```
# Import Numpy & PyTorch
import numpy as np
import torch
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
# Print tensors
print(x)
print(w)
print(b)
```

```
tensor(3.)
tensor(4., requires_grad=True)
tensor(5., requires_grad=True)
```

```
# Arithmetic operations
y = w * x + b
print(y)
```

```
tensor(17., grad_fn=<AddBackward0>)
```

```
# Compute gradients
y.backward()
```

```
# Display gradients
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

We'll create a model that predicts crop yeilds for apples and oranges (target variables) by looking at the average temperature, rainfall and humidity (input variables or features) in a region.

Region	Temp. (F)	Rainfall (mm)	Humidity (%)	Apples (ton)	Oranges (ton)
Kanto	73	67	43	56	70
Johto	91	88	64	81	101
Hoenn	87	134	58	119	133
Sinnoh	102	43	37	22	37
Unova	69	96	70	103	119

In a linear regression model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

yeild_apple = w11 * temp + w12 * rainfall + w13 * humidity + b1 yeild_orange = w21 * temp + w22 * rainfall + w23 * humidity + b2 Visually, it means that the yield of apples is a linear or planar function of the temperature, rainfall & humidity.

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')
```

```
# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')
```

```
# Convert inputs and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
print(inputs)
print(targets)
```

```
tensor([[ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 87., 134.,  58.],
        [102.,  43.,  37.],
        [ 69.,  96.,  70.]])
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

Linear Regression Model (from scratch) The weights and biases can also be represented as matrices, initialized with random values. The first row of w and the first element of b are use to predict the first target variable i.e. yield for apples, and similarly the second for oranges.

```
# Weights and biases
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
print(w)
print(b)
```

```
tensor([[ 1.6658,  0.5510,  0.9279],
        [-0.1205,  1.0328, -0.9826]], requires_grad=True)
tensor([-0.9630, -1.6259], requires_grad=True)
```

The model is simply a function that performs a matrix multiplication of the input x and the weights w (transposed) and adds the bias b (replicated for each observation).

```
# Define the model
def model(x):
    return x @ w.t() + b
```

```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[197.4580,  16.5227],
        [258.4995,  15.4079],
        [271.6161,  69.2943],
        [226.9740,  -5.8641],
        [231.8275,  20.4262]], grad_fn=<AddBackward0>)
```

```
# Compare with targets
print(targets)
```

```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

Because we've started with random weights and biases, the model does not a very good job of predicting the target variables.

Loss Function We can compare the predictions with the actual targets, using the following method:

Calculate the difference between the two matrices (preds and targets). Square all elements of the difference matrix to remove negative values. Calculate the average of the elements in the resulting matrix. The result is a single number, known as the mean squared error (MSE).

```
# MSE loss
def mse(t1, t2):
    diff = t1 - t2
    return torch.sum(diff * diff) / diff.numel()
```

```
# Compute loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(15921.7344, grad_fn=<DivBackward0>)
```

Compute Gradients With PyTorch, we can automatically compute the gradient or derivative of the loss w.r.t. to the weights and biases, because they have requires_grad set to True.

```
# Compute gradients
loss.backward()
```

```
# Gradients for weights
print(w)
print(w.grad)
```

```
tensor([[ 1.6658,  0.5510,  0.9279],
        [-0.1205,  1.0328, -0.9826]], requires_grad=True)
tensor([[13910.5889, 13345.9062,  8579.2725],
        [-5681.7695, -6191.5767, -3991.6965]])
```

```
# Gradients for bias
print(b)
print(b.grad)
```

```
tensor([-0.9630, -1.6259], requires_grad=True)
tensor([161.0750, -68.8426])
```

A key insight from calculus is that the gradient indicates the rate of change of the loss, or the slope of the loss function w.r.t. the weights and biases.

If a gradient element is positive, increasing the element's value slightly will increase the loss. decreasing the element's value slightly will decrease the loss. If a gradient element is negative, increasing the element's value slightly will decrease the loss. decreasing the element's value slightly will increase the loss. The increase or decrease is proportional to the value of the gradient.

Finally, we'll reset the gradients to zero before moving forward, because PyTorch accumulates gradients.

```
w.grad.zero_()
b.grad.zero_()
print(w.grad)
print(b.grad)
```

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

Adjust weights and biases using gradient descent We'll reduce the loss and improve our model using the gradient descent algorithm, which has the following steps:

1. Generate predictions
2. Calculate the loss
3. Compute gradients w.r.t the weights and biases
4. Adjust the weights by subtracting a small quantity proportional to the gradient
5. Reset the gradients to zero

```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[197.4580,  16.5227],
        [258.4995,  15.4079],
        [271.6161,  69.2943],
        [226.9740,  -5.8641],
        [231.8275,  20.4262]], grad_fn=<AddBackward0>)
```

```
# Calculate the loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(15921.7344, grad_fn=<DivBackward0>)
```

```
# Compute gradients
loss.backward()
```

```
# Adjust weights & reset gradients
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
    w.grad.zero_()
    b.grad.zero_()
```

```
print(w)
```

```
tensor([[ 1.5267,  0.4176,  0.8421],
        [-0.0637,  1.0947, -0.9427]], requires_grad=True)
```

With the new weights and biases, the model should have a lower loss.

```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
```

```
tensor(11078.8477, grad_fn=<DivBackward0>)
```

```
# Train for 100 epochs to further reduce the loss
for i in range(100):
    preds = model(inputs)
    loss = mse(preds, targets)
    loss.backward()
    with torch.no_grad():
        w -= w.grad * 1e-5
        b -= b.grad * 1e-5
        w.grad.zero_()
        b.grad.zero_()
```

```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
```

```
tensor(396.1728, grad_fn=<DivBackward0>)
```

```
# Print predictions
preds
```

```
tensor([[ 64.2540,  71.8969],
        [ 87.0418,  89.7244],
        [ 96.2975, 155.2948],
        [ 63.2366,  46.6102],
        [ 85.5389,  94.1767]], grad_fn=<AddBackward0>)
```

```
# Print targets
targets
```

```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```