ML with Pytorch on Google's Colab

AI – Gabor / Kim – 26 May '20

Introduction:

This document is a complete starting framework for Machine Learning (ML) using Pytorch on colab.research.google.com.

The site https://colab.research.google.com hosts a python development environment with some very powerful libraries. It has a Jupyter notebook environment. If you have not yet used it, you may think of it as similar to the Python interpreter that you get in the command window if you type `python3` (or `python`), except that (1) You (the user) group sequences of statements together into cells. (2) The cells/statements are inert until you direct them to run (vs. being executed immediately in the python interpreter) (3) You may run (and rerun) cells in an arbitrary order.

Now that you have an appreciation for the mathematics and low level implementation of neural networks, we're going to examine working with more powerful tools where the object is to create a neural network that works well rather than creating one from scratch.

.

There are some useful keyboard shortcuts (may be a bit different on a Mac).

Ctrl+m, h	Show shortcuts	Ctrl+m, d	Delete cell	Ctrl+m, b	Insrt cel belw
Ctrl+Enter	Run cell	Ctrl+F9	Run all cells	Ctrl+m, -	Split cel @ cursr
Ctrl+/	toggle comment	Ctrl+m, o	Toggle output	Ctrl+m, n	Goto next cell

In the next sections, we'll determine a neural network to compute an *exclusive-or* from three inputs. We will accomplish this with 5 cells.

Cell 1: Environment setup

```
import torch
```

This framework is minimalist, and we should only need the line above. Just in case the above is not sufficient for you, the below is what we saw in class:

```
# Pytorch should already be installed so the below line is probably redundant
!pip3 install torch torchvision
device = 'cpu'            # Default device
import sys
print(sys.version)        # To know python's version
import torch.nn as nn     # A convenience
```

Cell 2: Training data and NN structure

```
zo = [0.0,1.0]              # zo = zero-one
ins = torch.tensor([[a,b,c] for a in zo for b in zo for c in zo])
expected = torch.tensor([[sum(tpl)%2] for tpl in ins])
nodeCts = [len(ins[0]), 2, 1]
```

Although this code is compact, it bears a few comments. The first line sets up the possible input values. I have checked that they do need to be floats for pytorch to be able to handle them. The next line gets all eight possible combinations of three inputs. However, these 8 triples are arranged in a format internal to pytorch:

```
tensor([[0., 0., 0.],
        [0., 0., 1.],
        [0., 1., 0.],
        [0., 1., 1.],
        [1., 0., 0.],
        [1., 0., 1.],
        [1., 1., 0.],
        [1., 1., 1.]])
```

I encourage you to liberally enhance this code with `print` statements as you experiment with ML not just to see values, but to also observe structure. An exclusive-or is just the sum of its inputs mod 2, but there were two problems with the 3rd line of code in class. The most important was that a pair of brackets was missing during 6th period, so the code only accidentally worked because a second bug cancelled it out.

`nodeCts` specifies how many nodes there will be in each layer of the neural net. There are two things to note. The first is that the bias is not included here (in other words, there is no `+ 1` in the first element of `nodeCts` even though we have a bias. We'll add that "manually" in the next section. The second is that since our output nums always ended with 1 1, we'll drop the final 1 and remember it is implied. Thus, instead of `[4, 2, 1, 1]`, `nodeCts` will be `[3, 2, 1]`.

As a reminder, if you want to get ahold of a specific number in a pytorch tensor, then you get at it via `.item()`. You may see this in action in cell 4 with `loss.item()`.

Cell 3: NN creation

```
netSpec = [torch.nn.Sigmoid() if i%2 else
           torch.nn.Linear(nodeCts[i//2], nodeCts[1+i//2],bias=i<1)
             for i in range(2*len(nodeCts)-2)]
mynn = torch.nn.Sequential(*netSpec, torch.nn.Linear(1,1,bias=False))
criterion = torch.nn.MSELoss()
```

In the first statement we arrange all the elements that we want to use in our neural network. The arrangement happens by layers. We have two basic choices: we can either have nodes with a logistic transfer function (torch.nn.Sigmoid) or we can linearly combine the outputs of a prior layer to act as inputs to the next layer (torch.nn.Linear). Thus our neural network looks like:

[inputs, linear(3->2), sigmoid, linear(2->1), sigmoid, linear(1->1), output]

The three inputs and one output don't need to be specified in the NN structure (they're implied by the first 3 below and the last 1), so that leaves

[Linear(3,2), Sigmoid(), Linear(2,1), Sigmoid(), Linear(1,1)]

When actually specifying the Linear and Sigmoid elements in code, they should be prefixed with a `torch.nn.` However, there is an additional factor to account for and that is the biases. Although we considered biases going into nodes where we were learning about neural nets, in pytorch they are part of the Linear layer type. In particular, for each input value (of a Linear layer), there is a set of weights on the wires going to a particular output. One would take a dot product to get the output value. With the Linear unit there is a quantity that could be added to this dot product, and that is considered the bias. It need not feed into a Sigmoid unit. The default is for the Linear unit to have a bias for each output, and one turns it off by specifying `bias=False` in the arguments to Linear. Recollecting that our original lab only had bias in the intputs to the first layer of nodes, that means that the NN structure is actually specified by:

[Linear(3,2), Sigmoid(), Linear(2,1,bias=False), Sigmoid(), Linear(1,1,bias=False)]

The first statement in the cell accounts for the first four elements, while the second statement addresses the final Linear item. The first line takes care of the two Sigmoids (when `i` is 1 and 3), while the second line (when `i` is 0 and 2) handles the first two Linear elements. The reason that I've assembled them this way is that this scheme is adaptable to other layer configurations and only
nodeCts in Cell 2 needs to be changed. You are not compelled to use this same architecture, but at the end of this exercise, you'll adapt the neural net that your code discovers to the format that the grader knows, and this structure maps nicely.

The second statement does the actual creation of the neural net, and that statement wants to know about the layers, in order. Remember that the asterisk in front of a list dereferences it, so it's as if you're listing all elements with commas separating them. You may notice that weights haven't been mentioned yet, but we know they must be part of the NN, along with any biases. In fact,

these actually get created when you call Linear and not when you create the NN. This is relevant because if you want to reset the neural network as you are debugging, you must re-execute the first line of this cell. It is incorrect to start re-execution from the 2nd line of the cell (in particular, don't move the first line of this cell to the prior cell).

The third statement sets the error criteria – the quantity that should be minimized, the Mean Squared Error. The loss (ie. error) that we'll output in cell 4 is the average of the 8 individual errors. There are other possibilities (which I don't recommend at this point). For example, one could give an argument such as `criterion = torch.nn.MSELoss(reduction= 'sum')` so the error reported (in cell 4) would be the sum of the 8 individual errors, rather than the average. This would likely be confusing if you had different sizes of input that you were working with.

Cell 4: Training the NN – Back propagation

```
optimizer = torch.optim.SGD(mynn.parameters(), lr=0.5)
for epoch in range(40000+1):
  y_pred = mynn(ins)                    # Forward propagation
  loss = criterion(y_pred, expected)    # Compute and print error
  if not epoch%500 or epoch<10:
    print('epoch: ', epoch,' loss: ', loss.item())

  optimizer.zero_grad()                 # Zero the gradients
  loss.backward()                       # Back propagation
  optimizer.step()                      # Update the weights
```

The first line in the cell sets the optimization method (Stochastic Gradient Descent). Of more interest to us, it also sets the learning rate. You should be able to update this, should you wish, as you are running your back propagation algorithm. `mynn.parameters()` refers to the weights and biases.

Each epoch goes through each input output value once. 40000 is a large number, but I've frequently seen the NN flail about at a certain error point (specifically losses of .25 and .125 seem to trouble it the most) that it might take a long time to get past, after which the loss diminishes rapidly (for a while).

y_pred (standing for predicted y) are the actual outputs as a result of feeding the current NN the input values. In other words, they are the feed forward values. The next line determines the total error for this epoch. The next two print out the error – the first several at the beginning and then every so often.

The next section does the back propagation. The zero_grad line nulls out any prior gradient. The line after that does the actual back propagation, meaning that it determines the gradient. Finally optimizer.step() amends the weights using the gradient values.

Note that if you rerun this cell, your NN should improve because the weights are not reset. It also allows you to change the learning rate manually, if you'd like to alter it. If you rerun Cell 3, however, then your NN will reset.

What does not appear in the code above is handling the cases where the back propagation hits a dead end, either by increasing or stagnating. In other words, there are no reset and start over scenarios built in. For this initial lab, that is fine, as our point here is to put in the basic plumbing. You will subsequently modify your code to suit your purposes.

Cell 5: Output the NN

```
for k,v in mynn.state_dict().items(): print(k,v)
```

This part is fairly straightforward, with one exception. The weights are in a different order from how the grader takes them (although it was easier at the start, this was the downside to that labeling). However, it should be easy to understand the labeling scheme based on this example.

Application:

The first step is to get the above to work so that you know that you have a working environment. Once you have that, you are on your way to solving more significant problems.

For example, you could modify your code to get some really good results for the circle lab where the circle in question is the unit circle. What you are looking for is an actual neural net (ie. the weights, along with any biases) that will identify whether a point is in a unit circle or not. If the NN outputs a .5 or greater, the point is construed to be outside the circle and otherwise inside the circle. It is fine if your code has to run for an extended period of time to converge. Make sure that you can get ahold of the weights electronically because you will be using the actual NN weights that you come up with.