Watson Studio democratizes machine learning and deep learning to accelerate infusion of AI in your business to drive innovation. Watson Studio provides a suite of tools and a collaborative environment for data scientists, developers and domain experts.

(http://cocl.us/pytorch_link_top)

Linear Regression 1D: Prediction

Table of Contents

In this lab, we will review how to make a prediction in several different ways by using PyTorch.

Estimated Time Needed: **15 min**

Preparation

The following are the libraries we are going to use for this lab.

In [1]:

```
# These are the libraries will be used for this lab.

import torch
```

Prediction

Let us create the following expressions:

$b=-1,w=2$

$\hat{y}=-1+2x$

First, define the parameters:

In [2]:

```
# Define w = 2 and b = -1 for y = wx + b

w = torch.tensor(2.0, requires_grad = True)
b = torch.tensor(-1.0, requires_grad = True)
```

Then, define the function `forward(x, w, b)` makes the prediction:

In [3]:

```
# Function forward(x) for prediction

def forward(x):
    yhat = w * x + b
    return yhat
```

Let's make the following prediction at *x = 1*

$\hat{y}=-1+2x$

$\hat{y}=-1+2(1)$

In [4]:

```
# Predict y = 2x - 1 at x = 1

x = torch.tensor([[1.0]])
yhat = forward(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[1.]], grad_fn=<AddBackward0>)
```

Now, let us try to make the prediction for multiple inputs:

$$\hat{y} = -\mathbf{1} + 2\,\mathbf{x} \qquad \mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\hat{y} = \begin{bmatrix} -1 \\ -1 \end{bmatrix} + 2\begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \end{bmatrix} + \begin{bmatrix} 2x1 \\ 2x2 \end{bmatrix} = \begin{bmatrix} -1+2 \\ -1+4 \end{bmatrix} = \begin{bmatrix} 1 \\ 3 \end{bmatrix}$$

Let us construct the `x` tensor first. Check the shape of `x`.

In [5]:

```
# Create x Tensor and check the shape of x tensor

x = torch.tensor([[1.0], [2.0]])
print("The shape of x: ", x.shape)
```

```
The shape of x:  torch.Size([2, 1])
```

Now make the prediction:

In [6]:

```
# Make the prediction of y = 2x - 1 at x = [1, 2]

yhat = forward(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[1.],
        [3.]], grad_fn=<AddBackward0>)
```

The result is the same as what it is in the image above.

Practice

Make a prediction of the following `x` tensor using the `w` and `b` from above.

In [8]:

```
# Practice: Make a prediction of y = 2x - 1 at x = [[1.0], [2.0], [3.0]]

x = torch.tensor([[1.0], [2.0], [3.0]])
yhat = forward(x)
print('The prediction: ', yhat)
```

```
The prediction:  tensor([[1.],
        [3.],
        [5.]], grad_fn=<AddBackward0>)
```

Double-click **here** for the solution.

Class Linear

The linear class can be used to make a prediction. We can also use the linear class to build more complex models. Let's import the module:

In [9]:

```
# Import Class Linear

from torch.nn import Linear
```

Set the random seed because the parameters are randomly initialized:

In [10]:

```
# Set random seed

torch.manual_seed(1)
```

Out[10]:

```
<torch._C.Generator at 0x7f0ccc0361f0>
```

Let us create the linear object by using the constructor. The parameters are randomly created. Let us print out to see what *w* and *b*. The parameters of an `torch.nn.Module` model are contained in the model's parameters accessed with `lr.parameters()` :

In [11]:

```
# Create Linear Regression Model, and print out the parameters

lr = Linear(in_features=1, out_features=1, bias=True)
print("Parameters w and b: ", list(lr.parameters()))
```

```
Parameters w and b:  [Parameter containing:
tensor([[0.5153]], requires_grad=True), Parameter containing:
tensor([-0.4414], requires_grad=True)]
```

This is equivalent to the following expression:

$b=-0.44, w=0.5153$

$\hat{y}=-0.44+0.5153x$

A method `state_dict()` Returns a Python dictionary object corresponding to the layers of each parameter tensor.

In [12]:

```
print("Python dictionary: ",lr.state_dict())
print("keys: ",lr.state_dict().keys())
print("values: ",lr.state_dict().values())
```

```
Python dictionary:  OrderedDict([('weight', tensor([[0.5153]])), ('bia
s', tensor([-0.4414]))])
keys:  odict_keys(['weight', 'bias'])
values:  odict_values([tensor([[0.5153]]), tensor([-0.4414])])
```

The keys correspond to the name of the attributes and the values correspond to the parameter value.

In [13]:

```
print("weight:",lr.weight)
print("bias:",lr.bias)
```

```
weight: Parameter containing:
tensor([[0.5153]], requires_grad=True)
bias: Parameter containing:
tensor([-0.4414], requires_grad=True)
```

Now let us make a single prediction at *x = [[1.0]]*.

In [14]:

```
# Make the prediction at x = [[1.0]]

x = torch.tensor([[1.0]])
yhat = lr(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[0.0739]], grad_fn=<AddmmBackward>)
```

Similarly, you can make multiple predictions:

$$\hat{\boldsymbol{y}} = -\mathbf{0.44} + 0.51\mathbf{x} \qquad \mathbf{x} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

$$\hat{\boldsymbol{y}} = \begin{bmatrix} -0.44 \\ -0.44 \end{bmatrix} + 0.51 \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -0.44 \\ -0.44 \end{bmatrix} + \begin{bmatrix} 0.51x1 \\ 0.51x2 \end{bmatrix}$$

$$= \begin{bmatrix} -0.44 \\ -0.44 \end{bmatrix} + \begin{bmatrix} 0.51 \\ 1.02 \end{bmatrix} = \begin{bmatrix} 0.07 \\ 0.58 \end{bmatrix}$$

Use model `lr(x)` to predict the result.

In [15]:

```
# Create the prediction using linear model

x = torch.tensor([[1.0], [2.0]])
yhat = lr(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[0.0739],
        [0.5891]], grad_fn=<AddmmBackward>)
```

Practice

Make a prediction of the following `x` tensor using the linear regression model `lr`.

In [16]:

```
# Practice: Use the linear regression model object lr to make the prediction.

x = torch.tensor([[1.0],[2.0],[3.0]])
yhat = lr(x)
print('The prediction: ', yhat)
```

```
The prediction:  tensor([[0.0739],
        [0.5891],
        [1.1044]], grad_fn=<AddmmBackward>)
```

Double-click **here** for the solution.

Build Custom Modules

Now, let's build a custom module. We can make more complex models by using this method later on.

First, import the following library.

In [17]:

```
# Library for this section

from torch import nn
```

Now, let us define the class:

In [18]:

```
# Customize Linear Regression Class

class LR(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):

        # Inherit from parent
        super(LR, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

    # Prediction function
    def forward(self, x):
        out = self.linear(x)
        return out
```

Create an object by using the constructor. Print out the parameters we get and the model.

In [19]:

```
# Create the linear regression model. Print out the parameters.

lr = LR(1, 1)
print("The parameters: ", list(lr.parameters()))
print("Linear model: ", lr.linear)
```

```
The parameters:  [Parameter containing:
tensor([[-0.1939]], requires_grad=True), Parameter containing:
tensor([0.4694], requires_grad=True)]
Linear model:  Linear(in_features=1, out_features=1, bias=True)
```

Let us try to make a prediction of a single input sample.

In [20]:

```
# Try our customize linear regression model with single input

x = torch.tensor([[1.0]])
yhat = lr(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[0.2755]], grad_fn=<AddmmBackward>)
```

Now, let us try another example with multiple samples.

In [21]:

```
# Try our customize linear regression model with multiple input

x = torch.tensor([[1.0], [2.0]])
yhat = lr(x)
print("The prediction: ", yhat)
```

```
The prediction:  tensor([[0.2755],
        [0.0816]], grad_fn=<AddmmBackward>)
```

the parameters are also stored in an ordered dictionary :

In [22]:

```
print("Python dictionary: ", lr.state_dict())
print("keys: ",lr.state_dict().keys())
print("values: ",lr.state_dict().values())
```

```
Python dictionary:  OrderedDict([('linear.weight', tensor([[-0.193
9]])), ('linear.bias', tensor([0.4694]))])
keys:  odict_keys(['linear.weight', 'linear.bias'])
values:  odict_values([tensor([[-0.1939]]), tensor([0.4694])])
```

Practice

Create an object `lr1` from the class we created before and make a prediction by using the following tensor:

In [23]:

```
# Practice: Use the LR class to create a model and make a prediction of the followi
ng tensor.

x = torch.tensor([[1.0], [2.0], [3.0]])
lr1 = LR(1,1)
yhat = lr(x)
yhat
```

Out[23]:

```
tensor([[ 0.2755],
        [ 0.0816],
        [-0.1122]], grad_fn=<AddmmBackward>)
```

Double-click **here** for the solution.

About the Authors:

[Joseph Santarcangelo (https://www.linkedin.com/in/joseph-s-50398b136/)](https://www.linkedin.com/in/joseph-s-50398b136/) has a PhD in Electrical Engineering, his research focused on using machine learning, signal processing, and computer vision to determine how videos impact human cognition. Joseph has been working for IBM since he completed his PhD.

Other contributors: Michelle Carey (https://www.linkedin.com/in/michelleccarey/), Mavis Zhou (www.linkedin.com/in/jiahui-mavis-zhou-a4537814a)
