Sylvain Gugger

about me

fast.ai

A simple neural net in numpy

Posted on Tue 20 March 2018 in Basics

Numpy doesn't have GPU-acceleration, so this is just to force us to understand what's going on behind the scenes, and how to code the things pytorch does automatically. The main thing we have to dig into is how it computes the gradient of the loss with respect to all the parameters of our neural net. We'll see that despite the fact it seems a very hard thing to do, calculating the gradients involves roughly the same things (so takes approximately the same time) as computing the outputs of our network from the outputs.

Back propagation

If we take the same example as in this article our neural network has two linear layers, the first activation function being a ReLU and the last one softmax (or log softmax) and the loss function the Cross Entropy. If we really wanted to, we could write down the (horrible) formula that gives the loss in terms of our inputs, the theoretical labels and all the parameters of the neural net, then compute the derivatives with respect to each weight and each bias, and finally, implement the corresponding formulas.

Needless to say that would be painful (though what we'll do still is), and not very helpful in general since each time we change our network, we would have to redo the whole process. There is a smarter way to go that relies on the chain rule. Basically, our loss function is just a composition of simpler functions, let's say:

$$\text{loss} = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_n(x)$$

where f_1 would be the Cross Entropy, f_2 our softmax activation, f_3 the last linear layer and so on... Furthermore, let's note

$$\begin{cases} x_1 = f_n(x) \\ x_2 = f_{n-1}(f_n(x)) \\ \vdots \\ x_n = f_1 \circ f_2 \circ f_3 \circ \cdots \circ f_n(x) \end{cases}$$

Then our loss is simply x_n. We can compute (almost) easily the derivatives of all the f_i (because they are the simple parts composing our loss function) so we will start from the very end and go backward until we reach $x_n = x$. At the end, we have $x_n = f_n(x_{n-1})$ so

$$\frac{\text{d}loss}{\text{d}x_{n-1}} = \frac{\text{d}f_n}{\text{d}x_{n-1}}(x_{n-1})$$

Then $x_{n-1} = f_{n-1}(x_{n-2})$ so by the chain rule

$$\frac{\text{d}loss}{\text{d}x_{n-2}} = \frac{\text{d}loss}{\text{d}x_{n-1}} \times \frac{\text{d}f_{n-1}}{\text{d}x_{n-2}}(x_{n-2}) = \frac{\text{d}f_n}{\text{d}x_{n-1}} \times \frac{\text{d}f_{n-1}}{\text{d}x_{n-2}}(x_{n-2})$$

and so forth. In practice, this is just a tinier bit more complicated than this when the function f_i depends on more than one variable, but we will study that in details when we need it (for the softmax and a linear layer).

To code this in a flexible manner, and since I need some training in Oriented Object Programming in Python, we will define each tiny bit of our neural network as a class. Each one will have a forward method (that gives the result of an input going through that layer or activation function) and a backward method (that will compute the step in the back propagation of going from after this layer/activation to before). The forward method will get the output of the last part of the neural net to start the process.

The backward method will get the derivatives of the loss function with respect to the next part of the layer and will have to compute the derivatives of the loss function with regards to the inputs it received. If that sounds unclear, reread this after the next paragraph and I hope it'll make more sense.

It seems complicated but it's not that difficult, just a bit of math to bear with. Our hard bits will be the linear layer and the softmax activation, so let's keep them for the end.

Activation function

If f is an activation function, it receives the result of a layer x and is applied element-wise to compute the output which is $y = f(x)$. In practice, x is a whole mini-batch of inputs, so it's an array with as many rows as the size of our mini-batch and as many columns as there were neurons in the previous layer. It's not really important since the function is applied element-wise, so we can safely imagine that x is just one entry of the array.

The forward pass is straightforward, and the back propagation step isn't really complicated either. If we know the derivative of our loss function with respect to y, then

$$\frac{\text{d}loss}{\text{d}x} = \frac{\text{d}loss}{\text{d}y} \times \frac{\text{d}f}{\text{d}x} = \frac{\text{d}loss}{\text{d}y} \times f'(x).$$

where f' is the derivative of the function f. So if we receive a variable named grad that contains all the derivatives of the loss with respect to all the y, the derivatives of the loss with respect to all the x is simply

$$f'(x) \odot \text{grad}$$

where f' is applied element-wise and $\odot$ represents the product of the two arrays element-wise. Note that we will need to know x when it's time for the back propagation step, so let's save it when we do the forward pass inside a parameter of our class.

The easier to implement will be the ReLU activation function. Since we have

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x \text{ si } x > 0 \\ 0 \text{ sinon} \end{cases}$$

we can compute its derivative very easily:

$$\text{ReLU}'(x) = \begin{cases} 1 \text{ si } x > 0 \\ 0 \text{ sinon} \end{cases}$$

Then the ReLU class is easily coded.

```
class ReLU():
    def forward(self, x):
        self.old_x = np.copy(x)
        return np.clip(x,0,None)

    def backward(self, grad):
        return np.where(self.old_x>0,grad,0)
```

We just simplified the multiplication between grad and an array of 0 and 1 by the where statement.

We won't need it for our example of neural net, but let's do the sigmoid to. It's defined by

$$\sigma(x) = \frac{e^x}{1+e^x}$$

and by using the traditional rule to differentiate a quotient,

$$\begin{aligned} \sigma'(x) &= \frac{e^x(1+e^x) - e^x.e^x}{(1+e^x)^2} = \frac{e^x}{1+e^x} - \frac{e^{2x}}{(1+e^x)^2} \\ &= \sigma(x) - \sigma(x)^2 = \sigma(x)(1-\sigma(x)) \end{aligned}$$

Then the sigmoid class is

```
class Sigmoid():
    def forward(self, x):
        self.old_y = np.exp(x) / (1. + np.exp(x))
        return self.old_y

    def backward(self, grad):
        return self.old_y * (1. - self.old_y) * grad
```

Note that here we store the result of the forward pass and not the old value of x, because that's what we will need for the back propagation step.

The tanh class would be very similar to write.

Softmax

The softmax activation is a bit different since the results depends of all the inputs and it's not just applied to each element. If our input is $x_1, \ldots, x_n$ the output $y_1, \ldots, y_n$ is defined by

$$\text{softmax}_i(x) = y_i = \frac{e^{x_i}}{e^{x_1} + \cdots + e^{x_n}}$$

When we want to take the derivative of y_i, we have 4 different variables with respect to which differentiate. We compute

$$\begin{aligned} \frac{\partial y_i}{\partial x_i} &= \frac{e^{x_i}(e^{x_1} + \cdots + e^{x_i}) - e^{x_i} \times e^{x_i}}{(e^{x_1} + \cdots + e^{x_i})^2} \\ &= \frac{e^{x_i}}{e^{x_1} + \cdots + e^{x_i}} - \frac{e^{2x_i}}{(e^{x_1} + \cdots + e^{x_i})^2} \\ &= y_i - y_i^2 = y_i(1-y_i) \end{aligned}$$

and if $i \neq i$

$$\begin{aligned} \frac{\partial y_i}{\partial x_j} &= -\frac{e^{x_i}e^{x_j}}{(e^{x_1} + \cdots + e^{x_n})^2} \\ &= \frac{e^{x_i}}{e^{x_1} + \cdots + e^{x_n}} \times \frac{e^{x_j}}{e^{x_1} + \cdots + e^{x_n}} \\ &= -y_i y_j \end{aligned}$$

Now we will get the derivatives of the loss with respect to the y_i and we will have to compute the derivatives of the loss with respect to the x_i. In this case, since each y_i depends on the variable x_i, the chain rule is written:

$$\begin{aligned} \frac{\text{d}loss}{\text{d}x_i} &= \sum_{i=1}^{n} \frac{\text{d}loss}{\text{d}y_i} \times \frac{\partial y_i}{\partial x_i} \\ &= \frac{\text{d}loss}{\text{d}y_i}(y_i - y_i^2) - \sum_{i \neq j} y_i y_j \frac{\text{d}loss}{\text{d}y_i} \\ &= y_i \frac{\text{d}loss}{\text{d}y_i} - \sum_{i=1}^{n} y_i y_j \frac{\text{d}loss}{\text{d}y_i} \end{aligned}$$

Now when we implement this, we have to remember that x is a mini-batch of inputs. In the forward pass, the sum that we see in the denominator is to be taken on each line of the exponential of x which we can do with

```
np.exp(x).sum(axis=1)
```

This array will have a shape of (mb,), were mb is the size of our mini-batch. We want to divide np.exp(x) by this array, but since x has a shape (mb,n), we have to convert this array into an array of shape (mb,1), otherwise the two won't be broadcastable (numpy tries to add the ones at the beginning of the shape when two arrays don't have the same dimension). The trick is done with resize or expand_dims:

```
np.expand_dims(np.exp(x).sum(axis=1),axis=1)
```

Another way that's shorter is to add a None index:

```
np.exp(x).sum(axis=1)[:,None]
```

For the backward function, we can note that in our formula, y_i can be factored, then the values we have to compute are the

$$y_i \left(g_i - \sum_{i=1}^{n} y_i g_i \right)$$

where I noted g the gradient of the loss with respect to the y_i. Again, the sum is to be taken on each line (and we have to add a dimension this time as well), which gives us:

```
class Softmax():
    def forward(self,x):
        self.old_y = np.exp(x) / np.exp(x).sum(axis=1) [:,None]
        return self.old_y

    def backward(self,grad):
        return self.old_y * (grad -(grad * self.old_y).sum(axis=1)[:,None])
```

Cross Entropy cost

The cost function is a little different in the sense it takes an output and a target, then returns a single real number. When we apply it to a mini-batch though, we have two arrays x and y of the same size (mb by n, the number of outputs) which represent a mini-batch of outputs of our network and the targets they should match, and it will return a vector of size mb.

The cross-entropy cost is the sum of the $-\ln(x_i)$ over all the indexes i for which y_i equals 1. In practice though, just in case our network returns a value of x_i to close to zero, we clip its value to a minimum of 10^{-8} (usually).

For the backward function, we don't have any old gradients to pass, since this is the first step of computing the derivatives of our loss. In the case of the cross-entropy loss, those are $-\frac{1}{x_i}$ for each i for which y_i equals 1, 0 otherwise. Thus we can code:

```
class CrossEntropy():
    def forward(self,x,y):
        self.old_x = x.clip(min=1e-8,max=None)
        self.old_y = y
        return (np.where(y==1,-np.log(self.old_x), 0)).sum(axis=1)

    def backward(self,y):
        return np.where(self.old_y==1,-1/self.old_x, 0)
```

Linear Layer

We have done everything else, so now is the time to focus on a linear layer. Here we have a few parameters, the weights and the biases. If the layer we consider has n_{in} inputs and n_{out} outputs, the weights are stored in a matrix W of size n_{in}, n_{out} and the bias is a vector B. The output is given by $Y = XW + B$ where X is the input.

This formula can be seen for just one vector of inputs or a mini-batch, in the second case, we just have to think of B as a matrix with mb lines, all equal to the bias vector (which is the usual broadcasting in numpy).

The forward pass will be very easy to implement, for the backward pass, not only will we have to compute the gradients of the loss with regards to X while given the gradients of the loss with regards to Y (to be able to continue our back propagation) but we will also have to calculate and store the gradients of the loss with regards to all the weights and biases, since those are the things we will need to do a step in our gradient descent (and the whole reason we are doing back propagation).

Let's begin with this. In terms of coordinates, the formula above can be rewritten

$$y_i = \sum_{k=1}^{n_{in}} x_k w_{i,j} + b_i$$

So we have immediately

$$\frac{\text{d}loss}{\text{d}b_i} = \frac{\text{d}loss}{\text{d}y_i} \cdot \frac{\partial y_i}{\partial b_i} = \frac{\text{d}loss}{\text{d}y_i}$$

and

$$\frac{\text{d}loss}{\text{d}w_{i,j}} = \frac{\text{d}loss}{\text{d}y_i} \cdot \frac{\partial y_i}{\partial w_{i,j}} = x_i \frac{\text{d}loss}{\text{d}y_i}$$

There are no sums here because b_i and $w_{i,j}$ only appear to define y_i.

If we have the derivatives of the loss with respect to the y_i in a variable called grad, the derivatives of the loss with respect to the biases are in grad, and the derivatives of the loss with respect to the weights are in the array

```
np.matmul(self.old_x[:,:,None],grad[:,None,:])
```

Why is that? Since x has a size (mb, n_{in}) and grad has a size (mb, n_{out}), we transform those two arrays into tensors with dimensions (mb, n_{in}, 1) and (mb, 1, n_{out}). That way, the traditional matrix product applied for the two last dimensions will give us, for each mini-batch, the product of x_i by grad,

As we explained in the introduction we average the gradients over the mini-batch to apply our step of the SGD, so we will store the mean over the first axis of those two arrays.

Then, once this is done, we still need to compute the derivatives of the loss with respect to the x_i. This is given by the formula

$$\frac{\text{d}loss}{\text{d}x_i} = \sum_{i=1}^{n_{out}} \frac{\text{d}loss}{\text{d}y_i} \times \frac{\partial y_i}{\partial x_i} = \sum_{i=1}^{n_{out}} \frac{\text{d}loss}{\text{d}y_i} w_{i,j}.$$

This can be rewritten as a simple matrix product:

$$new\ \text{grad} = (\text{old grad}) \times W.$$

We all of this, we can finally code our own linear class.

```
class Linear():
    def __init__(self,n_in,n_out):
        self.weights = np.random.randn(n_in,n_out) * np.sqrt(2/n_in)
        self.biases = np.zeros(n_out)

    def forward(self, x):
        self.old_x = x
        return np.dot(x,self.weights) + self.biases

    def backward(self,grad):
        self.grad_b = grad.mean(axis=0)
        self.grad_w = np.matmul(self.old_x[:,:,None],grad[:,None,:]).mean(axis=0)
        return np.dot(grad,self.weights.transpose())
```

Note that we initialize the weights randomly as we create the network. The rule usually used for this is explained here, I may write another article detailing the reasoning behind it later. The biases are initialized to zero.

We can now group all those layers in a model

```
class Model():
    def __init__(self, layers, cost):
        self.layers = layers
        self.cost = cost

    def forward(self,x):
        for layer in self.layers:
            x = layer.forward(x)
        return x

    def loss(self,x,y):
        return self.cost.forward(self.forward(x),y)

    def backward(self):
        grad = self.cost.backward()
        for i in range(len(self.layers)-1,-1,-1):
            grad = self.layers[i].backward(grad)
```

We can then create a model that looks like the one we defined for our digit classification like this:

```
net = Model([Linear(784,100), Relu(), Linear(100,10), Softmax()], CrossEntropy())
```

The training loop would then look like something like this:

```
def train(model,lr,nb_epoch,data):
    for epoch in range(nb_epoch):
        running_loss = 0.
        num_inputs = 0
        for mini_batch in data:
            inputs,targets = mini_batch
            num_inputs += inputs.shape[0]
            #Forward pass
            running_loss += model.loss(inputs,targets).sum()
            #Back propagation
            model.backward()
            #Update of the parameters
            for layer in model.layers:
                if type(layer) == Linear:
                    layer.weights -= lr * layer.grad_w
                    layer.biases -= lr * layer.grad_b
        print(f'Epoch {epoch+1}/{nb_epoch}: loss = {running_loss/num_inputs}')
```

To test it, we can use the data from the MNIST dataset loaded in this notebook, we just have to convert all the torch arrays obtained into numpy array, flatten the inputs, and for the targets, replace each label by a vector with zeros and a one (because that's what our loss function needs). Here's an example of how to do this:

```
def load_mini_batches(batch_size=64):
    tsfms = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.1307,), (0.3081,))])
    trn_set = datasets.MNIST('.', train=True, download=True, transform=tsfms)
    trn_loader = torch.utils.data.DataLoader(trn_set, batch_size=batch_size, shuffle=True, num_workers=4)
    data = []
    for mb in trn_loader:
        inputs_t,targets_t = mb
        inputs = np.zeros((inputs_t.size(0),784))
        targets = np.zeros((inputs_t.size(0),10))
        for i in range(0,inputs_t.size(0)):
            targets[i,targets_t[i]] = 1.
            for j in range(0,28):
                for k in range(0,28):
                    inputs[i,j*28+k] = inputs_t[i,0,j,k]
        data.append((inputs,targets))
    return data
```

It's slower than the pytorch version but at least we can say we've fully got in the whole details of building a neural net from scratch.

Neural Net | Back Propagation