PyTorch-Assignment

April 27, 2020

0.1 PyTorch Assignment

You have to implement ReLU, Softmax, and Neuron using PyTorch. Most probably, you have already implemented these functions using NumPy in the NumPy assignment.

0.2 Marking Scheme

Maximum Points: 30

```
<table>
    <tr><td><h3>Sr. no.</h3></td> <td><h3>Problem</h3></td> <td><h3>Points</h3></td> </tr>
    <tr><td><h3>1</h3></td> <td><h3>ReLU Implementation</h3></td> <td><h3>5</h3></td> </tr>
    <tr><td><h3>2</h3></td> <td><h3>Softmax Implementation</h3></td> <td><h3>10</h3></td> </tr
    <tr><td><h3>3</h3></td> <td><h3>Neuron Implementation</h3></td> <td><h3>15</h3></td> </tr>
</table>
```

```
In [1]: import torch
```

0.3 1. ReLU Implementation (5 Points)

ReLU stands for Rectified Linear Unit. It defined as follows:

```
z = max(0, x)
```

In this part, you have to implement this ReLU function definition for the NumPy array. So if the input is:

```
tensor([[ 1.0000,  2.0000, -3.0000],
        [ 2.5000, -0.2000,  6.0000]])
```

You have to return the following output:

```
tensor([[1.0000, 2.0000, 0.0000],
        [2.5000, 0.0000, 6.0000]])
```

```
<b>1. ReLU Implementation: 5 Points</b>
```

```
In [2]: def ReLU(tensor):

            ###
            tensor_from_array = torch.tensor(tensor)
            return torch.clamp(tensor, min = 0)
            ###
```

```
In [3]: # test your result
        a = torch.tensor([[1, 2, -3], [2.5, -0.2, 6]])
        ReLU(a)
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:4: UserWarning: To copy construct from a
  after removing the cwd from sys.path.
```

```
Out[3]: tensor([[1.0000, 2.0000, 0.0000],
                [2.5000, 0.0000, 6.0000]])
```

```
In [4]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

0.4 2. Softmax Implementation (10 Points)

Softmax is defined as follows:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

For example, we have following an array as an input:

```
tensor([0.6000, 5.2000, 9.2000])
```

Then, the function should return the following as output:

```
tensor([1.8076e-04, 1.7983e-02, 9.8184e-01])
```

```
<b>2. Softmax Implementation: 10 Points</b>
```

```
In [5]: def softmax(array):

            ###
            tensor_from_array = torch.tensor(array)
            exp = torch.exp(tensor_from_array)
            expo_sum = torch.sum(torch.exp(tensor_from_array))
            return exp / expo_sum

            ###
```

```
In [6]: # Test your result
        a = torch.tensor([0.6, 5.2, 9.2])
        softmax(a)
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:4: UserWarning: To copy construct from a
  after removing the cwd from sys.path.
```

```
Out[6]: tensor([1.8076e-04, 1.7983e-02, 9.8184e-01])
```

```
In [7]: ###
        ### AUTOGRADER TEST - DO NOT REMOVE
        ###
```

0.5 3. Neural Network Neuron Implementation (15 Points)

We all are familiar with the `1-dimensional` linear equation:

$$y = mx + c$$

We can re-write the equation as:

$$y = w_1 x + b$$

We can write the `n-dimensional` linear equation as follows:

$$y = w_1 x_1 + w_2 x_2 + ... + w_n x_n + b$$

Following is a pictorial representation `n-dimensional` linear equations. This linear function is called a neuron in neural networks.

let's define W as,

$$W = \begin{bmatrix} w_1 & w_2 & w_3 & \dots & w_n \end{bmatrix}$$

and X as,

$$X = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Using above W and X, we can re-write the n-dimensional linear equation as follows:

$$y = WX + b$$

In a neural network, X, W, b, and y are called input, weight, bias, and output of the neuron, respectively.

In a neural network, usually, we use to have much more than one neuron. The same input X passes through all neurons of the layer and gets output y for each neuron. We don't have to calculate linear function for each neuron at a time; we can calculate all in one go using Numpy.

Let's assume we have `m` neurons. So we will have `m` output. Let's stack all weight horizontally and do matrix multiplication by input and add stacked b. It will give `m` output for all `m` neurons.

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1n} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & w_{m3} & \dots & w_{mn} \end{bmatrix}$$

$$B = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

$$Y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

You have to implement the following function:

$$Y = WX + B$$

The function will take weight W, bias B, and input X as arguments. You have to return outputs Y.

For example, arguments W, B and X are as follows:

```
W = tensor([[1.2000, 0.3000, 0.1000],
            [0.0100, 2.1000, 0.7000]])
B = tensor([2.1000, 0.8900])
X = tensor([0.3000, 6.8000, 0.5900])
```

Function should return:

```
tensor([ 4.5590, 15.5860])
```

```
<b>3. Neuron Implementation: 15 Points</b>
```

```
In [8]: def neural_network_neurons(W, B, X):

            ###
            tensor_from_w = torch.tensor(W)
            tensor_from_b = torch.tensor(B)
            tensor_from_x = torch.tensor(X)
            Y = torch.matmul(W, X) + B
            return Y
            ###
```

```
In [9]: # test your code
        W = torch.tensor([[1.2, 0.3, 0.1], [.01, 2.1, 0.7]])
        B = torch.tensor([2.1, 0.89])
        X = torch.tensor([0.3, 6.8, 0.59])

        neural_network_neurons(W, B, X)
```

```
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:4: UserWarning: To copy construct from a
  after removing the cwd from sys.path.
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:5: UserWarning: To copy construct from a
  """
/usr/lib/python3.7/site-packages/ipykernel_launcher.py:6: UserWarning: To copy construct from a
```

```
Out[9]: tensor([ 4.5590, 15.5860])

In [10]: ###
         ### AUTOGRADER TEST - DO NOT REMOVE
         ###

In [ ]:
```