Tensors, Gradient Descent and Linear Regression

In [360…

```
# Run this cell to install PyTorch
!pip3 install torch torchvision
```

```
Requirement already satisfied: torch in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (2.3.1+cu121)
Requirement already satisfied: torchvision in c:\users\mvs aditya\appdata\local\p
rograms\python\python311\lib\site-packages (0.18.1+cu121)
Requirement already satisfied: filelock in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (4.9.0)
Requirement already satisfied: sympy in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (from torch) (1.12)
Requirement already satisfied: networkx in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (2023.12.2)
Requirement already satisfied: mkl<=2021.4.0,>=2021.1.1 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (2021.4.0)
Requirement already satisfied: numpy in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (from torchvision) (1.24.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\mvs aditya\appda
ta\local\programs\python\python311\lib\site-packages (from torchvision) (9.5.0)
Requirement already satisfied: intel-openmp==2021.* in c:\users\mvs aditya\appdat
a\local\programs\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.
1->torch) (2021.4.0)
Requirement already satisfied: tbb==2021.* in c:\users\mvs aditya\appdata\local\p
rograms\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.1->torch)
(2021.11.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\mvs aditya\appdata\loc
al\programs\python\python311\lib\site-packages (from jinja2->torch) (2.1.2)
Requirement already satisfied: mpmath>=0.19 in c:\users\mvs aditya\appdata\local
\programs\python\python311\lib\site-packages (from sympy->torch) (1.3.0)
```

Tensors

At its core, PyTorch is a library for processing tensors. A tensor is a number, vector, matrix or any n-dimensional array. Let's create a tensor with a single number:

In [361…

```
import torch


# Number

t1 = torch.tensor(4.)
t1
# Vector
t2 = torch.tensor([1., 2, 3, 4])
t2
```

```
# Matrix
t3 = torch.tensor([[5., 6],
                   [7, 8],
                   [9, 10]])
t3
# 3-dimensional array
t4 = torch.tensor([
    [[11, 12, 13],
     [13, 14, 15]],
    [[15, 16, 17],
     [17, 18, 19.]]])
t4
```

Out[361]:
```
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
```

Tensors can have any number of dimensions, and different lengths along each dimension. We can inspect the length along each dimension using the .shape property of a tensor.

In [362…
```
print(t1)
print(t1.shape)
print("---------------")
print(t2)
print(t2.shape)
print("---------------")
print(t3)
print(t3.shape)
print("---------------")
print(t4)
print(t4.shape)
print("---------------")
```

```
tensor(4.)
torch.Size([])
---------------
tensor([1., 2., 3., 4.])
torch.Size([4])
---------------
tensor([[ 5.,  6.],
        [ 7.,  8.],
        [ 9., 10.]])
torch.Size([3, 2])
---------------
tensor([[[11., 12., 13.],
         [13., 14., 15.]],

        [[15., 16., 17.],
         [17., 18., 19.]]])
torch.Size([2, 2, 3])
---------------
```

Tensor operations and gradients

In [363…

```
# Create tensors.
x = torch.tensor(3.)
w = torch.tensor(4., requires_grad=True)
b = torch.tensor(5., requires_grad=True)
print(x, w, b)
# Arithmetic operations
y = w * x + b
print(y)
```

```
tensor(3.) tensor(4., requires_grad=True) tensor(5., requires_grad=True)
tensor(17., grad_fn=<AddBackward0>)
```

y is a tensor with the value 3 * 4 + 5 = 17. 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. To compute the derivatives, we can call the .backward method on our result y.

The derivates of y w.r.t the input tensors are stored in the .grad property of the respective tensors.

In [364…

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

In [365…

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

```
dy/dx: None
dy/dw: tensor(3.)
dy/db: tensor(1.)
```

dy/dw has the same value as x i.e. 3, and dy/db has the value 1. Note that x.grad is None, because x doesn't have requires_grad set to True.

The "grad" in w.grad stands for gradient, which is another term for derivative, used mainly when dealing with matrices.

Interoperability with Numpy

Instead of reinventing the wheel, PyTorch interoperates really well with Numpy to leverage its existing ecosystem of tools and libraries.

In [366…

```
import numpy as np

x = np.array([[1, 2], [3, 4.]])
x
```

Out[366]:

```
array([[1., 2.],
       [3., 4.]])
```

We can convert a PyTorch tensor to a Numpy array using the .numpy method of a tensor.

In [367…
```
# Convert the numpy array to a torch tensor.
y = torch.from_numpy(x)
y
```

Out[367]:
```
tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

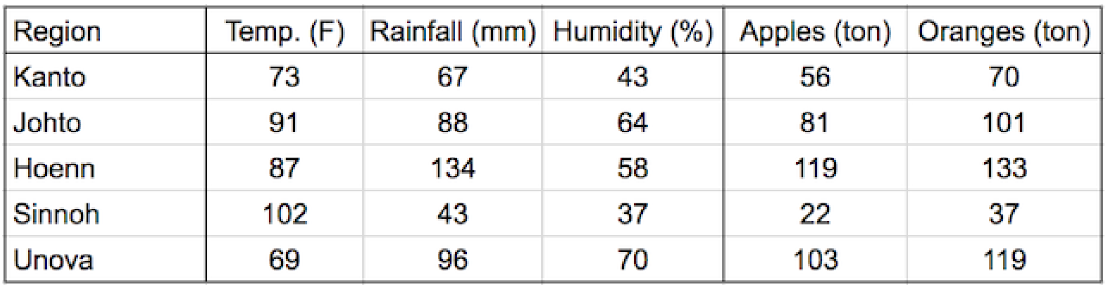
In [368…
```
x.dtype, y.dtype
```

Out[368]:
```
(dtype('float64'), torch.float64)
```

Gradient Descent and Linear Regression with PyTorch

Introduction to Linear Regression

In this tutorial, we'll discuss one of the foundational algorithms in machine learning: *Linear regression*. 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:

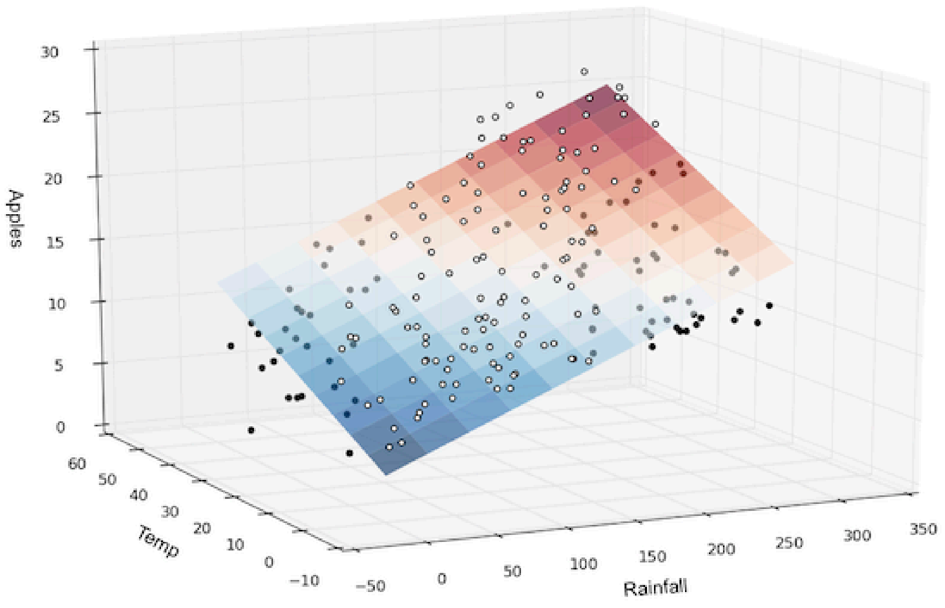


Region	Temp. (F)	Rainfall (mm)	Humidity (%)	Apples (ton)	Oranges (ton)
Kanto	73	67	43	56	70
Johto	91	88	64	81	101
Hoenn	87	134	58	119	133
Sinnoh	102	43	37	22	37
Unova	69	96	70	103	119

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 [369…

```
# Installing Pytorch in Windows OS using pip
!pip3 install torch torchvision torchaudio
!pip install --upgrade torch
```

```
Requirement already satisfied: torch in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (2.3.1+cu121)
Requirement already satisfied: torchvision in c:\users\mvs aditya\appdata\local\p
rograms\python\python311\lib\site-packages (0.18.1+cu121)
Requirement already satisfied: torchaudio in c:\users\mvs aditya\appdata\local\pr
ograms\python\python311\lib\site-packages (2.3.1+cu121)
Requirement already satisfied: filelock in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (4.9.0)
Requirement already satisfied: sympy in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (from torch) (1.12)
Requirement already satisfied: networkx in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (2023.12.2)
Requirement already satisfied: mkl<=2021.4.0,>=2021.1.1 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (2021.4.0)
Requirement already satisfied: numpy in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (from torchvision) (1.24.3)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in c:\users\mvs aditya\appda
ta\local\programs\python\python311\lib\site-packages (from torchvision) (9.5.0)
Requirement already satisfied: intel-openmp==2021.* in c:\users\mvs aditya\appdat
a\local\programs\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.
1->torch) (2021.4.0)
Requirement already satisfied: tbb==2021.* in c:\users\mvs aditya\appdata\local\p
rograms\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.1->torch)
(2021.11.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\mvs aditya\appdata\loc
al\programs\python\python311\lib\site-packages (from jinja2->torch) (2.1.2)
Requirement already satisfied: mpmath>=0.19 in c:\users\mvs aditya\appdata\local
\programs\python\python311\lib\site-packages (from sympy->torch) (1.3.0)
Requirement already satisfied: torch in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (2.3.1+cu121)
Requirement already satisfied: filelock in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (4.9.0)
Requirement already satisfied: sympy in c:\users\mvs aditya\appdata\local\program
s\python\python311\lib\site-packages (from torch) (1.12)
Requirement already satisfied: networkx in c:\users\mvs aditya\appdata\local\prog
rams\python\python311\lib\site-packages (from torch) (3.2.1)
Requirement already satisfied: jinja2 in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\mvs aditya\appdata\local\progra
ms\python\python311\lib\site-packages (from torch) (2023.12.2)
Requirement already satisfied: mkl<=2021.4.0,>=2021.1.1 in c:\users\mvs aditya\ap
pdata\local\programs\python\python311\lib\site-packages (from torch) (2021.4.0)
Requirement already satisfied: intel-openmp==2021.* in c:\users\mvs aditya\appdat
a\local\programs\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.
1->torch) (2021.4.0)
Requirement already satisfied: tbb==2021.* in c:\users\mvs aditya\appdata\local\p
rograms\python\python311\lib\site-packages (from mkl<=2021.4.0,>=2021.1.1->torch)
(2021.11.0)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\mvs aditya\appdata\loc
al\programs\python\python311\lib\site-packages (from jinja2->torch) (2.1.2)
Requirement already satisfied: mpmath>=0.19 in c:\users\mvs aditya\appdata\local
\programs\python\python311\lib\site-packages (from sympy->torch) (1.3.0)
```

In [370…

```
import numpy as np
import pandas as pd
import torch
```

Training data

We can represent the training data using two matrices: `inputs` and `targets`, each with one row per observation, and one column per variable.

In [371…

```
# 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')
```

In [372…

```
# 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 (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 [373…

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

```
tensor([[-1.2059, -0.7075, -0.7227],
        [ 1.6156,  0.5227, -0.0555]], requires_grad=True)
tensor([-1.0917, -0.1764], 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).

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

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

We can define the model as follows:

In [374…
```
def model(x):
    return x @ w.t() + b
```

In [375…
```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[-167.5979,  150.3928],
        [-219.3372,  189.2835],
        [-242.7227,  207.1985],
        [-181.2525,  185.0326],
        [-202.8038,  157.5894]], grad_fn=<AddBackward0>)
```

In [376…
```
# Compare with targets
print(targets)
```

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

A big difference between model's predictions and the actual targets because we've initialized our model with random weights and biases.

Loss function

It is a metric to analyse model performance. FOllowing seps are used to compute a loss function.

- 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 [377...
```
# MSE 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 [378...
```
# Compute loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(44903.4531, 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.*

The above value is called MSE(Mean Square Error) and 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 [379...
```
# Compute gradients
loss.backward()
```

The gradients are stored in the `.grad` property of the respective tensors. Note that the derivative of the loss w.r.t. the weights matrix is itself a matrix with the same dimensions.

In [380...
```
# Gradients for weights
print(w)
print(w.grad)
```

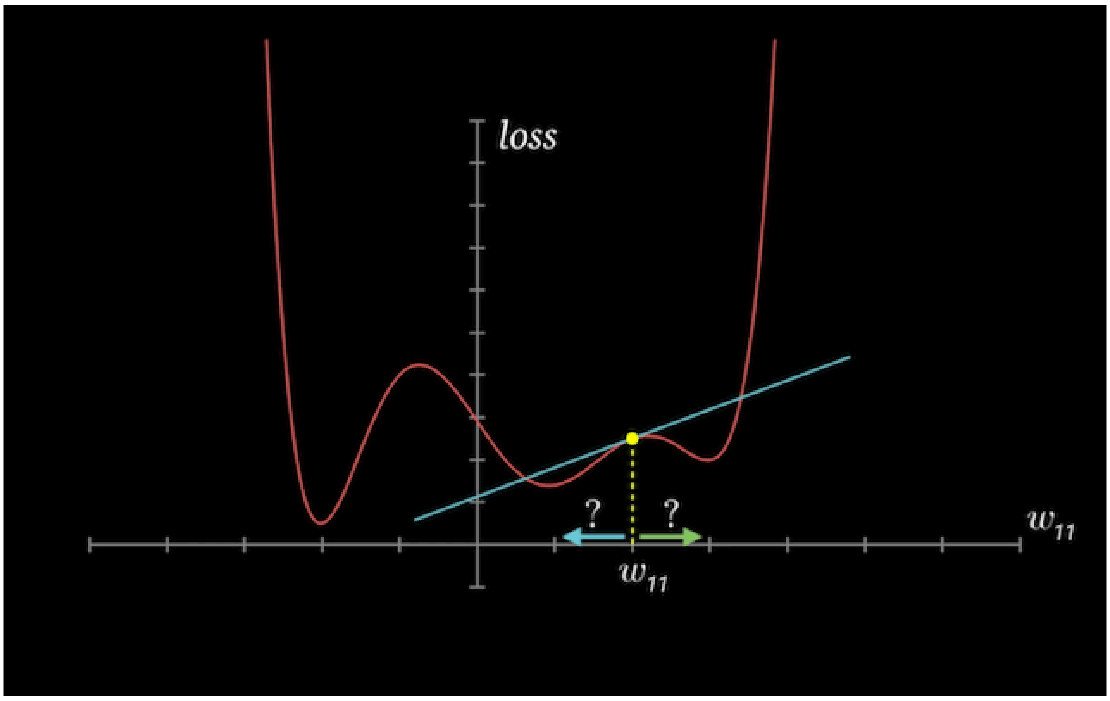
```
tensor([[-1.2059, -0.7075, -0.7227],
        [ 1.6156,  0.5227, -0.0555]], requires_grad=True)
tensor([[-23391.0840, -25595.7188, -15748.5635],
        [  7623.9482,   6633.5713,   4317.8027]])
```

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.

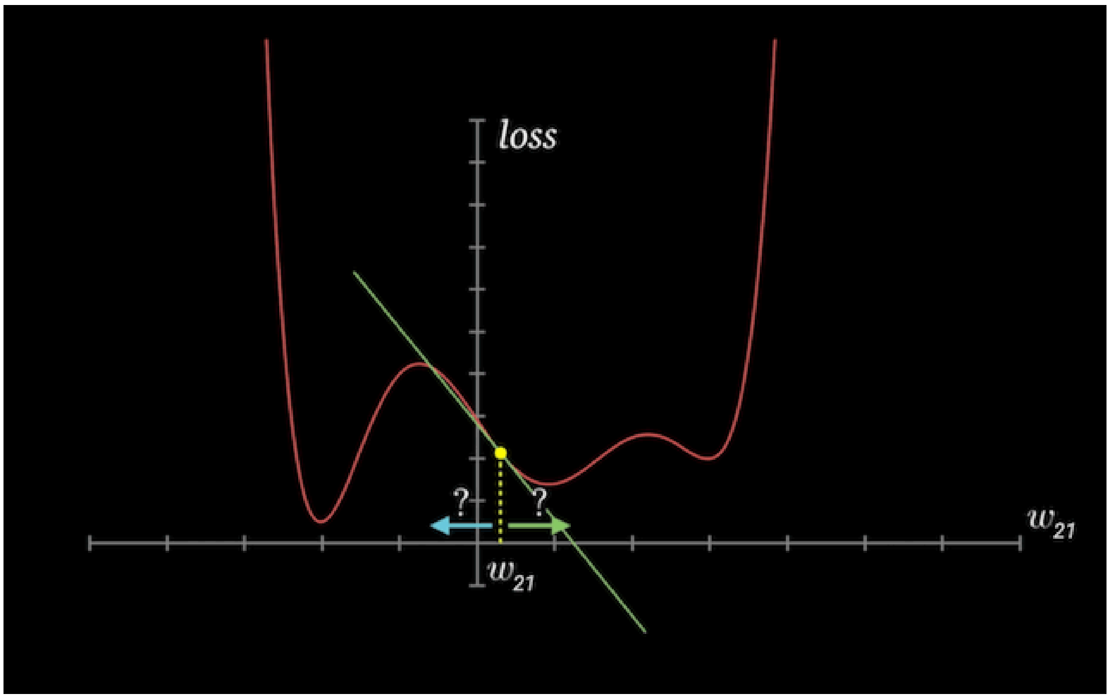
If a gradient element is **positive**:

- **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 [381…
```
w
w.grad
```

Out[381]:
```
tensor([[-23391.0840, -25595.7188, -15748.5635],
        [  7623.9482,   6633.5713,   4317.8027]])
```

In [382…
```
with torch.no_grad():
    w -= w.grad * 1e-5
    b -= b.grad * 1e-5
```

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 [383…
```
# Verifying decrease in loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(44903.4531, 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 [384...
```
w.grad.zero_()
b.grad.zero_()
print(w.grad)
print(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 [385...
```
# Generate predictions
preds = model(inputs)
print(preds)
```

```
tensor([[-126.5986,  138.5253],
        [-165.4452,  173.7439],
        [-178.9373,  189.1715],
        [-140.5577,  172.8053],
        [-151.0653,  142.9373]], grad_fn=<AddBackward0>)
```

In [386...
```
# Calculate the loss
loss = mse(preds, targets)
print(loss)
```

```
tensor(30597.6992, grad_fn=<DivBackward0>)
```

In [387...
```
# Compute gradients
loss.backward()
print(w.grad)
print(b.grad)
```

```
tensor([[-19157.6289, -21045.0254, -12940.7607],
        [  6402.5557,   5331.4507,   3512.1108]])
tensor([-228.7208,   71.4367])
```

In [388...
```
# 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 [389…

```
#Observing new values of weights and bias
print(w)
print(b)
```

```
tensor([[-0.7804, -0.2411, -0.4358],
        [ 1.4753,  0.4030, -0.1338]], requires_grad=True)
tensor([-1.0866, -0.1779], requires_grad=True)
```

In [390…

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

```
tensor(20953.2773, 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 [391…

```
# 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 [392…

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

```
tensor(339.4136, grad_fn=<DivBackward0>)
```

In [393…

```
# Predictions
preds
```

Out[393]:

```
tensor([[ 59.5055,  78.1480],
        [ 80.5209,  98.6568],
        [118.7615, 124.9346],
        [ 35.4027,  82.0193],
        [ 90.4176,  89.3362]], grad_fn=<AddBackward0>)
```

In [394…

```
# Targets
targets
```

Out[394]:

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

The predictions are now quite close to the target variables. We can get even better results by training for a few more epochs.

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 [395…
```
import torch.nn as nn
```

In [396…
```
# 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)
```

In [397…
```
inputs
```

Out[397]:
```
tensor([[ 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.]])
```

We are using 15 training examples to illustrate how to work with large datasets in small batches.

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 [398…
```python
from torch.utils.data import TensorDataset
```

In [399…
```python
# Define dataset
train_ds = TensorDataset(inputs, targets)
train_ds[0:3]
```

Out[399]:
```
(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 [400…
```python
from torch.utils.data import DataLoader
```

In [401…
```python
# Define data loader
batch_size = 5
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

In [402…
```
for xb, yb in train_dl:
    print(xb)
    print(yb)
    break
```

```
tensor([[ 74.,  66.,  43.],
        [ 73.,  66.,  44.],
        [103.,  43.,  36.],
        [ 68.,  96.,  71.],
        [ 73.,  67.,  43.]])
tensor([[ 57.,  69.],
        [ 57.,  69.],
        [ 20.,  38.],
        [104., 118.],
        [ 56.,  70.]])
```

In each iteration of above for loop, 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 [403…
```
# Define model
model = nn.Linear(3, 2)
print(model.weight)
print(model.bias)
```

```
Parameter containing:
tensor([[-0.1585,  0.0751, -0.3877],
        [-0.5218, -0.4689,  0.4455]], requires_grad=True)
Parameter containing:
tensor([ 0.5094, -0.2023], 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 [404…
```
# Parameters
print(list(model.parameters()))
```

```
[Parameter containing:
tensor([[-0.1585,  0.0751, -0.3877],
        [-0.5218, -0.4689,  0.4455]], requires_grad=True), Parameter containing:
tensor([ 0.5094, -0.2023], requires_grad=True)]
```

In [405…
```
# Generate predictions
preds = model(inputs)
preds
```

Out[405]:
```
tensor([[-22.6933, -50.5562],
        [-32.1086, -60.4410],
        [-25.6928, -82.5947],
        [-26.7655, -57.1089],
        [-30.3476, -50.0389],
        [-22.9269, -50.6091],
        [-32.5714, -59.5266],
        [-26.2390, -82.6711],
        [-26.5319, -57.0560],
        [-30.5768, -49.0717],
        [-23.1561, -49.6418],
        [-32.3422, -60.4939],
        [-25.2301, -83.5091],
        [-26.5363, -58.0762],
        [-30.1140, -49.9860]], grad_fn=<AddmmBackward0>)
```

Loss Function

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

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

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 [406…
```python
# Import nn.functional
import torch.nn.functional as F
# Define loss function
loss_fn = F.mse_loss
loss = loss_fn(model(inputs), targets)
print(loss)
```
```
tensor(18438.6562, grad_fn=<MseLossBackward0>)
```

In [407…
```python
# 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 [408…

```
# Utility function to train the model
def fit(num_epochs, model, loss_fn, opt, train_dl):

    # Repeat for given number of epochs
    for epoch in range(num_epochs):

        # Train with batches of data
        for xb,yb in train_dl:

            # 1. Generate predictions
            pred = model(xb)

            # 2. Calculate loss
            loss = loss_fn(pred, yb)

            # 3. Compute gradients
            loss.backward()

            # 4. Update parameters using gradients
            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
```

In [409…

```
fit(100, model, loss_fn, opt, train_dl)
```

```
Epoch [10/100], Loss: 414.5959
Epoch [20/100], Loss: 84.9622
Epoch [30/100], Loss: 237.6550
Epoch [40/100], Loss: 132.7703
Epoch [50/100], Loss: 75.2448
Epoch [60/100], Loss: 44.2275
Epoch [70/100], Loss: 52.3838
Epoch [80/100], Loss: 26.4388
Epoch [90/100], Loss: 19.1119
Epoch [100/100], Loss: 24.6725
```

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.

Above model is trained for 100 epochs and predictions are generated below.

In [410…

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

Out[410]:

```
tensor([[ 58.3567,  71.1177],
        [ 78.8143, 101.7421],
        [123.3406, 129.1967],
        [ 28.0023,  41.9787],
        [ 92.1643, 118.0690],
        [ 57.1787,  70.1890],
        [ 78.0176, 101.9929],
        [123.3123, 129.9212],
        [ 29.1802,  42.9074],
        [ 92.5456, 119.2486],
        [ 57.5600,  71.3686],
        [ 77.6363, 100.8134],
        [124.1372, 128.9458],
        [ 27.6210,  40.7991],
        [ 93.3422, 118.9977]], grad_fn=<AddmmBackward0>)
```

In [411…

```
# Compare with targets
targets
```

Out[411]:

```
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.]])
```

In [412…

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

Out[412]:

```
tensor([[54.2309, 68.8099]], grad_fn=<AddmmBackward0>)
```

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.

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

References

- https://jovian.ml/learn/deep-learning-with-pytorch-zero-to-gans

https://www.youtube.com/playlist?list=PLWKjhJtqVAbm3T2Eq1_KgloC7ogdXxdRa

- https://jovian.ml/learn/deep-learning-with-pytorch-zero-to-gans

https://www.youtube.com/playlist?list=PLWKjhJtqVAbm3T2Eq1_KgloC7ogdXxdRa

```
In [ ]:
```