# Applied Deep Learning with Tensorflow and Pytorch, Chapter 3

## Linear Regression Implementation in PyTorch

- Regression refers to a set of methods for modeling the relationship between one or more independent variables and a dependent variable.
- Linear regression is a single-layer neural network.

### 1: Import libraries and Initialize the Model Parameters

```
import random
import torch

w = torch.normal(0, 0.01, size=(2, 1), requires_grad=True)
b = torch.zeros(1, requires_grad=True)
```

### 2: Defining the Model and the Loss Function

```
def linreg(X, w, b):
    return torch.matmul(X, w) + b

def squared_loss(y_hat, y):
    return (y_hat - y.reshape(y_hat.shape))**2 / 2
```

### 3: Defining the Optimization Algorithm

```
def sgd(params, lr, batch_size):
    with torch.no_grad():
        for param in params:
            param -= lr * param.grad / batch_size
            param.grad.zero_()
```

### 4: Ttraining

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        l = loss(net(X, w, b), y)
        l.sum().backward()
        sgd([w, b], lr, batch_size)
    with torch.no_grad():
        train_l = loss(net(features, w, b), labels)
        print(f'epoch {epoch + 1}, loss {float(train_l.mean()):f}')
```

## TensorFlow Implementation of Linear Regression

### 1: Import libraries and Initialize Model Parameters

```
import random
import tensorflow as tf

w = tf.Variable(tf.random.normal(shape=(2, 1), mean=0, stddev=0.01),
                trainable=True)
b = tf.Variable(tf.zeros(1), trainable=True)
```

### 2: Defining the Model and Loss Function

```
def linreg(X, w, b):
    return tf.matmul(X, w) + b

def linreg(X, w, b):
    return tf.matmul(X, w) + b
```

### 3: Defining the Optimization Algorithm

```
def sgd(params, grads, lr, batch_size):
    for param, grad in zip(params, grads):
        param.assign_sub(lr * grad / batch_size)
```

### 4: Ttraining

```
lr = 0.03
num_epochs = 3
net = linreg
loss = squared_loss

for epoch in range(num_epochs):
    for X, y in data_iter(batch_size, features, labels):
        with tf.GradientTape() as g:
            l = loss(net(X, w, b), y)
        dw, db = g.gradient(l, [w, b])
        sgd([w, b], [dw, db], lr, batch_size)
    train_l = loss(net(features, w, b), labels)
    print(f'epoch {epoch + 1}, loss {float(tf.reduce_mean(train_l)):f}')
```

- Using TensorFlow's high-level APIs, we can implement models much more concisely.
- In TensorFlow, the data module provides tools for data processing, the keras module defines a large number of neural network layers and common loss functions.
- TensorFlow's module initializers provides various methods for model parameter initialization.

## Concise Implementation of Linear Regression

### 1: Import libraries

```
import random
import tensorflow as tf
```

### 2: Defining the Model

```
net = tf.keras.Sequential()
net.add(tf.keras.layers.Dense(1))
```

### 3: Initializing Model Parameters

```
initializer = tf.initializers.RandomNormal(stddev=0.01)
net = tf.keras.Sequential()
net.add(tf.keras.layers.Dense(1, kernel_initializer=initializer))
```

### 4: Defining the Loss Function

```
loss = tf.keras.losses.MeanSquaredError()
```

### 5: Defining the Optimization Algorithm

```
trainer = tf.keras.optimizers.SGD(learning_rate=0.03)
```

### 6: Ttraining

```
num_epochs = 3
for epoch in range(num_epochs):
    for X, y in data_iter:
        with tf.GradientTape() as tape:
            l = loss(net(X, training=True), y)
        grads = tape.gradient(l, net.trainable_variables)
        trainer.apply_gradients(zip(grads, net.trainable_variables))
    l = loss(net(features), labels)
    print(f'epoch {epoch + 1}, loss {l:f}')
```

- In PyTorch, the data module provides tools for data processing, the *nnmodule* defines a large number of neural network layers and common loss functions.
- We can initialize the parameters by replacing their values with methods ending with _.