# Tutorial 6 - Neural Network

## Part A: linear regression on synthetic dataset

Oct, 2023

Yuan Zhong

Official website: https://pytorch.org/get-started/locally/

In [1]:
```
# Import Numpy & PyTorch
import numpy as np
import torch
```

A tensor is a number, vector, matrix or any n-dimensional array.

In [2]:
```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
```

In [3]:
```
# Print tensors
print(x)
print(w)
print(b)
```

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

We can combine tensors with the usual arithmetic operations.

In [4]:
```
# Arithmetic operations
y = w * x + b
print(y)
```

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

What makes PyTorch special, is that we can automatically compute the derivative of `y` w.r.t. the tensors that have `requires_grad` set to `True` i.e. `w` and `b`.

In [5]:
```
# Compute gradients
```

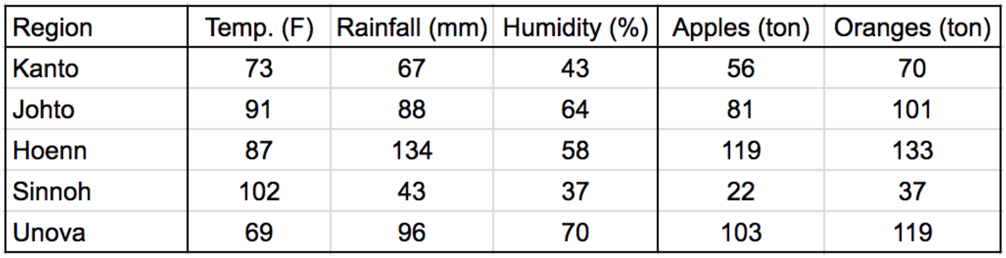
```
y.backward()
```

In [6]:
```
# Display gradients
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

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

## Problem Statement

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. Here's the training data:

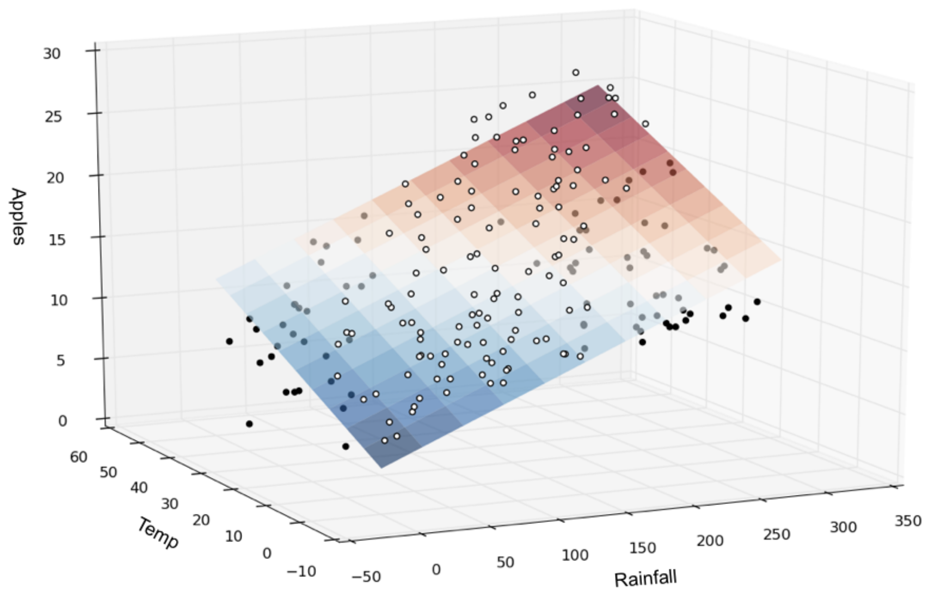


| Region | Temp. (F) | Rainfall (mm) | Humidity (%) | Apples (ton) | Oranges (ton) |
|---|---|---|---|---|---|
| Kanto | 73 | 67 | 43 | 56 | 70 |
| Johto | 91 | 88 | 64 | 81 | 101 |
| Hoenn | 87 | 134 | 58 | 119 | 133 |
| Sinnoh | 102 | 43 | 37 | 22 | 37 |
| Unova | 69 | 96 | 70 | 103 | 119 |

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 :

```
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.



**Our objective**: Find a suitable set of *weights* and *biases* using the training data, to make accurate predictions.

## Training Data

The training data can be represented using 2 matrices (inputs and targets), each with one row per observation and one column per variable.

In [7]:
```
# 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 [8]:
```
# Targets (apples, oranges)
targets = np.array([[56, 70],
                    [81, 101],
                    [119, 133],
                    [22, 37],
                    [103, 119]], dtype='float32')
```

Before we build a model, we need to convert inputs and targets to PyTorch tensors.

In [9]:
```
# 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.

In [10]:
```
# Weights and biases
w = torch.randn(2, 3, requires_grad=True)
b = torch.randn(2, requires_grad=True)
print(w)
print(b)
```

```
tensor([[ 0.5896, -0.5372,  0.7526],
        [ 0.6566,  0.9928,  0.6209]], requires_grad=True)
tensor([1.3376, 1.0164], 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).

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

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

In [11]:
```
# Define the model
def model(x):
    return x @ w.t() + b
```

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

In [12]:
```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[ 40.7492, 142.1647],
        [ 55.8860, 187.8710],
        [ 24.3021, 227.1887],
        [ 66.2237, 133.6535],
        [ 43.1334, 185.0933]], grad_fn=<AddBackward0>)
```

In [13]:
```
# 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 varaibles.

## 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).

In [14]:
```
# MSE loss
def mse(t1, t2):
    diff = t1 - t2
    return torch.sum(diff * diff) / diff.numel()
```

In [15]:
```
# Compute loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(5070.6777, grad_fn=<DivBackward0>)
```

The resulting number is called the **loss**, because it indicates how bad the model is at predicting the target variables. Lower the loss, 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` .

In [16]:
```
# Compute gradients
loss.backward()
```

The gradients are stored in the `.grad` property of the respective tensors.

```
In [17]: # Gradients for weights
         print(w)
         print(w.grad)
```

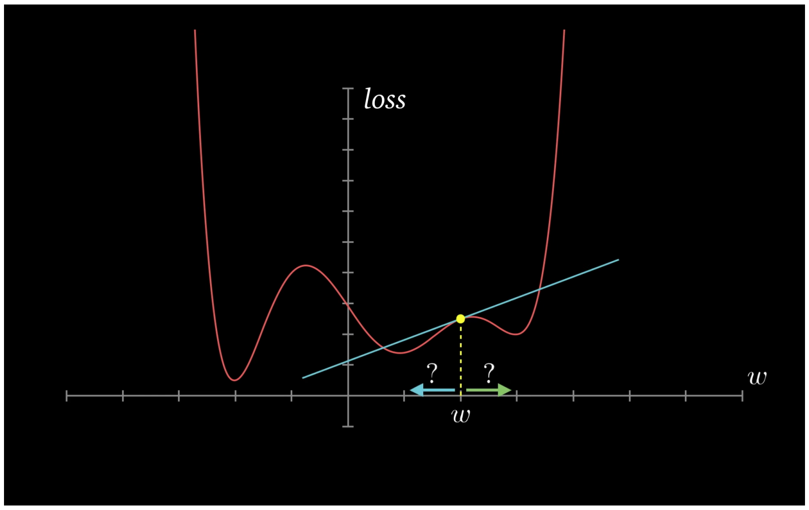
```
tensor([[ 0.5896, -0.5372,  0.7526],
        [ 0.6566,  0.9928,  0.6209]], requires_grad=True)
tensor([[-2251.4731, -3953.3833, -2061.9873],
        [ 7157.3589,  7120.4053,  4465.6963]])
```

```
In [18]: # Gradients for bias
         print(b)
         print(b.grad)
```

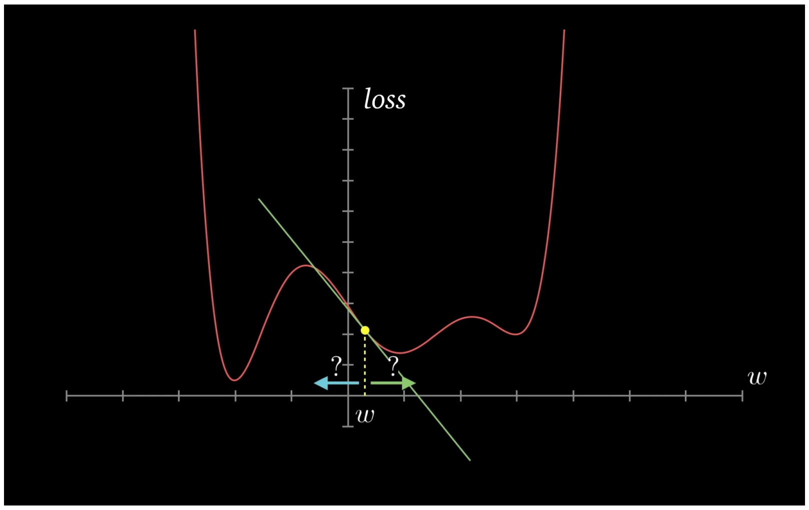
```
tensor([1.3376, 1.0164], requires_grad=True)
tensor([-30.1411,  83.1942])
```

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 **postive**,
  - **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.

In [19]:
```
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

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

```
tensor([[ 40.7492, 142.1647],
        [ 55.8860, 187.8710],
        [ 24.3021, 227.1887],
        [ 66.2237, 133.6535],
        [ 43.1334, 185.0933]], grad_fn=<AddBackward0>)
```

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

```
print(loss)
```

```
tensor(5070.6777, grad_fn=<DivBackward0>)
```

In [22]:
```
# Compute gradients
loss.backward()
```

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)
```

```
tensor([[ 0.6121, -0.4976,  0.7732],
        [ 0.5850,  0.9216,  0.5762]], requires_grad=True)
```

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

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

```
tensor(3732.7915, grad_fn=<DivBackward0>)
```

## Train for multiple epochs

To reduce the loss further, we repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an epoch.

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(289.7875, grad_fn=<DivBackward0>)
```

In [28]:
```
# Print predictions
preds
```

Out[28]:
```
tensor([[ 63.9860,  73.4480],
        [ 89.5102,  99.2098],
        [ 91.1217, 131.2367],
        [ 59.6985,  53.9162],
        [ 92.3537, 106.9338]], grad_fn=<AddBackward0>)
```

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

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

## Linear Regression Model using PyTorch built-ins

Let's re-implement the same model using some built-in functions and classes from PyTorch.

In [30]:
```
# Imports
import torch.nn as nn
```

In [31]:
```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43], [91, 88, 64], [87, 134, 58], [102, 43, 37],
# Targets (apples, oranges)
targets = np.array([[56, 70], [81, 101], [119, 133], [22, 37], [103, 119],
                    [56, 70], [81, 101], [119, 133], [22, 37], [103, 119],
                    [56, 70], [81, 101], [119, 133], [22, 37], [103, 119]],
```

In [32]:
```
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. We'll also create a DataLoader, to split the data into batches while training. It also provides other utilities like shuffling and sampling.

In [33]:
```
# Import tensor dataset & data loader
from torch.utils.data import TensorDataset, DataLoader
```

In [34]:
```
# Define dataset
train_ds = TensorDataset(inputs, targets)
train_ds[0:3]
```

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

```
In [35]: # Define data loader
         batch_size = 5
         train_dl = DataLoader(train_ds, batch_size, shuffle=True)
         next(iter(train_dl))
```

```
Out[35]: [tensor([[ 69.,  96.,  70.],
                  [102.,  43.,  37.],
                  [ 91.,  88.,  64.],
                  [ 73.,  67.,  43.],
                  [ 87., 134.,  58.]]),
          tensor([[103., 119.],
                  [ 22.,  37.],
                  [ 81., 101.],
                  [ 56.,  70.],
                  [119., 133.]])]
```

## nn.Linear

Instead of initializing the weights & biases manually, we can define the model using `nn.Linear`.

```
In [36]: # Define model
         model = nn.Linear(3, 2)
         print(model.weight)
         print(model.bias)
```

```
Parameter containing:
tensor([[-0.2792,  0.4736,  0.1632],
        [ 0.2605, -0.1820, -0.0282]], requires_grad=True)
Parameter containing:
tensor([-0.1752,  0.1164], requires_grad=True)
```

## Optimizer

Instead of manually manipulating the weights & biases using gradients, we can use the optimizer `optim.SGD`.

```
In [37]: # Define optimizer
         opt = torch.optim.SGD(model.parameters(), lr=1e-5)
```

## Loss Function

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

In [38]:
```
# Import nn.functional
import torch.nn.functional as F
```

In [39]:
```
# Define loss function
loss_fn = F.mse_loss
```

In [40]:
```
loss = loss_fn(model(inputs), targets)
print(loss)
```

```
tensor(6082.8408, grad_fn=<MseLossBackward0>)
```

## Train the model

We are ready to train the model now. We can define a utility function `fit` which trains the model for a given number of epochs.

In [41]:
```
# Define a utility function to train the model
def fit(num_epochs, model, loss_fn, opt):
    for epoch in range(num_epochs):
        for xb,yb in train_dl:
            # Generate predictions
            pred = model(xb)
            loss = loss_fn(pred, yb)
            # Perform gradient descent
            loss.backward()
            opt.step()
            opt.zero_grad()
    print('Training loss: ', loss_fn(model(inputs), targets))
```

In [42]:
```
# Train the model for 100 epochs
fit(100, model, loss_fn, opt)
```

```
Training loss:  tensor(24.2029, grad_fn=<MseLossBackward0>)
```

In [43]:
```
# Generate predictions
preds = model(inputs)
preds
```

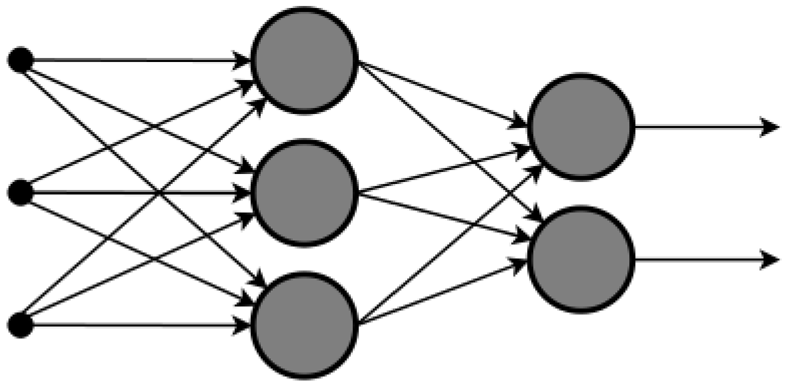
Out[43]:
```
tensor([[ 57.8584,  72.1569],
        [ 80.7428,  98.6317],
        [121.2946, 135.4737],
        [ 24.7784,  46.6363],
        [ 97.1537, 109.9387],
        [ 57.8584,  72.1569],
        [ 80.7428,  98.6317],
        [121.2946, 135.4737],
        [ 24.7784,  46.6363],
        [ 97.1537, 109.9387],
        [ 57.8584,  72.1569],
        [ 80.7428,  98.6317],
        [121.2946, 135.4737],
        [ 24.7784,  46.6363],
        [ 97.1537, 109.9387]], grad_fn=<AddmmBackward0>)
```

In [44]:
```
# Compare with targets
```

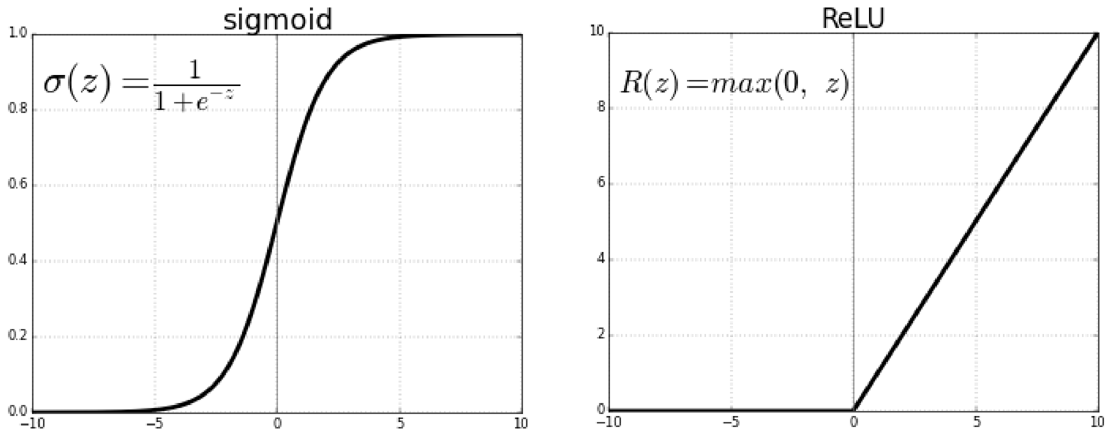
```
targets
```

```
Out[44]: tensor([[ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.],
        [ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.],
        [ 56.,  70.],
        [ 81., 101.],
        [119., 133.],
        [ 22.,  37.],
        [103., 119.]])
```

## Bonus: Feedfoward Neural Network



Conceptually, you think of feedforward neural networks as two or more linear regression models stacked on top of one another with a non-linear activation function applied between them.



To use a feedforward neural network instead of linear regression, we can extend the `nn.Module` class from PyTorch.

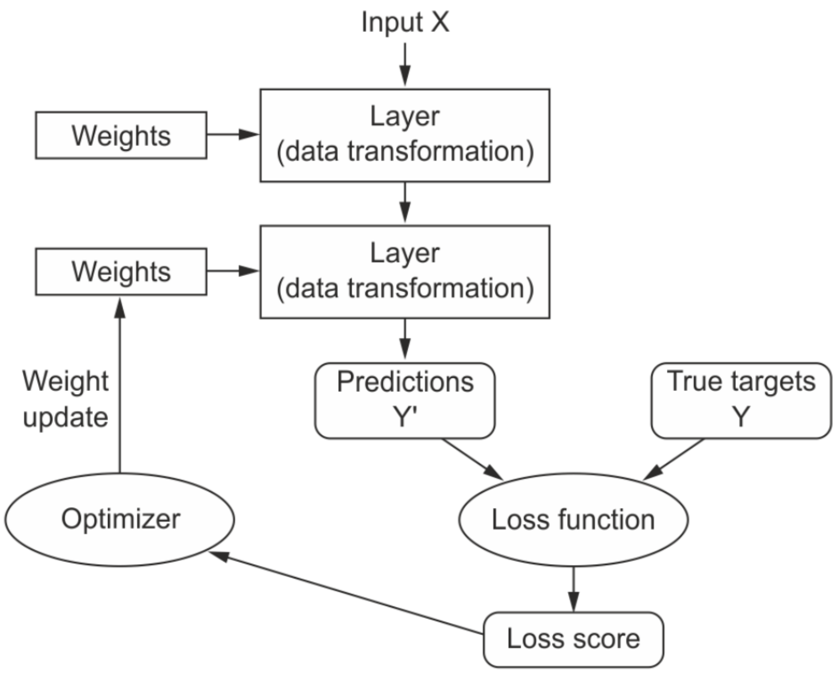
In [45]:
```
class SimpleNet(nn.Module):
    # Initialize the layers
    def __init__(self):
        super().__init__()
        self.linear1 = nn.Linear(3, 3)
        self.act1 = nn.ReLU() # Activation function
        self.linear2 = nn.Linear(3, 2)

    # Perform the computation
    def forward(self, x):
        x = self.linear1(x)
        x = self.act1(x)
        x = self.linear2(x)
        return x
```

Now we can define the model, optimizer and loss function exactly as before.

In [46]:
```
model = SimpleNet()
opt = torch.optim.SGD(model.parameters(), 1e-5)
loss_fn = F.mse_loss
```

Finally, we can apply gradient descent to train the model using the same `fit` function defined earlier for linear regression.



In [47]:
```
fit(100, model, loss_fn, opt)
```

```
Training loss:  tensor(5.9847, grad_fn=<MseLossBackward0>)
```