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)

Multiple Linear Regression

Table of Contents

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

Estimated Time Needed: **15 min**

Preparation

Import the libraries and set the random seed.

In [1]:

```
# Import the libraries and set the random seed

from torch import nn
import torch
torch.manual_seed(1)
```

Out[1]:

```
<torch._C.Generator at 0x7fc409057d10>
```

Prediction

Set weight and bias.

In [2]:

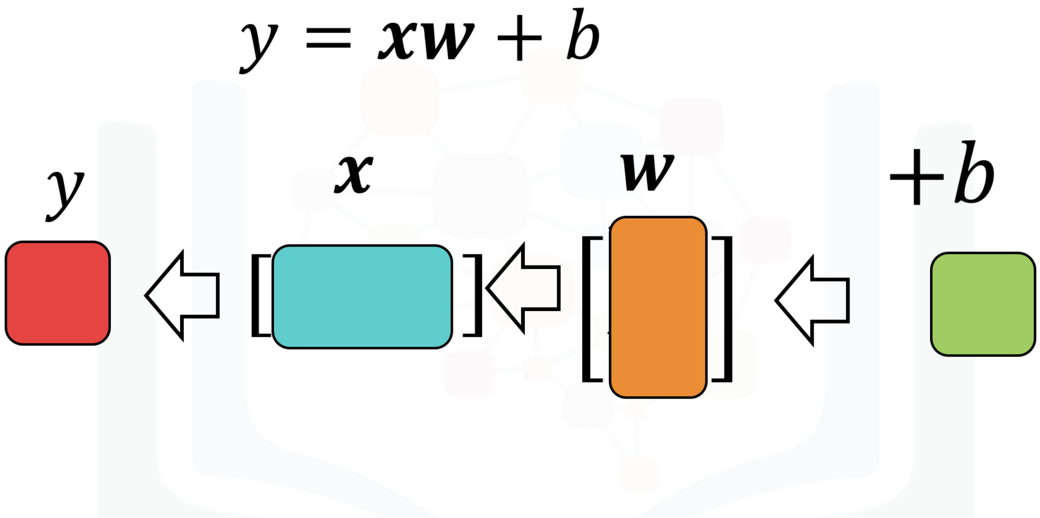
```
# Set the weight and bias

w = torch.tensor([[2.0], [3.0]], requires_grad=True)
b = torch.tensor([[1.0]], requires_grad=True)
```

Define the parameters. `torch.mm` uses matrix multiplication instead of scaler multiplication.

In [3]:

```
# Define Prediction Function

def forward(x):
    yhat = torch.mm(x, w) + b
    return yhat
```

The function `forward` implements the following equation:



If we input a *1x2* tensor, because we have a *2x1* tensor as `w` , we will get a *1x1* tensor:

In [4]:

```
# Calculate yhat

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

```
The result:  tensor([[9.]], grad_fn=<AddBackward0>)
```

$$b = \boxed{-1}, w = \boxed{\begin{bmatrix} 2 \\ 3 \end{bmatrix}}$$

$$\hat{y} = \boldsymbol{xw} + b$$

$$x = \boxed{[3,2]}$$

$$\boxed{\hat{y}} = \boxed{[1,2]} \boxed{\begin{bmatrix} 2 \\ 3 \end{bmatrix}} + \boxed{1}$$

$$\hat{y}: 9$$

Each row of the following tensor represents a sample:

In [5]:

```
# Sample tensor X

X = torch.tensor([[1.0, 1.0], [1.0, 2.0], [1.0, 3.0]])
```

In [6]:

```
# Make the prediction of X

yhat = forward(X)
print("The result: ", yhat)
```

```
The result:  tensor([[ 6.],
        [ 9.],
        [12.]], grad_fn=<AddBackward0>)
```

Class Linear

We can use the linear class to make a prediction. You'll also use the linear class to build more complex models.

Let us create a model.

In [7]:

```
# Make a linear regression model using build-in function

model = nn.Linear(2, 1)
```

Make a prediction with the first sample:

In [8]:

```
# Make a prediction of x

yhat = model(x)
print("The result: ", yhat)
```

```
The result:  tensor([[-0.3969]], grad_fn=<AddmmBackward>)
```

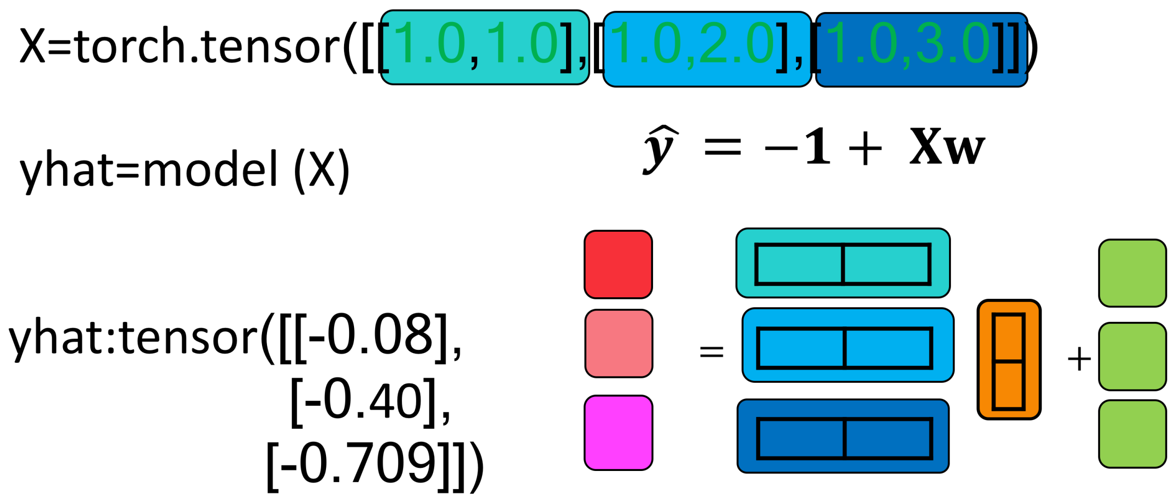
Predict with multiple samples `X` :

In [9]:

```
# Make a prediction of X

yhat = model(X)
print("The result: ", yhat)
```

```
The result:  tensor([[-0.0848],
        [-0.3969],
        [-0.7090]], grad_fn=<AddmmBackward>)
```

The function performs matrix multiplication as shown in this image:



Build Custom Modules

Now, you'll build a custom module. You can make more complex models by using this method later.

In [10]:

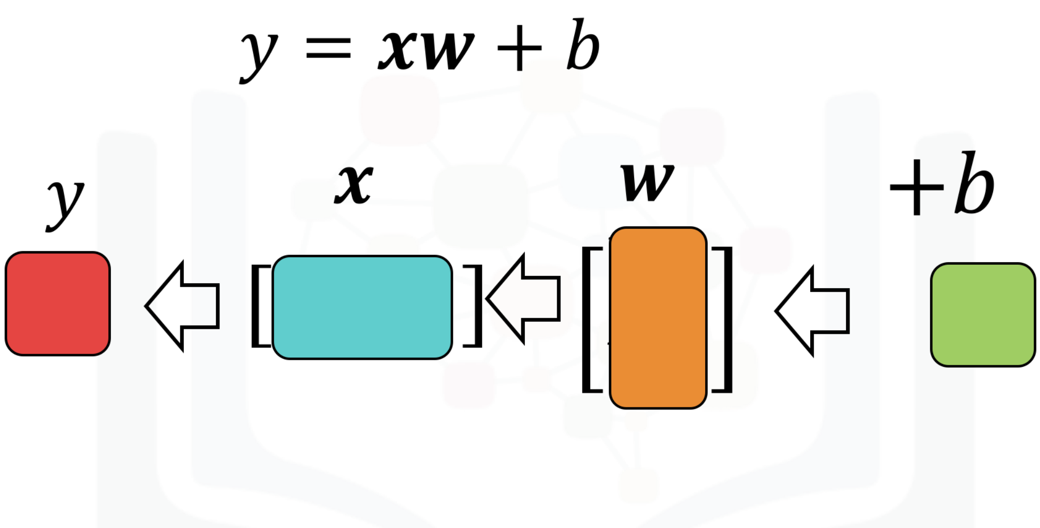
```
# Create linear_regression Class

class linear_regression(nn.Module):

    # Constructor
    def __init__(self, input_size, output_size):
        super(linear_regression, self).__init__()
        self.linear = nn.Linear(input_size, output_size)

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

Build a linear regression object. The input feature size is two.

In [11]:

```
model = linear_regression(2, 1)
```

This will input the following equation:



You can see the randomly initialized parameters by using the `parameters()` method:

In [12]:

```
# Print model parameters

print("The parameters: ", list(model.parameters()))
```

```
The parameters:  [Parameter containing:
tensor([[ 0.3319, -0.6657]], requires_grad=True), Parameter containing:
tensor([0.4241], requires_grad=True)]
```

You can also see the parameters by using the `state_dict()` method:

In [13]:

```
# Print model parameters

print("The parameters: ", model.state_dict())
```

```
The parameters:  OrderedDict([('linear.weight', tensor([[ 0.3319, -0.66
57]])), ('linear.bias', tensor([0.4241]))])
```

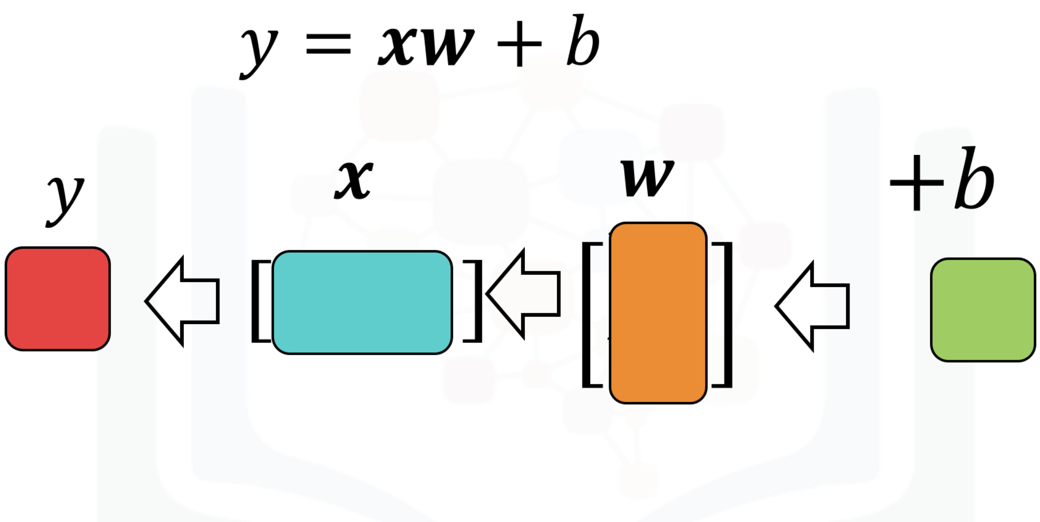
Now we input a 1x2 tensor, and we will get a 1x1 tensor.

In [14]:

```
# Make a prediction of x

yhat = model(x)
print("The result: ", yhat)
```

```
The result:  tensor([[-0.5754]], grad_fn=<AddmmBackward>)
```

The shape of the output is shown in the following image:



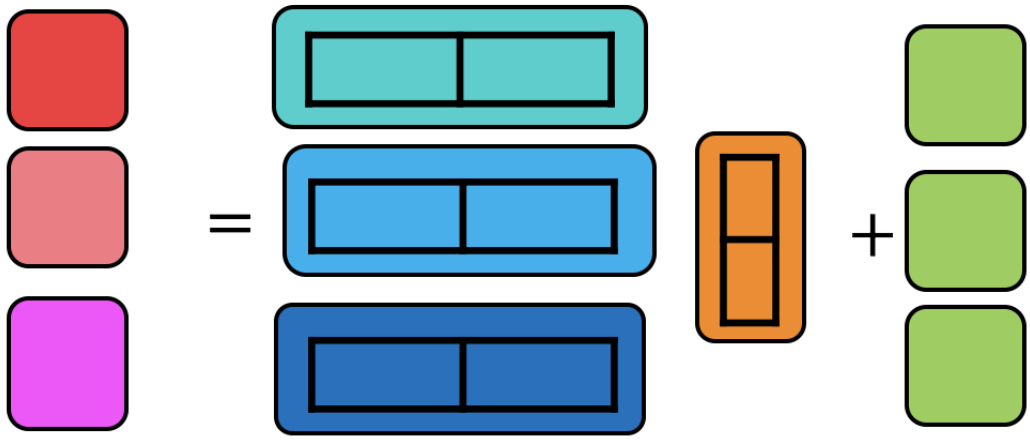
Make a prediction for multiple samples:

In [15]:

```
# Make a prediction of X

yhat = model(X)
print("The result: ", yhat)
```

```
The result:  tensor([[ 0.0903],
        [-0.5754],
        [-1.2411]], grad_fn=<AddmmBackward>)
```

The shape is shown in the following image:



Practice

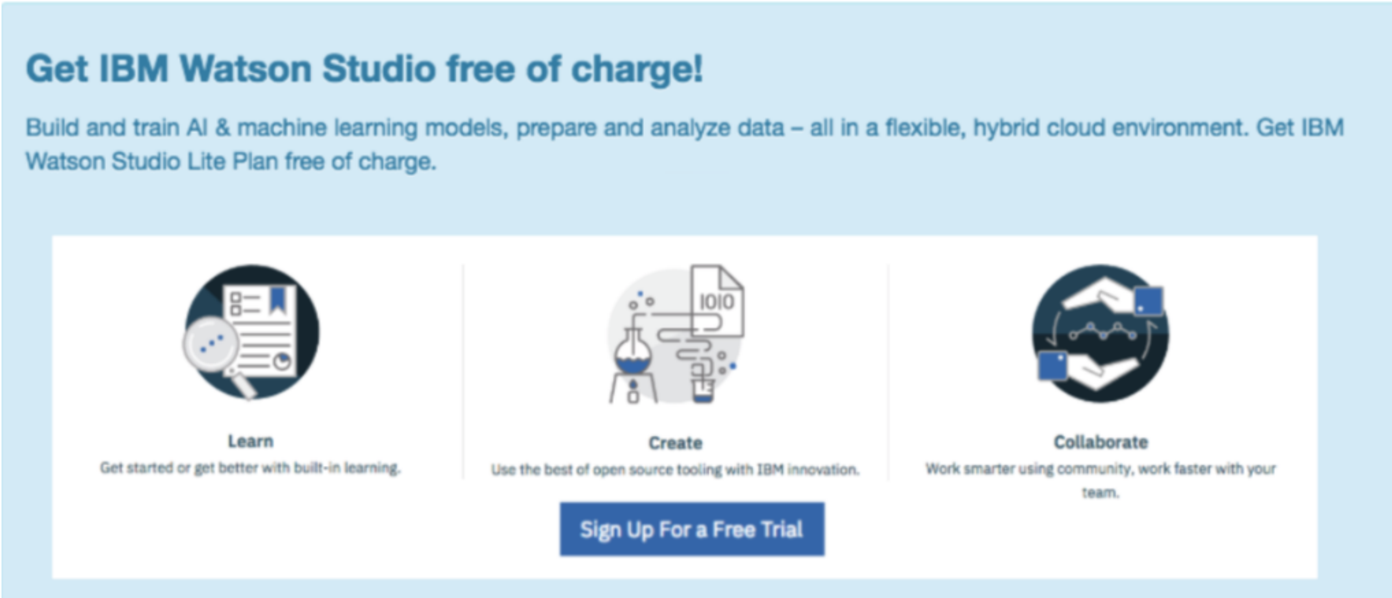
Build a model or object of type `linear_regression` . Using the `linear_regression` object will predict the following tensor:

In [17]:

```
# Practice: Build a model to predict the follow tensor.

X = torch.tensor([[11.0, 12.0, 13, 14], [11, 12, 13, 14]])
model = linear_regression(4, 1)
yhat = model(X)
print('The result: ', yhat)
```

```
The result:  tensor([[2.1062],
        [2.1062]], grad_fn=<AddmmBackward>)
```

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



(http://cocl.us/pytorch_link_bottom)

About the Authors:

Joseph Santarcangelo (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)
