/Examinations/2015q3.pdf) .

Basic back propagation

Now let's go over the basic back-propagation algorithm. To perform back propagation on a given training point, or set of inputs:

1. Use forward propagation with the *sigmoid* threshold function to compute the output of each neuron in the network.
2. Compute the update coefficient `delta_B` for each neuron in the network, starting from the output neuron and working backward toward the input neurons. (Note that you may not need to use `get_back_prop_dependencies`, depending on your implementation.)
3. Use the update coefficients `delta_B` to compute new weights for the network.
4. Update all of the weights in the network.

Computing δB

You have already implemented the `forward_propagation` routine. To complete the definition of back propagation, you'll define a helper function `calculate_deltas` for computing the update coefficients `delta_B` of each neuron in the network, and a function `update_weights` that retrieves the list of update coefficients using `calculate_deltas`, then modifies the weights of the network accordingly.

Implement `calculate_deltas` to return a dictionary mapping neurons to update coefficients (`delta_B` values):

```
def calculate_deltas(net, desired_output, neuron_outputs):
```

Note that this function takes in `neuron_outputs`, a dictionary mapping neurons to the outputs yielded in one iteration of forward propagation; this is the same dictionary that is returned from `forward_prop`.

Updating weights

Next, use `calculate_deltas` to implement `update_weights`, which performs a single step of back propagation. The function should compute `delta_B` values and weight updates for the entire neural net, then update all weights. The function `update_weights` should return the modified neural net with appropriately updated weights.

```
def update_weights(net, input_values, desired_output, neuron_outputs, r=1):
```

Putting it all together

Now you're ready to complete the `back_prop` function, which repeatedly updates weights in the neural net until the accuracy surpasses the accuracy threshold. (Recall that accuracy is always non-positive, and that an accuracy closer to 0 is better.) `back_prop` should return a tuple containing:

1. The modified neural net, with trained weights; and
2. The number of iterations (that is, the number of times you batch-updated the weights)

```
def back_prop(net, input_values, desired_output, r=1, minimum_accuracy=-0.001):
```

Once you finish, you're all done writing code in this lab!

Part 5: Training a Neural Net

In practice, we would want to use multiple training points to train a neural net, not just one. There are many possible implementations -- for instance, you could put all the training points into a queue and perform back propagation with each point in turn. Alternatively, you could use a multidimensional accuracy function and try to train with multiple training points simultaneously.

In this part of the lab, we'll attempt to train various neural nets using the code you've written so far. We have supplied you with a file called `training.py`, which should take care of all of the overhead for you.

Example datasets

Here are six example datasets that you could use to train a 2-input neural net, all of which are defined in `training.py`. Each training dataset has up to 25 data points, each of which is associated with one of two possible classifications ("+" or "-").

In the 2D graphs below, each axis represents an input value, and a `+` or `-` represents that training point's classification. If a space is instead blank, that indicates that there is no training point associated with that particular combination of input values. 1. A horizontally divided space ("horizontal")

```
4 - - - - -
3 - - - - -
2 - - - - -
1 + + + + +
0 + + + + +
  0 1 2 3 4
```

2. A diagonally divided space ("diagonal")

```
4 + + + + -
3 + + + - -
2 + + - - -
1 + - - - -

0 - - - - -
  0 1 2 3 4
```

3. A diagonal stripe ("stripe")

```
4 - - - - +
3 - - - + -
2 - - + - -
1 - + - - -
0 + - - - -
  0 1 2 3 4
```

4. This patchy checkerboard shape ("checkerboard")

```
4 - -   + +
3 - -   + +
2
1 + +   - -
0 + +   - -
  0 1 2 3 4
```

5. The letter L ("letterL")

```
4 + -
3 + -
2 + -
1 + - - - -
0 - + + + +
  0 1 2 3 4
```

6. This moat-like shape ("moat")

```
4 - - - - -
3 -       -
2 -   +   -
1 -       -
0 - - - - -
  0 1 2 3 4
```

With the correct wiring, it's possible to train a neural net with 6 or fewer neurons to classify any one of the shapes.

What `training.py` does

In `training.py`, we've provided code to train a neural net, written by past 6.034 students Joel Gustafson and Kenny Friedman. This code imports functions that you wrote in `lab6.py` and generalizes the functions to consider multiple training points, instead of just one. `training.py` has several features:

- Provides the six encoded training data sets illustrated above
- Provides three fully connected neural nets architectures:
  - `get_small_nn()` returns a small [2, 1] neural net (ideal for drawing two lines and combining them with a simple logic function such as AND or OR)
  - `get_medium_nn()` returns a medium [3, 2, 1] neural net (ideal for drawing three lines and combining them with a more complex logic function such as XOR)
  - `get_large_nn()` returns a large [10, 10, 5, 1] neural net
- Generalizes forward and backward propagation to train on arbitrary datasets of multiple training points by considering multiple data points in parallel.
- Uses NumPy and Matplotlib (Python libraries) to display the neural net's output in real time as a heatmap, with a spectrum of colors representing sigmoid output values from 0 (blue) to 1 (yellow).

Note that for each training attempt, `training.py` randomly initializes network weights.

How to run `training.py`

To run the code, you'll need the Python packages Matplotlib and NumPy. If you don't have them, see below. Once you have the packages, you can use either a terminal command line or an interactive Python prompt.

On the command line

Run `python3 training.py` with up to three optional arguments: **`-net [small|medium|large]`**: Selects which neural net

- configuration to train, as described above. *Default*: `medium`.
- **`-resolution`** ***`POSITIVE_INT`***: Sets the resolution of the dynamic heatmap -- a resolution of `1` will display a 5x5 grid, and a resolution of `10` will display a 50x50 grid on a 1:10 scale. Be aware that increasing the resolution exponentially increases the time each simulation iteration takes, so resolutions of over `10` are not recommended. *Default*: `1`.
- **`-data [diagonal|horizontal|stripe|checkerboard|letterL|moat]`**: Selects the training dataset from the six shown above. *Default*: `diagonal`.

For example, to train the large [3, 2, 1] neural net on the checkerboard dataset, and display the progress with a resolution of 10 pixels per integer coordinate, run:

```
python3 training.py -data checkerboard -net large -resolution 10
```

from the command line. Any of the parameters that are omitted will be replaced with their default values. For example,

```
python3 training.py
```

will train the default medium net on the diagonal dataset with resolution 1.

At a Python prompt (e.g. IDLE)

Run the file `training.py`, then call the function start_training() with up to three optional arguments:

- **`net = ['small'|'medium'|'large']`**: Selects which neural net configuration to train, as described above. *Default*: `'medium'`. **`resolution =`** ***`POSITIVE_INT`***: Sets the resolution of the dynamic heatmap -- a resolution of `1` will display a 5x5 grid, and a resolution of `10` will display a 50x50 grid on a 1:10 scale. Be aware that increasing the resolution exponentially increases the time each simulation iteration takes, so resolutions of over `10` are not recommended. *Default*: `1`.
- **`data = ['diagonal'|'horizontal'|'stripe'|'checkerboard'|'letterL'|'moat']`**: Selects the training dataset from the six shown above. *Default*: `'diagonal'`.

For example, to train the large [3, 2, 1] neural net on the checkerboard dataset, and display the progress with a resolution of 10 pixels per integer coordinate, call:

```
start_training(data='checkerboard', net='large', resolution=10)
```

from the Python prompt. Any of the parameters that are omitted will be replaced with their default values. For example,

```
start_training()
```

will train the default large net on the diagonal dataset with resolution 1.

What if I don't have Matplotlib and NumPy?

If you don't have Matplotlib and NumPy installed, you have a few options. You can do one of the following:

- install them (just Google them), e.g. with `pip` or by downloading them manually.
- create a new virtual environment (using the `virtualenv` utility) with Python 3, Matplotlib, NumPy
  1. Go to the directory in which you want to place your virtual environment and call `virtualenv -p python3 myVirtualEnvName`
  2. Activate that Python environment by calling `source myVirtualEnvName/bin/activate`. Now whenever you call `python3`, it refers to the Python 3 binary installed inside the `myVirtualEnvName` directory. (Try calling `which python3` to see.) Note that only the current window/tab of your terminal is using this virtual environment.
  3. Install NumPy into your virtual environment by calling `pip3 install numpy`
  4. Install Matplotlib into your virtual environment by calling `pip3 install matplotlib`
  5. Navigate to your lab directory. You can now run `training.py`!
  6. To deactivate the virtual environment, call `deactivate`.

- install a stand-alone Python distribution that comes with the packages and won't interfere with your current Python installation (e.g. Anaconda or Python(x,y))
- work with a friend, running the code on their computer

If none of these options work for you, you can try to execute your code on Athena using *Python 2* (not Python 3). This *should* work, assuming your lab 6 code has no Python-3-exclusive syntax in it, but we can't guarantee it:

- use an Athena cluster computer (the Athena version of Python 2 already includes Matplotlib and NumPy)
- use Athena locally via `ssh -X` (which enables Athena to display GUI windows, including colored plots, on your screen). Note that this option is *very* slow, and should only be a last resort: `ssh -X username@athena.dialup.mit.edu`

If you are still having issues and none of the above solutions are working, please come to office hours so we can try to help in person!

Your task: Multiple-choice questions based on `training.py`

Questions 1-5: How many iterations?

For each of the combinations of neural nets and datasets below, try training the neural net on the dataset **a few times**. Then, fill in the appropriate `ANSWER_n` with an int to answer the question: **When the neural net didn't get stuck, how many iterations did it generally take to train?**

(The tester will check that your answer is in the right range of numbers, based on the repeated trials that we ran. Note that all the answers are expected to be under 200; if the neural net seems to be stuck or is taking more than 200 steps to train, feel free to abort by typing Ctrl+C or the equivalent Keyboard Interrupt. Depending on your system, you may need to type Ctrl+C multiple times and/or manually close the heatmap window.)

You may do this with any resolution; the resolution shouldn't affect the number of steps required for back prop to converge. Higher resolution makes it easier to see how the neural net is dividing up the space, but it also greatly increases the real time required for training because it has to perform forward propagation to compute the color output for each cell.

Question 1: `small` neural net, `diagonal` dataset

Question 2: `medium` neural net, `diagonal` dataset

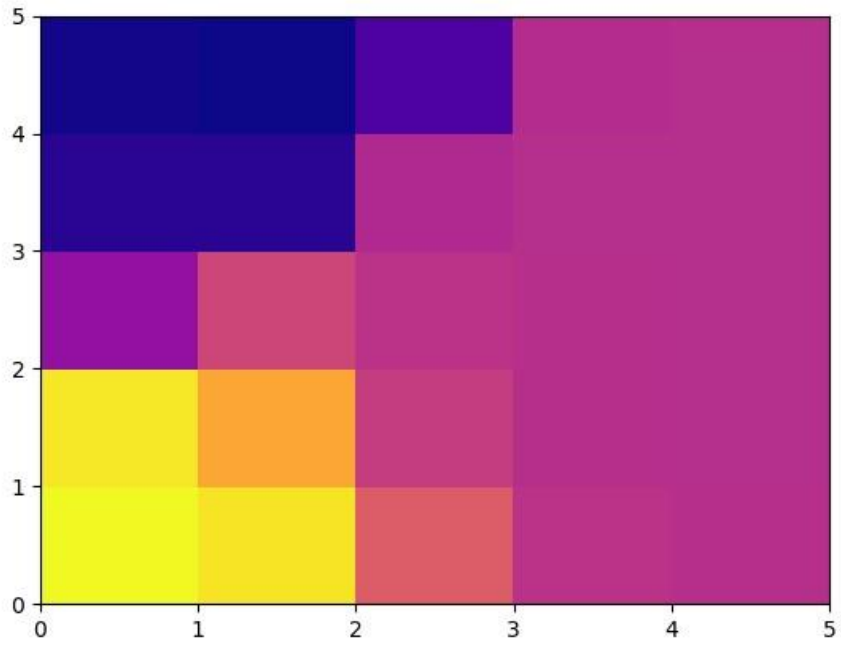
Question 3: `large` neural net, `diagonal` dataset

Question 4: `medium` neural net, `checkerboard` dataset

Question 5: `large` neural net, `checkerboard` dataset

Questions 6-9: Identifying parameters

Suppose that after training for 200 iterations, the neural net heatmap looks like this:



Question 6: What is the training resolution? (Fill in ANSWER_6 with an int.)

Question 7: Of the six datasets, which one is the neural net probably being trained on? (Fill in ANSWER_7 with a string.)
Question 8: Which neural net could be producing the heatmap? (Fill in ANSWER_8 with a list of one or more strings, choosing from 'small', 'medium', and 'large. For example: ['small', 'large'])

Question 9: What is likely the state of the simulation? (Fill in ANSWER_9 with a one-letter string representing the one best answer, e.g. 'A'.)

A. Training is complete, and the data is fully classified.
B. Training is stuck at a local maximum.
C. The neural net is overfitting to the data.
D. The neural net is classifying three classes of points.

Questions 10-12: Conceptual Questions

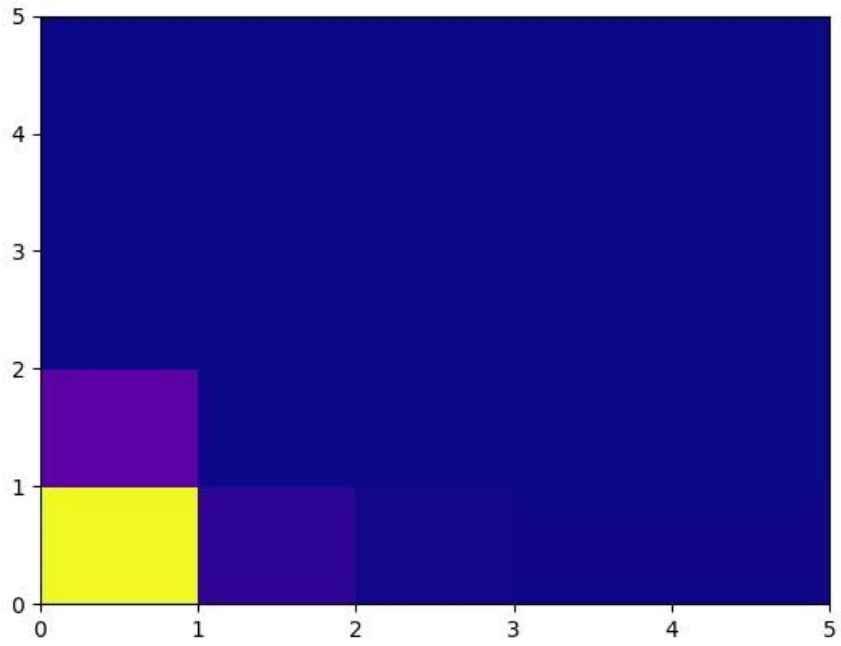
Question 10: Why does the `diagonal` dataset generally take less iterations to train than the `checkerboard` dataset? (Fill in ANSWER_10 with a letter representing the one best answer.)

A. Diagonal lines are easier for neural nets to draw than horizontal or vertical lines.
B. The neural nets tended to overfit to the checkerboard data more than to the diagonal data.
C. The neural nets tended to underfit to the checkerboard data more than to the diagonal data.
D. It requires fewer lines to separate the diagonal data than the checkerboard data.
E. The diagonal data is more constrained than the checkerboard data.

Question 11: The large neural net generally requires less iterations to train than the small or medium neural nets. What are some reasons why it might *not* be the best choice for these datasets? (Fill in ANSWER_11 with a list of one-letter strings, representing all answers that apply, e.g. ['A', 'B', 'C'].)

A. It might overfit the data because there are too many parameters.
B. It might underfit the data because it spends too few iterations training.
C. It takes more time to compute each iteration.
D. Because there are more parameters, it's more likely to get stuck on a local maximum.

Question 12: You may have noticed that the neural net often either converged on a solution quickly, or got stuck with a partial solution early on and had trouble escaping from the local maximum. For example, consider this common result when attempting to train on the `stripe` dataset:

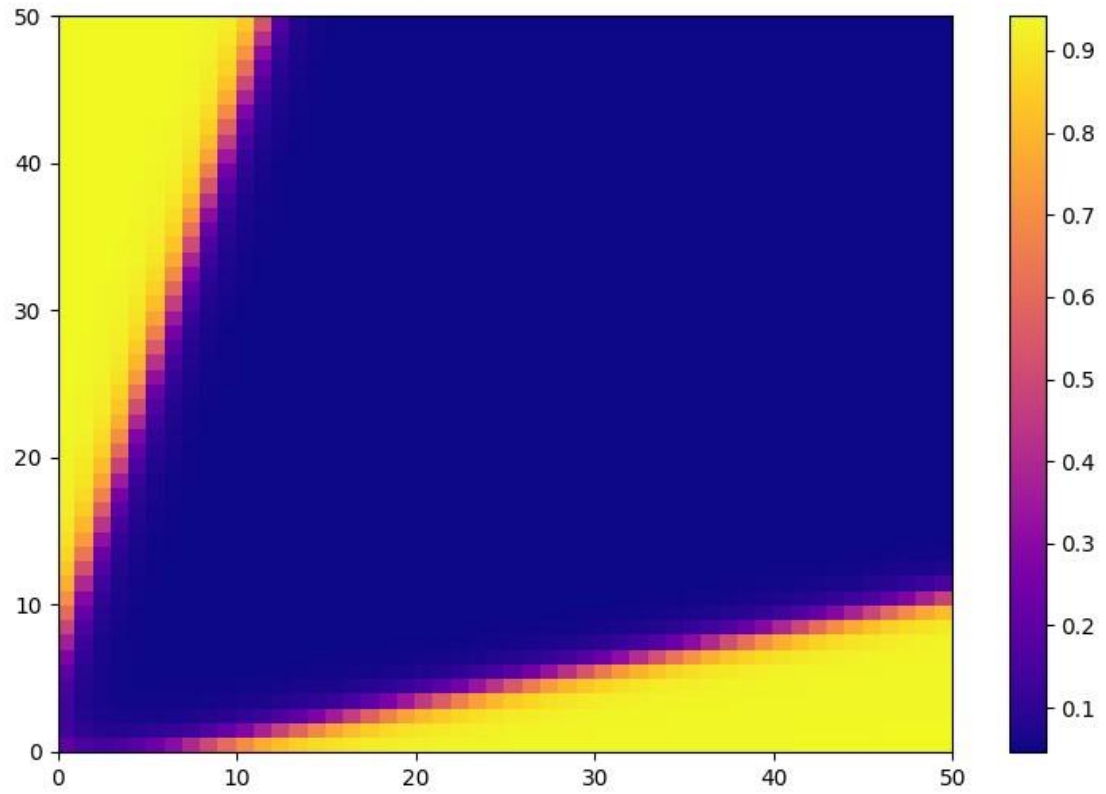


Suppose you're training an *any* neural net on an *any* dataset, and it just got stuck at a local maximum. Which of the following changes could reasonably help it to not get stuck the next time? (Fill in ANSWER_12 with a list of one-letter strings, representing all answers that apply, e.g. ['A', 'B', 'C'].)

A. Restart training with different randomly initialized weights.
B. Re-train multiple times using the same initial weights.
C. Use the current weights as the initial weights, but restart training.
D. Use less neurons.
E. Use more neurons.

If you want to do more...

You can continue experimenting with different combinations of neural nets, datasets, and resolutions. As inspiration, here's a really nice heatmap produced by training the `medium` neural net on the `letterL` dataset:



As a side note, the writers of training.py found that the medium neural net trained on `moat` often got stuck in local maxima, but occasionally slowly converged and terminated successfully after over 300 iterations and more than an hour. The medium neural net tended to convert relatively reliably and quickly with any of the other datasets.

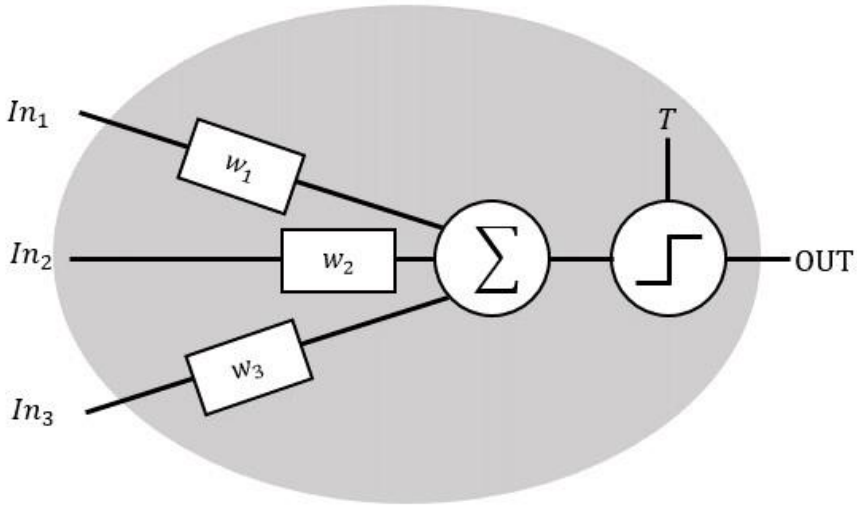
You're also welcome to add your own datasets or neural net architectures, following the examples found in training.py. If you do something cool, we'd love to see it! Feel free to send your code and/or results to 6.034-2018-staff@mit.edu (ideally with some sort of documentation). Your code could even end up in a future version of this lab! (With your permission, of course.)

Exploring the playground

If you'd like to play around with a neural net in your browser, you can do so at the TensorFlow playground (http://playground.tensorflow.org/) . This web-based neural net visualization offers more customization than we've provided in lab 6, including access to another threshold function (tanh), the ability to batch training points together, and the option to introduce noise into your data.

API

Remember, a very simple artificial neural network (with just one neuron) might look like this:



Each input (In_i) is multiplied by a constant *weight* (w_i); these products are then summed together ($In_1*w_1 + ... + In_n*w_n$)and put through a *threshold function* (T) which evaluates the result (which typically ranges from 0 to 1) based on the value of the input.

The file `neural_net_api.py` defines the `NeuralNet` and `Wire` classes, described below.

NeuralNet

A neural net is represented as a directed graph defined by a set of edges. In particular, the topology of the neural net is enforced by the edges, each of which defines the placement of the nodes and neurons.

In our case, each *edge* of a neural net is a `Wire` object. Each *node* of a neural net is either an *input* or a *neuron*.

- An **input** node represents a value that is fed into the input layer of the neural net (note the distinction between an *input* and an *input layer neuron*). An input is either represented by a string denoting its variable name (e.g. `"x"` represents the variable *x*), if the input is a variable input, *or* a raw number denoting its constant value (e.g. `2.5` represents the constant input 2.5), if the input is a constant input.
- A **neuron** node, conceptually, takes in values via wires and amalgamates them into an output. A neuron is represented as a string denoting its name, e.g. `"N1"` or `"AND-neuron"`. Note that these strings have no semantic meaning or association to the neuron's function or position in the neural net; the strings are only used as unique identifiers so that `Wire` objects (edges) know which neurons they are connected to.

As a consequence of how `Wire` objects store start and end nodes, *no variable **input** node may have the same name as a **neuron** node.*

A `NeuralNet` instance has two attributes: `inputs`

A list of named **input** nodes (including both constant inputs and variable inputs) to the network.

`neurons`
A list of named **neuron** nodes in the network.

In this lab, input values for `non-constant` inputs are supplied to neural nets in the form of a dictionary `input_values` that associates each named variable input with an input value.

You can retrieve particular nodes (neurons or inputs) in a network: `get_incoming_neighbors(node)`

Returns a list of the nodes which are connected as inputs to <tt>node.

`get_outgoing_neighbors(node)`
Returns a list of the nodes to which `node` sends its output.

`get_output_neuron()`
Returns the one output neuron of the network, which is the final neuron that computes a response. **In this lab, each neural net has exactly one output neuron.**

`topological_sort()`
Returns a sorted list of all the neurons in the network. The list is "topologically" sorted, which means that each neuron appears in the list after all the neurons that provide its inputs. Thus, the input layer neurons are first, the output neuron is last, etc.

You can also retrieve the various wires (edges) of a neural net: `get_wires(startNode=None, endNode=None)`

Returns a list of all the wires in the network. If `startNode` or `endNode` are provided, returns only wires that start/end at the particular nodes.

Finally, you can query specific parts of the network: `is_output_neuron(neuron)`

Returns `True` if the neuron is the final output neuron in the network, otherwise `False`.

`is_connected(startNode, endNode)`
Returns `True` if there is a wire from `startNode` to `endNode` in the network, otherwise `False`

Wire

A `Wire` is represented as a weighted, directed edge in a graph. A wire can connect an input to a neuron or a neuron to another neuron. A `Wire`'s attributes are: `startNode`

The input or neuron at which the wire starts. Recall that an input can be a string (e.g. `"x"`) or a number (e.g. `2.5`).

`endNode`
The the neuron at which the wire ends.

In addition, one can access and modify a `Wire`'s weight: `get_weight()`

Returns the weight of the wire.

`set_weight(new_weight)`
Sets the weight of the wire and returns the new weight.

Survey

Please answer these questions at the bottom of your lab file:

- `NAME`: What is your name? (string)
- `COLLABORATORS`: Other than 6.034 staff, whom did you work with on this lab? (string, or empty string if you worked alone)
- `HOW_MANY_HOURS_THIS_LAB_TOOK`: Approximately how many hours did you spend on this lab? (number or string)
- `WHAT_I_FOUND_INTERESTING`: Which parts of this lab, if any, did you find interesting? (string)
- `WHAT_I_FOUND_BORING`: Which parts of this lab, if any, did you find boring or tedious? (string)
- (optional) `SUGGESTIONS`: What specific changes would you recommend, if any, to improve this lab for future years? (string)

(We'd ask which parts you find confusing, but if you're confused you should really ask a TA.)

When you're done, run the online tester to submit your code.