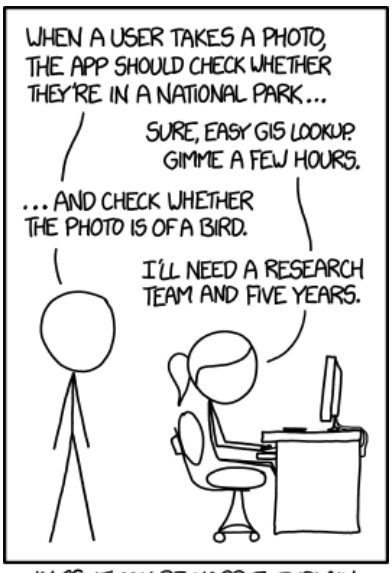
CS51A—Neural Network Lab



http://www.xkcd.com/1425/

In this lab, we will experiment with a few simple neural networks. The idea is to obtain preliminary answers to questions like:

- "How quickly can a network be trained?"
- "How reliable (or repeatable) is the training?"
- "How accurate is the resulting network?"

For the last half an hour of lab (or, when it seems like people have mostly wrapped up their experiments), each pair/group will give a short 1-2 minute, informal presentation on what they found. Although there is nothing to submit for this lab, you should keep notes and document your results for the presentation/discussion.

Intro to the neural net package

To start with, read this section and play with a few of the examples to familiarize yourself with the neural network package we'll be using.

- Create a PyCharm project called `nnlab` in the 'cs51a' folder on your desktop.
- Download the following zip file and put its contents into your `nnlab` folder.

  `https://cs.pomona.edu/classes/cs51a/assignments/nn-lab-starter.zip`
- Create a new Python file in your PyCharm project. Add the following import statement at the top:

```
from cs51neural import *
```

  and then save the file as `nnlab.py`.

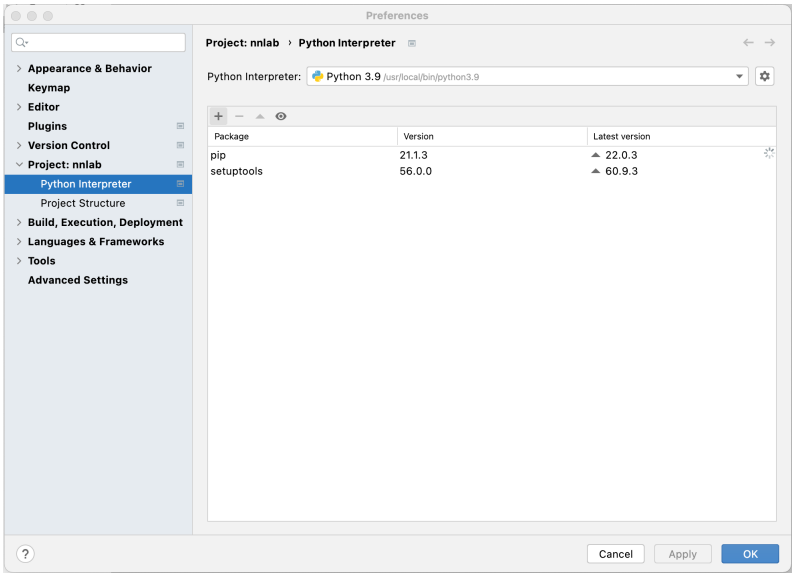
Before we can proceed, we need to install a *dependency*: `pytorch`, a library for doing stuff with computation graphs, including neural networks. This is just another module of Python code, but you don't have it on your computer (or in your project's *virtual environment* yet). We'll install that now.

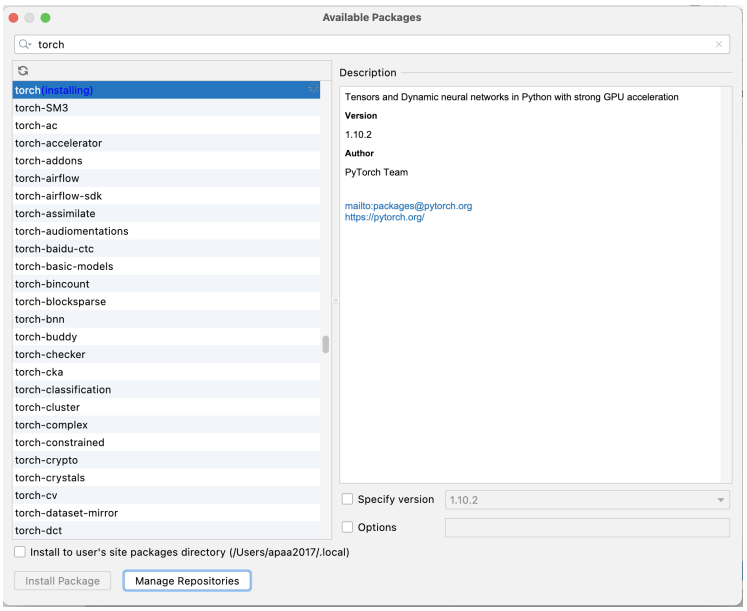
In PyCharm:

– Windows: open the menu item `File->Settings`

– Mac: open the menu item `PyCharm->Properties`

then choose `Project: nnlab>Project Interpreter` in the left hand column. Next, hit the plus button.





In the window you see next, search for `torch` (sometimes you have to wait a minute for it to download the options before the search works), select it from the list, and click `Install Package`. If all goes well you'll see a green box in this window stating `Package 'torch' installed successfully` and you can close this search window and the settings window. Keep in mind that you might need to separately install `numpy`, a different package.

If the above steps do not work for you, please open the terminal window on the bottom left of PyCharm, and manually type in "pip3 install torch" (and potentially "pip3 install numpy" if needed).

Note: when you play with the examples in the following sections, if pytorch doesn't work, then it may be due to the reason that pytorch is not compatible with the latest version of python (e.g., python 3.13). In this case, you need to install an older version of python (e.g., python 3.12 or python 3.11) and select this older version as your python interpreter in PyCharm for conducting the following lab exercises. Ask us for help :)

Data

The neural network package trains on labeled training data. Each example is a tuple consisting of the data and the label, both of which are represented as lists. For example, we saw the `xor` operator in class:

data		label
0	0	0
0	1	1
1	0	1
1	1	0

This data would be represented as:

```
xor = [([0, 0], [0]),
       ([0, 1], [1]),
       ([1, 0], [1]),
       ([1, 1], [0])]
```

I've written it across multiple lines to highlight the structure, but it could also have been written as:

```
xor = [([0, 0], [0]), ([0, 1], [1]), ([1, 0], [1]), ([1, 1], [0])]
```

Training the model

To use the neural network model, you first have to create a new instance of the neural network class. To create a new neural network you need to specify the configuration of the network. Our neural network package implements a two-layer neural network, so you need to specify three parameters: the number of inputs into the network (this should be the number of features/dimensions your data has), the number of hidden nodes, and the number of output nodes (one, for most of our data).

Note that in PyTorch, unlike in the diagrams in our class, neurons themselves do *not* have activation functions although they do have an activation parameter learned along with the weights. Instead, an activation *layer* is placed after the neurons, applying to the net activation of each node in the previous layer. So, after a layer that computes activation, an activation function layer like `Sigmoid` or `ReLU` is inserted before the subsequent layer of the network.

Go ahead and read the definition of `__init__` in `cs51neural.py` to see an example of this. On scratch paper or the whiteboard, sketch out an example of the type of neural network that this module produces (e.g. for `NeuralNet(3, 2, 2)`) and check it with a TA or Prof. Ye.

For example, to create a network with two inputs, three hidden nodes, and one output node you would write:

```
nn = NeuralNet(2, 3, 1)
```

(Make sure you've imported the neural network package by either running your `nnlab.py` file or typing the import statement above.)

Calling `NeuralNet` creates a new neural network object and we save it into a variable called `nn`.

Now, just like other objects that we've used (e.g., lists, strings, dictionaries), we can call the methods on that object. For example, to train the model on the xor data (assuming you've defined the `xor` variable already) we can do the following:

```
>>> nn.train(xor)
error 1.694987
error 0.600590
error 0.225041
```

```
error 0.082480
error 0.024184
error 0.003562
error 0.000380
error 0.000037
error 0.000004
error 0.000000
```

(Note that your numbers will be slightly different since the network starts off with random weights, though it should be roughly in the same relative range. If your model's error does not converge, try to reset a few times. If you are still getting high errors, you might want to increase the number of hidden nodes.)

The default is that the program will train for 1000 iterations and it will output the training error (i.e., the error on the data that the network is training on) after every 100 iterations. As the model trains, you'd hopefully see the training errors generally getting smaller.

Once you've trained a model, you can apply it to data using the `test` method. For example, we could test on the training data (though, in practice, train and test should be different, it's a simple way to do a sanity check on your model).

```
>>> nn.test(xor)
[([1, 1], [0], [3.457069396972656e-05]),
 ([1, 0], [1], [0.999976634979248]),
 ([0, 1], [1], [0.9998949766159058]),
 ([0, 0], [0], [0.00013521313667297363])]
```

`test` returns the triplets of: data, label/correct answer, model output. You can process them in a more friendly way by unpacking the tuple:

```
for triple in nn.test(xor):
    (input, label, prediction) = triple # unpack the triple
     # remember outputs are lists, so we need to get the first item from the list
     print(str(input) + "\t" + str(label[0]) + "\t" + str(prediction[0]))
```

If you ran this, you'd see the predictions are starting to get pretty close to the real values:

```
[1, 1] 0 3.457069396972656e-05
[1, 0] 1 0.999976634979248
[0, 1] 1 0.9998949766159058
[0, 0] 0 0.00013521313667297363
```

If you want to evaluate just one input, you may do it with the `evaluate` method. For example, if we wanted to see what the answer was for `(`[0,0]):

```
>>> print(nn.evaluate([0, 0]))
[0.00013521313667297363]
```

This is the basic functionality of the neural network class. See the Appendix at the end of this document for more information about the different methods. In particular, take a look at the documentation for the `train` method since it has a number of optional parameters that you can specify to change how the training works. For example, to train the model for 2000 iterations we can specify the `iterations` optional parameter:

```
>>> nn.train(xor, iterations=2000)
error 1.258905
error 0.728916
error 0.647577
error 0.598436
error 0.517484
error 0.375881
error 0.207650
error 0.084402
error 0.028093
error 0.008192
error 0.002118
error 0.000545
error 0.000131
error 0.000031
error 0.000008
error 0.000002
error 0.000000
error 0.000000
error 0.000000
error 0.000000
```

It prints out 20 errors now, one every 100 iterations. If I wanted to do this same thing, but only print out the errors every 200 iterations:

```
>>> nn.train(xor, iterations=2000, print_interval=200)
error 1.310576
error 0.782371
error 0.663791
error 0.626011
error 0.331933
error 0.003130
error 0.000025
error 0.000000
error 0.000000
error 0.000000
```

Experiments

Now that you're comfortable using the neural network class, let's run a few experiments. Jot down notes as you go for discussion later.

1. Construct a network with two hidden nodes and train it on the XOR data below. Notice that we have complete information about the function we are trying to approximate—an unusual situation for a neural network. How quickly do the weights/errors converge, i.e., to the point where the difference from iteration to iteration is small? How well does the resulting network perform? Try training several different networks to see the variation in the convergence rate. You may also want to reduce `print_interval` to get finer information about the changes in the error.

inputs		output
0	0	0
0	1	1
1	0	1
1	1	0

2. Repeat part 1 with a network with eight hidden nodes. Does the convergence go faster? Is the resulting network a better approximation to the XOR function?

   Notice that "faster" can mean "fewer iterations" or "less clock time." A network with more nodes will have more weights to adjust and will take more clock time for each training cycle—but it *may* require fewer iterations to perform well and the total time *may* be shorter.

3. See what happens when you repeat part 1 using a network with just one hidden node. The XOR function cannot be computed with a single node. Why is a network with one hidden node equivalent to a single node?

4. Table 1 contains a sampling of voter opinions. The idea is to deduce voters party affiliations from their views on the importance of various topics. Six voters were asked to rate the importance of five issues on a scale from 0.0 to 1.0 and to identify themselves as Democrat (0.0) or Republican (1.0).

   Write a neural network and train it on the data in Table 1. Then try it on the samples from Table 2 or other cases of your own creation. Can you explain the conclusions drawn by the network?

5. Once you've finished these experiments, look at the three questions at the beginning of the handout and decide which one is the most interesting to you. Spend a bit more time playing with examples and coming up with concrete data/examples that illustrate any observations that you have. Be prepared to share these at the end of lab.

budget	defense	crime	environment	social security	party
0.9	0.6	0.8	0.3	0.1	1.0
0.8	0.8	0.4	0.6	0.4	1.0
0.7	0.2	0.4	0.6	0.3	1.0
0.5	0.5	0.8	0.4	0.8	0.0
0.3	0.1	0.6	0.8	0.8	0.0
0.6	0.3	0.4	0.3	0.6	0.0

Table 1: A sampling of voter opinions. This example is taken from notes by Dave Reed of Creighton University.

budget	defense	crime	environment	social security	party
1.0	1.0	1.0	0.1	0.1	?
0.5	0.2	0.2	0.7	0.7	?
0.8	0.3	0.3	0.3	0.8	?
0.8	0.3	0.3	0.8	0.3	?
0.9	0.8	0.8	0.3	0.6	?

Table 2: Some test cases for the voter network.

And in a format that's friendly for our package:

```
voter_train = [([0.9,0.6,0.8,0.3,0.1],[1.0]),
               ([0.7,0.2,0.4,0.6,0.3],[1.0]),
               ([0.5,0.5,0.8,0.4,0.8],[0.0]),
               ([0.3,0.1,0.6,0.8,0.8],[0.0]),
               ([0.6,0.3,0.4,0.3,0.6],[0.0])]
```

Appendix: The Neural Network Class

The software we're playing with represents a neural network as an *object* of the `NeuralNet` *class.* We've already seen instances of this, for example, *lists* are a class of objects.

The main difference between a `list` object and our neural network object is that there is no special way for constructing neural network objects (to create lists we use the square brackets, `[]`). Instead, there is a special function called a *constructor* that takes some number of arguments (in our case, three) and creates a neural network object. Once you have that object, you can call methods on it just like you would any other object.

In the following summary, the neural network object is named `nn`. You may, of course, give your networks any name you like.

`nn = NeuralNet(num_input, num_hidden, num_output)` this calls the constructor and creates a network with the specified number of nodes in each layer. The initial weights are random values between -2.0 and 2.0. Notice that all of the other methods below are called *on* a neural network object that has been created. You may also choose to create the network with a different activation function by e.g. `nn = NeuralNet(num_input, num_hidden, num_output, activation='relu')`. Allowed values are `'relu'`, `'tanh'`, and `'sigmoid'`.

`nn.evaluate(input)` returns the output of the neural network when it is presented with the given input. Remember that the input and output are *lists.*

`nn.train(training_data)` carries out a training cycle. As specified earlier, the training data is a list of input-output pairs. There are four optional arguments to the `train` function:

`learning_rate` defaults to 0.5.

`iterations` defaults to 1000. It specifies the number of passes over the training data.

`print_interval` defaults to 100. The value of the error is displayed after `print_interval` passes over the data; we hope to see the value decreasing. Set the value to 0 if you do not want to see the error values.

You may specify some, or all, of the optional parameters by name in the following format.

```
nn.train(training_data,
         learning_rate=0.05,
         iterations=100,
         print_interval=5)
```

`nn.test(testing_data)` evaluates the network on a list of examples. As mentioned above, the testing data is presented in the same format as the training data. The result is a list of triples which can be used in further evaluation.

`nn.get_IH_weights()` returns a list of two lists representing the weights between the input and hidden layers. If there are t input nodes and u hidden nodes, the result will be first a list containing u lists of length t, then a list of u activation parameters.

`nn.get_HO_weights()` returns a list of two lists representing the weights between the hidden and output layers. If there are u hidden nodes and v output nodes, the result will be first a list containing v lists of length u, then a list of v activation parameters.