Implementation of Linear regression from Sctrach using pyTorch

Linear regression performs a regression task on a target variable based on independent variables in a given data. It is a machine learning algorithm and is often used to find the relationship between the target and independent variables.

The **Simple Linear Regression** model is to predict the target variable using one independent variable.

When one variable/column in a dataset is not sufficient to create a good model and make more accurate predictions, we'll use a multiple linear regression model instead of a simple linear regression model

The line equation for the multiple linear regression model is: `y = w1x1 + w2x2 + ... + wnxn + b`

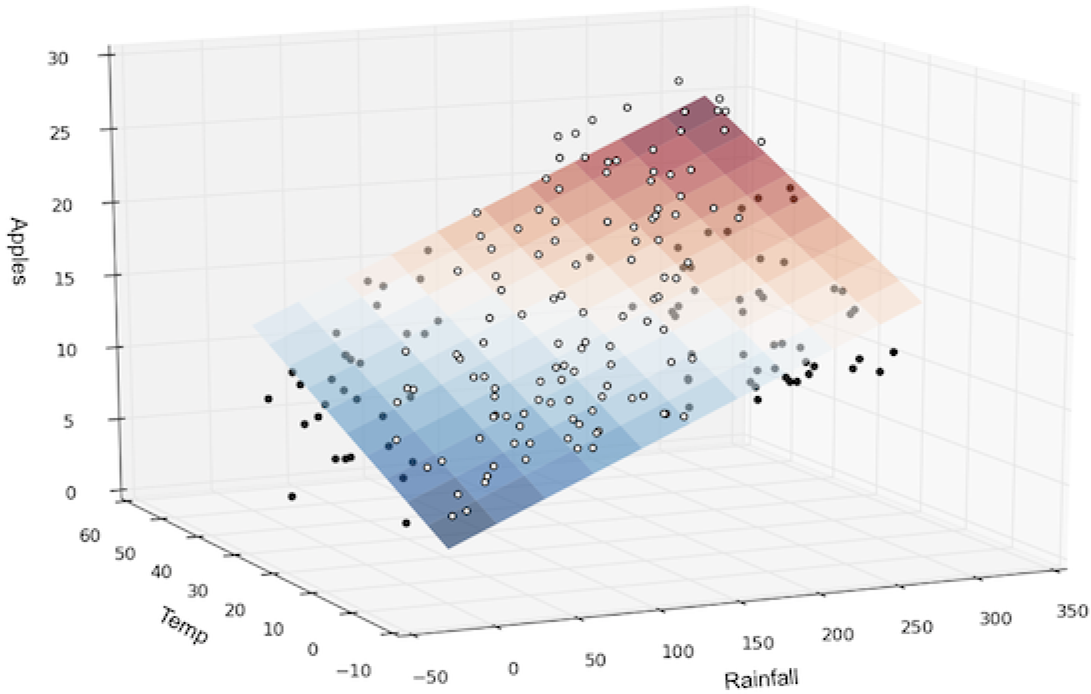
We'll create a model that predicts crop yields for apples and oranges (*target variables*) by looking at the average temperature, rainfall, and humidity (*input variables or features*) in a region. Here's the training data:

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 :

```
yield_apple  = w11 * temp + w12 * rainfall + w13 * humidity + b1
yield_orange = w21 * temp + w22 * rainfall + w23 * humidity + b2
```

Visually, it means that the yield of apples is a linear or planar function of temperature, rainfall and humidity:



In [1]:
```
!pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.or
```

```
Looking in indexes: https://pypi.org/simple, https://download.pytorch.org/whl/cu116
Requirement already satisfied: torch in c:\users\mihir chauhan\anaconda3\lib\site-packag
es (1.13.1)
Requirement already satisfied: torchvision in c:\users\mihir chauhan\anaconda3\lib\site-
packages (0.14.1+cu116)
Requirement already satisfied: torchaudio in c:\users\mihir chauhan\anaconda3\lib\site-p
ackages (0.13.1+cu116)
Requirement already satisfied: typing-extensions in c:\users\mihir chauhan\anaconda3\lib
\site-packages (from torch) (4.1.1)
Requirement already satisfied: requests in c:\users\mihir chauhan\anaconda3\lib\site-pac
kages (from torchvision) (2.27.1)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\mihir chauhan\anaconda3
\lib\site-packages (from torchvision) (9.0.1)
Requirement already satisfied: numpy in c:\users\mihir chauhan\anaconda3\lib\site-packag
es (from torchvision) (1.21.5)
Requirement already satisfied: idna<4,>=2.5 in c:\users\mihir chauhan\anaconda3\lib\site
-packages (from requests->torchvision) (3.3)
Requirement already satisfied: charset-normalizer~=2.0.0 in c:\users\mihir chauhan\anaco
nda3\lib\site-packages (from requests->torchvision) (2.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\mihir chauhan\anaconda3\li
b\site-packages (from requests->torchvision) (2021.10.8)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in c:\users\mihir chauhan\anaconda3
\lib\site-packages (from requests->torchvision) (1.26.9)
```

In [2]:
```
import torch
import numpy as np
```

Training Data

In [3]:
```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
```

```
                   [102, 43, 37],
                   [69, 96, 70]], dtype='float32')
```

In [4]:
```python
# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')
```

In [5]:
```python
# 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.]])
```

Weight and baises

The weights and biases (w11, w12,... w23, b1 & b2) can also be represented as matrices, initialized as random values. The first row of w and the first element of b are used to predict the first target variable, i.e., yield of apples, and similarly, the second for oranges.

In [6]:
```python
w = torch.randn(2,3, requires_grad= True)
b = torch.randn(2, requires_grad= True)
print(w,'\n',b)
```

```
tensor([[-0.5002, -0.1654, -0.7574],
        [-0.7656, -0.6830,  0.0349]], requires_grad=True)
 tensor([-0.6475, -1.0123], requires_grad=True)
```

`torch.randn` creates a tensor with the given shape, with elements picked randomly from a normal distribution with mean 0 and standard deviation 1.

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

$$X \qquad \times \qquad W^T \qquad + \qquad b$$

$$\begin{bmatrix} 73 & 67 & 43 \\ 91 & 88 & 64 \\ \vdots & \vdots & \vdots \\ 69 & 96 & 70 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{21} \\ w_{12} & w_{22} \\ w_{13} & w_{23} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 \\ b_1 & b_2 \\ \vdots & \vdots \\ b_1 & b_2 \end{bmatrix}$$

In [7]:
```
def model(x):
    return x @ w.t() + b
```

`@` represents matrix multiplication in PyTorch, and the `.t` method returns the transpose of a tensor.

The matrix obtained by passing the input data into the model is a set of predictions for the target variables.

In [8]:
```
preds = model(inputs)
```

In [9]:
```
preds
```

Out[9]:
```
tensor([[ -80.8109, -101.1633],
        [-109.1931, -128.5549],
        [-110.2558, -157.1216],
        [ -86.8024, -107.1817],
        [-104.0567, -116.9670]], grad_fn=<AddBackward0>)
```

In [10]:
```
# compare preds with targets
print(targets)
```

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

Loss function

Before we improve our model, we need a way to evaluate how well our model is performing. We can compare the model's 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).

In [11]:
```
# mean squared error loss
def mse(t1,t2):
    diff = t1-t2
    return torch.sum(diff*diff)/diff.numel()
```

`torch.sum` returns the sum of all the elements in a tensor. The `.numel` method of a tensor returns the number of elements in a tensor. Let's compute the mean squared error for the current predictions of our model.

In [12]:
```
loss = mse(preds,targets)
```

In [13]:
```
print(loss)
```

```
tensor(40479.0977, grad_fn=<DivBackward0>)
```

Here's how we can interpret the result: *On average, each element in the prediction differs from the actual target by the square root of the loss*. And that's pretty bad, considering the numbers we are trying to predict are themselves in the range 50–200. The result is called the *loss* because it indicates how bad the model is at predicting the target variables. It represents information loss in the model: the lower the loss, the better the model.

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`. We'll see how this is useful in just a moment.

In [14]:
```
loss.backward()
```

In [15]:
```
# Gradients for weights
print(w)
print(w.grad)
```

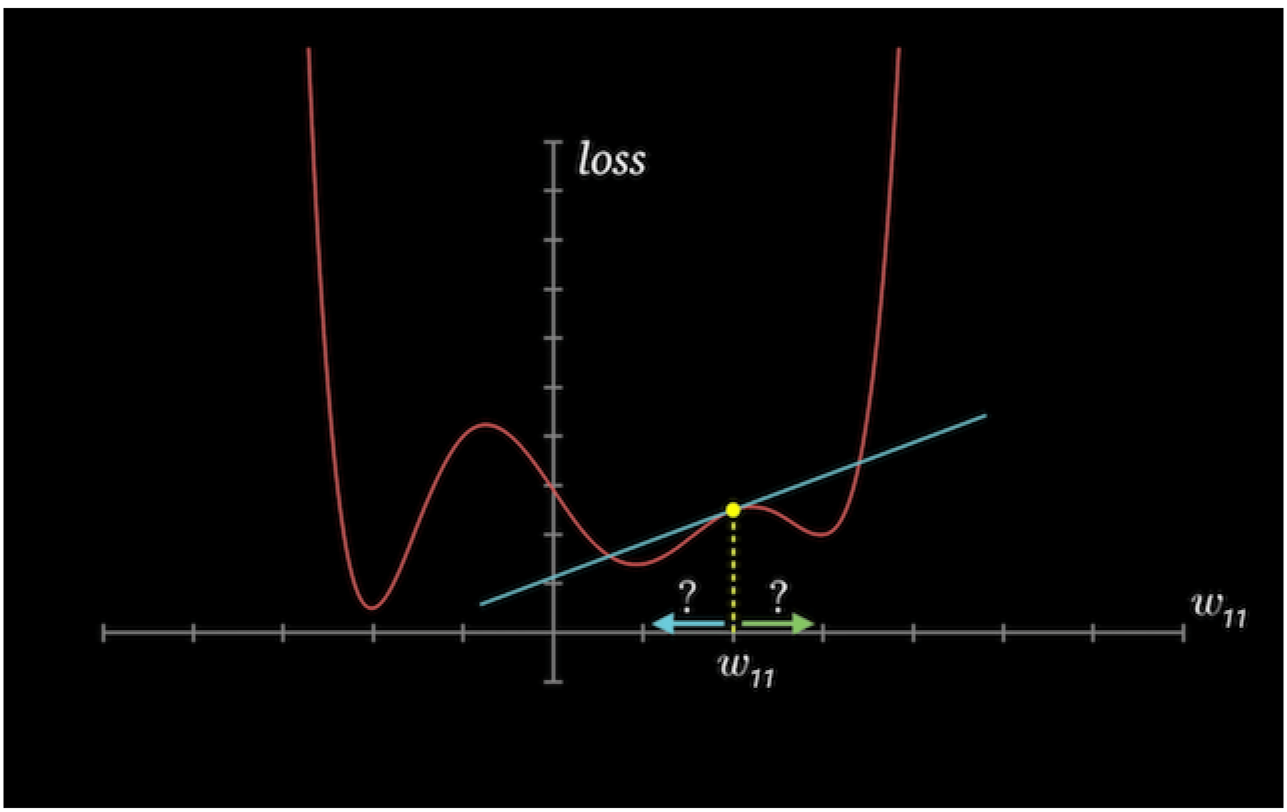
```
tensor([[-0.5002, -0.1654, -0.7574],
        [-0.7656, -0.6830,  0.0349]], requires_grad=True)
tensor([[-14524.9570, -16235.9102,  -9974.3447],
        [-17922.6523, -19879.5430, -12146.2012]])
```

Adjust weights and biases to reduce the loss

The loss is a quadratic function of our weights and biases, and our objective is to find the set of weights where the loss is the lowest. If we plot a graph of the loss w.r.t any individual weight or bias element, it will look like the figure shown below. An important insight from calculus is that the gradient indicates the rate of change of the loss, i.e., the loss function's slope w.r.t. the weights and biases.

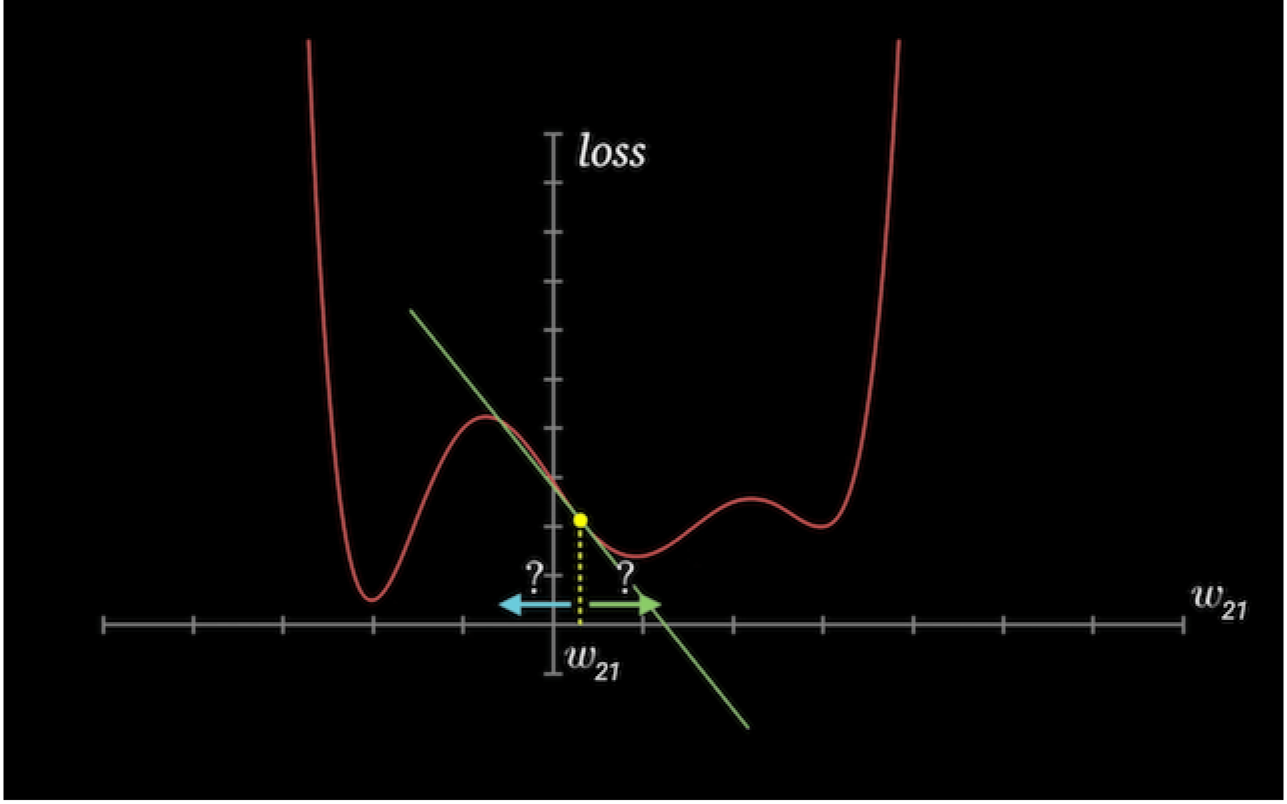
If a gradient element is **positive**:

- **increasing** the weight element's value slightly will **increase** the loss
- **decreasing** the weight element's value slightly will **decrease** the loss



If a gradient element is **negative**:

- **increasing** the weight element's value slightly will **decrease** the loss
- **decreasing** the weight element's value slightly will **increase** the loss



The increase or decrease in the loss by changing a weight element is proportional to the gradient of the loss w.r.t. that element. This observation forms the basis of *the gradient descent* optimization algorithm that we'll use to improve our model (by *descending* along the *gradient*).

We can subtract from each weight element a small quantity proportional to the derivative of the loss w.r.t. that element to reduce the loss slightly.

In [16]:
```
w
w.grad
```

Out[16]:
```
tensor([[-14524.9570, -16235.9102,  -9974.3447],
        [-17922.6523, -19879.5430, -12146.2012]])
```

We multiply the gradients with a very small number (`10^-5` in this case) to ensure that we don't modify the weights by a very large amount. We want to take a small step in the downhill direction of the gradient, not a giant leap. This number is called the *learning rate* of the algorithm.

We use `torch.no_grad` to indicate to PyTorch that we shouldn't track, calculate, or modify gradients while updating the weights and biases.

In [17]:
```
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
```

In [18]:
```
# Let's verify that the loss is actually lower
loss = mse(preds, targets)
print(loss)
```

```
tensor(40479.0977, grad_fn=<DivBackward0>)
```

Before we proceed, we reset the gradients to zero by invoking the `.zero_()` method. We need to do this because PyTorch accumulates gradients. Otherwise, the next time we invoke `.backward` on the loss, the new gradient values are added to the existing gradients, which may lead to unexpected results.

In [19]:
```
w.grad.zero_()
b.grad.zero_()
print(w.grad,'\n',b.grad)
```

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

Train the model using gradient descent

As seen above, we reduce the loss and improve our model using the gradient descent optimization algorithm. Thus, we can *train* the model using 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

Let's implement the above step by step.

In [20]:
```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[ -55.0389,  -69.5355],
        [ -75.3025,  -86.9756],
        [ -70.0761, -107.8433],
        [ -61.3133,  -75.8562],
        [ -71.4642,  -77.0115]], grad_fn=<AddBackward0>)
```

In [21]:
```
# Calculate the loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(27385.6289, grad_fn=<DivBackward0>)
```

In [22]:
```
# Compute gradients
loss.backward()
print(w.grad)
print(b.grad)
```

```
tensor([[-11862.9951, -13372.2910,  -8207.9062],
        [-14656.2734, -16366.7324,  -9979.1738]])
tensor([-142.8390, -175.4444])
```

In [23]:
```
# Adjust weights & reset gradients
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
    w.grad.zero_()
    b.grad.zero_()
```

In [24]:
```
print(w)
print(b)
```

```
tensor([[-0.2363,  0.1307, -0.5756],
        [-0.4398, -0.3206,  0.2561]], requires_grad=True)
tensor([-0.6444, -1.0084], requires_grad=True)
```

In [25]:
```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
```

```
tensor(18560.9160, grad_fn=<DivBackward0>)
```

Train for multiple epochs

To reduce the loss further, we can repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an *epoch*. Let's train the model for 100 epochs.

In [26]:
```
# Train for 100 epochs
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_()
```

In [27]:
```
# Calculate loss
preds = model(inputs)
loss = mse(preds, targets)
print(loss)
```

```
tensor(116.7225, grad_fn=<DivBackward0>)
```

In [28]:
```
# Predictions
preds
```

Out[28]:
```
tensor([[ 60.6085,  73.0626],
        [ 78.7150, 101.8833],
        [121.0971, 125.7696],
        [ 41.1303,  53.9283],
        [ 84.0197, 111.1808]], grad_fn=<AddBackward0>)
```

In [29]:
```
# Targets
targets
```

Out[29]:
```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

Linear regression using PyTorch built-ins

We've implemented linear regression & gradient descent model using some basic tensor operations. However, since this is a common pattern in deep learning, PyTorch provides several built-in functions and classes to make it easy to create and train models with just a few lines of code.

Let's begin by importing the `torch.nn` package from PyTorch, which contains utility classes for building neural networks.

In [30]:
```
from torch import nn
```

In [31]:
```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43],
                   [91, 88, 64],
                   [87, 134, 58],
                   [102, 43, 37],
                   [69, 96, 70],
                   [74, 66, 43],
                   [91, 87, 65],
                   [88, 134, 59],
                   [101, 44, 37],
                   [68, 96, 71],
                   [73, 66, 44],
                   [92, 87, 64],
                   [87, 135, 57],
                   [103, 43, 36],
                   [68, 97, 70]],
                  dtype='float32')

# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119],
                    [57, 69],
                    [80, 102],
                    [118, 132],
                    [21, 38],
                    [104, 118],
                    [57, 69],
                    [82, 100],
                    [118, 134],
                    [20, 38],
                    [102, 120]],
                   dtype='float32')

inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
```

Dataset and DataLoader

We'll create a `TensorDataset`, which allows access to rows from `inputs` and `targets` as tuples, and provides standard APIs for working with many different types of datasets in PyTorch.

In [32]:
```
from torch.utils.data import TensorDataset
```

In [33]:
```
train_ds = TensorDataset(inputs,targets)
```

In [34]:
```
train_ds[0:3]
```

Out[34]:
```
(tensor([[ 73.,  67.,  43.],
         [ 91.,  88.,  64.],
         [ 87., 134.,  58.]]),
 tensor([[ 56.,  70.],
         [ 81., 101.],
         [119., 133.]]))
```

The `TensorDataset` allows us to access a small section of the training data using the array indexing notation (`[0:3]` in the above code). It returns a tuple with two elements. The first element contains the input variables for the selected rows, and the second contains the targets.

We'll also create a `DataLoader`, which can split the data into batches of a predefined size while training. It also provides other utilities like shuffling and random sampling of the data.

In [35]:
```
from torch.utils.data import DataLoader
```

In [36]:
```
# Define a DataLoader
batch_size = 5
train_dl = DataLoader(train_ds,batch_size,shuffle= True)
```

We can use the data loader in a `for` loop. Let's look at an example.

In [37]:
```
for xb, yb in train_dl:
    print(xb)
    print(yb)
    break
```

```
tensor([[ 73.,  67.,  43.],
        [ 91.,  87.,  65.],
        [ 92.,  87.,  64.],
        [102.,  43.,  37.],
        [ 88., 134.,  59.]])
tensor([[ 56.,  70.],
        [ 80., 102.],
        [ 82., 100.],
        [ 22.,  37.],
        [118., 132.]])
```

In each iteration, the data loader returns one batch of data with the given batch size. If `shuffle` is set to `True`, it shuffles the training data before creating batches. Shuffling helps randomize the input to the optimization algorithm, leading to a faster reduction in the loss.

nn.Linear

Instead of initializing the weights & biases manually, we can define the model using the `nn.Linear` class from PyTorch, which does it automatically.

In [38]:
```
model = nn.Linear(3,2)
print(model.weight,'\n',model.bias)
```

```
Parameter containing:
tensor([[ 0.4834, -0.4376, -0.2987],
        [-0.5542,  0.4785,  0.0545]], requires_grad=True)
 Parameter containing:
tensor([-0.2166,  0.4393], requires_grad=True)
```

PyTorch models also have a helpful `.parameters` method, which returns a list containing all the weights and bias matrices present in the model. For our linear regression model, we have one weight matrix and one bias matrix.

In [39]:
```
# Parameters
list(model.parameters())
```

Out[39]:
```
[Parameter containing:
 tensor([[ 0.4834, -0.4376, -0.2987],
         [-0.5542,  0.4785,  0.0545]], requires_grad=True),
 Parameter containing:
 tensor([-0.2166,  0.4393], requires_grad=True)]
```

In [40]:
```
# Generate predictions
```

```
preds = model(inputs)
preds
```

Out[40]:

```
tensor([[ -7.0964,  -5.6083],
        [-13.8589,  -4.3890],
        [-34.1321,  19.5127],
        [ 19.2175, -33.4907],
        [-29.7868,  11.9579],
        [ -6.1754,  -6.6410],
        [-13.7200,  -4.8130],
        [-33.9474,  19.0131],
        [ 18.2965, -32.4580],
        [-30.5689,  12.5666],
        [ -6.9575,  -6.0323],
        [-12.9379,  -5.4217],
        [-34.2710,  19.9367],
        [ 19.9996, -34.0994],
        [-30.7078,  12.9906]], grad_fn=<AddmmBackward0>)
```

Loss Function

Instead of defining a loss function manually, we can use the built-in loss function `mse_loss`.

In [41]:

```
# importing nn.Functional
import torch.nn.functional as F
```

The `nn.functional` package contains many useful loss functions and several other utilities.

In [42]:

```
# Define a loss function
loss_fn = F.mse_loss
```

In [43]:

```
loss = loss_fn(model(inputs),targets)
loss
```

Out[43]:

```
tensor(10015.3135, grad_fn=<MseLossBackward0>)
```

Optimizer

Instead of manually manipulating the model's weights & biases using gradients, we can use the optimizer `optim.SGD`. SGD is short for "stochastic gradient descent". The term *stochastic* indicates that samples are selected in random batches instead of as a single group.

In [44]:

```
# Define Optimizer
opt = torch.optim.SGD(model.parameters(),lr= 1e-5)
```

Note that `model.parameters()` is passed as an argument to `optim.SGD` so that the optimizer knows which matrices should be modified during the update step. Also, we can specify a learning rate that controls the amount by which the parameters are modified.

Train the model

We are now ready to train the model. We'll follow the same process to implement gradient descent:

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

The only change is that we'll work batches of data instead of processing the entire training data in every iteration. Let's define a utility function `fit` that trains the model for a given number of epochs.

In [45]:
```
# Utility  function to train model
def fit(model, loss_fn, num_epochs, opt, train_dl):
    # repeat for given no. of epoch
    for epoch in range(num_epochs):
        # train with batches
        for xb,yb in train_dl:
            # 1. Generate prediction
            preds = model(xb)
            # 2. calculate loss
            loss = loss_fn(preds,yb)
            # 3. compute gradients
            loss.backward()
            # 4. update the paramete using gradient
            opt.step()
            # 5. Reset the gradients to zero
            opt.zero_grad()

        # Print the progress
        if (epoch+1) % 10 == 0:
            print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, num_epochs, loss.item()))
```

Some things to note above:

- We use the data loader defined earlier to get batches of data for every iteration.
- Instead of updating parameters (weights and biases) manually, we use `opt.step` to perform the update and `opt.zero_grad` to reset the gradients to zero.
- We've also added a log statement that prints the loss from the last batch of data for every 10th epoch to track training progress. `loss.item` returns the actual value stored in the loss tensor.

Let's train the model for 100 epochs.

In [46]:
```
fit(model,loss_fn,100,opt,train_dl)
```

```
Epoch [10/100], Loss: 695.2487
Epoch [20/100], Loss: 646.9416
Epoch [30/100], Loss: 183.1446
Epoch [40/100], Loss: 200.3592
Epoch [50/100], Loss: 31.3893
Epoch [60/100], Loss: 51.1295
Epoch [70/100], Loss: 116.3545
Epoch [80/100], Loss: 57.9499
Epoch [90/100], Loss: 50.5161
Epoch [100/100], Loss: 19.7801
```

In [47]:
```
preds = model(inputs)
```

```
preds
```

In [48]:

Out[48]:
```
tensor([[ 59.1774,  70.9450],
        [ 80.3733,  97.6725],
        [119.1597, 138.8373],
        [ 33.0392,  39.8691],
        [ 91.7433, 112.3391],
        [ 58.1802,  69.8128],
        [ 79.7653,  97.2224],
        [119.2849, 139.1403],
        [ 34.0365,  41.0013],
        [ 92.1325, 113.0212],
        [ 58.5694,  70.4949],
        [ 79.3760,  96.5403],
        [119.7677, 139.2874],
        [ 32.6499,  39.1870],
        [ 92.7405, 113.4713]], grad_fn=<AddmmBackward0>)
```

In [49]:
```
targets
```

Out[49]:
```
tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.],
        [ 57.,  69.],
        [ 80., 102.],
        [118., 132.],
        [ 21.,  38.],
        [104., 118.],
        [ 57.,  69.],
        [ 82., 100.],
        [118., 134.],
        [ 20.,  38.],
        [102., 120.]])
```

Indeed, the predictions are quite close to our targets. We have a trained a reasonably good model to predict crop yields for apples and oranges by looking at the average temperature, rainfall, and humidity in a region. We can use it to make predictions of crop yields for new regions by passing a batch containing a single row of input.

In [50]:
```
model(torch.tensor([[75, 63, 44.]]))
```

Out[50]:
```
tensor([[55.7096, 67.2879]], grad_fn=<AddmmBackward0>)
```

The predicted yield of apples is 54.3 tons per hectare, and that of oranges is 68.3 tons per hectare.

In [ ]: