

Gradient Descent and Linear Regression with PyTorch

Part 2 of "Deep Learning with Pytorch: Zero to GANs"

This tutorial series is a hands-on beginner-friendly introduction to deep learning using [PyTorch](#), an open-source neural networks library. These tutorials take a practical and coding-focused approach. The best way to learn the material is to execute the code and experiment with it yourself. Check out the full series here:

1. [PyTorch Basics: Tensors & Gradients](#)
2. [Gradient Descent & Linear Regression](#)
3. [Working with Images & Logistic Regression](#)
4. [Training Deep Neural Networks on a GPU](#)
5. [Image Classification using Convolutional Neural Networks](#)
6. [Data Augmentation, Regularization and ResNets](#)
7. [Generating Images using Generative Adversarial Networks](#)

This tutorial covers the following topics:

- Introduction to linear regression and gradient descent
- Implementing a linear regression model using PyTorch tensors
- Training a linear regression model using the gradient descent algorithm
- Implementing gradient descent and linear regression using PyTorch built-in

How to run the code

This tutorial is an executable [Jupyter notebook](#) hosted on [Jovian](#) (don't worry if these terms seem unfamiliar; we'll learn more about them soon). You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. [Google Colab](#) is a free online platform for running Jupyter notebooks using Google's cloud infrastructure. You can also select "Run on Binder" or "Run on Kaggle" if you face issues running the notebook on Google Colab.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up [Python](#), download the notebook and install the required libraries. We recommend using the [Conda](#) distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Jupyter Notebooks: This tutorial is a [Jupyter notebook](#) - a document made of *cells*. Each cell can contain code written in Python or explanations in plain English. You can execute code cells and view the results, e.g., numbers, messages, graphs, tables, files, etc. instantly within the notebook. Jupyter is a powerful platform for experimentation and analysis. Don't be afraid to mess around with the code & break things -

you'll learn a lot by encountering and fixing errors. You can use the "Kernel > Restart & Clear Output" or "Edit > Clear Outputs" menu option to clear all outputs and start again from the top.

Before we begin, we need to install the required libraries. The installation of PyTorch may differ based on your operating system / cloud environment. You can find detailed installation instructions here: <https://pytorch.org>.

```
# Uncomment and run the appropriate command for your operating system, if required

# Linux / Binder
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f https

# Windows
# !pip install numpy torch==1.7.0+cpu torchvision==0.8.1+cpu torchaudio==0.7.0 -f https

# MacOS
# !pip install numpy torch torchvision torchaudio
```

Introduction to Linear Regression

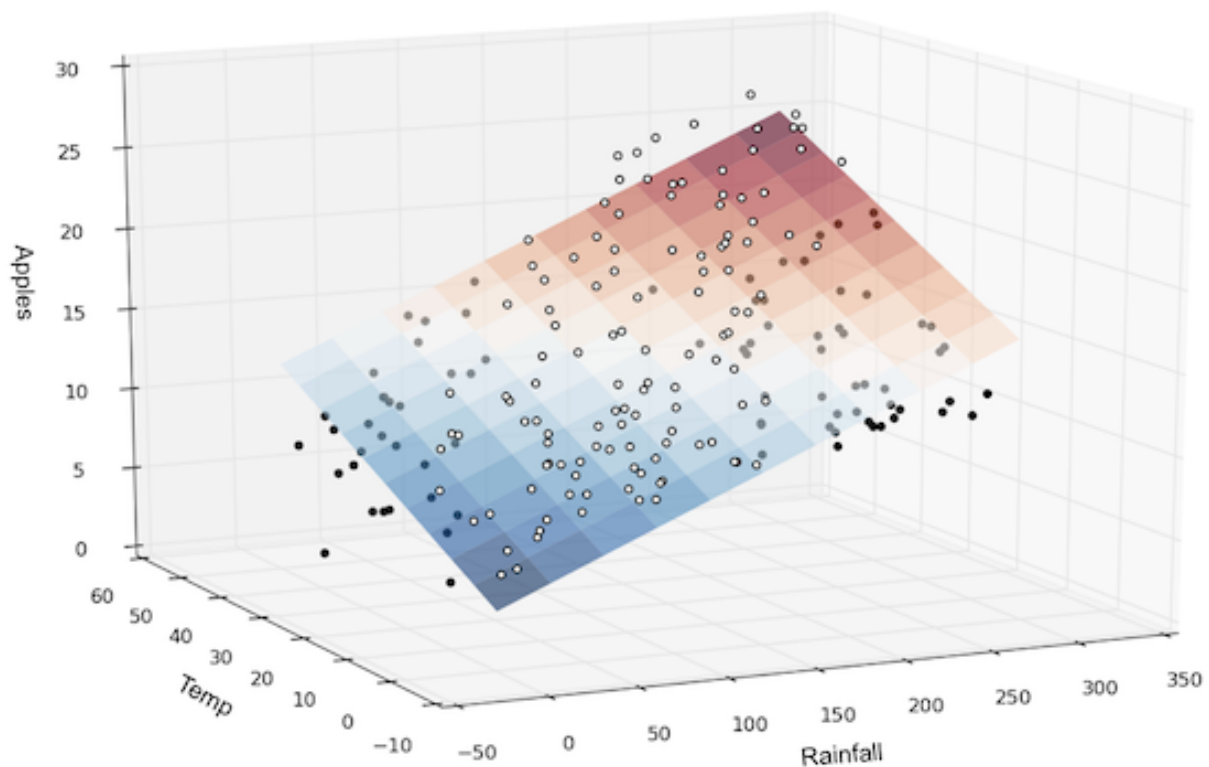
In this tutorial, we'll discuss one of the foundational algorithms in machine learning: *Linear regression*. We'll create a model that predicts crop yields for apples and oranges (*target variables*) by looking at the average temperature, rainfall, and humidity (*input variables or features*) in a region. 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 :

$$\begin{aligned}\text{yield_apple} &= w_{11} * \text{temp} + w_{12} * \text{rainfall} + w_{13} * \text{humidity} + b_1 \\ \text{yield_orange} &= w_{21} * \text{temp} + w_{22} * \text{rainfall} + w_{23} * \text{humidity} + b_2\end{aligned}$$

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



The *learning* part of linear regression is to figure out a set of weights w_{11} , w_{12} , \dots w_{23} , b_1 & b_2 using the training data, to make accurate predictions for new data. The *learned* weights will be used to predict the yields for apples and oranges in a new region using the average temperature, rainfall, and humidity for that region.

We'll *train* our model by adjusting the weights slightly many times to make better predictions, using an optimization technique called *gradient descent*. Let's begin by importing Numpy and PyTorch.

```
import numpy as np
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.

```
# 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')
```

We've separated the input and target variables because we'll operate on them separately. Also, we've created numpy arrays, because this is typically how you would work with training data: read some CSV files as numpy arrays, do some processing, and then convert them to PyTorch tensors.

Let's convert the arrays to PyTorch tensors.

```
# 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 (w_{11} , w_{12} , ... w_{23} , b_1 & b_2) 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.

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

```
tensor([[ -0.2910, -0.3450,  0.0305],  
        [-0.6528,  0.7386, -0.5153]], requires_grad=True)  
tensor([-0.9161, -0.7779], 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).

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

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

We can define the model as follows:

```
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.

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

```
tensor([[ -43.9569, -21.1025],
        [-55.7975, -28.1628],
        [-70.6863,  11.5154],
        [-44.2982, -54.6685],
        [-51.9732, -10.9839]], grad_fn=<AddBackward0>)
```

Let's compare the predictions of our model with the actual targets.

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

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

You can see a big difference between our model's predictions and the actual targets because we've initialized our model with random weights and biases. Obviously, we can't expect a randomly initialized model to *just work*.

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).

```
# 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.

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

```
tensor(15813.8125, 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.

```
# 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.

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

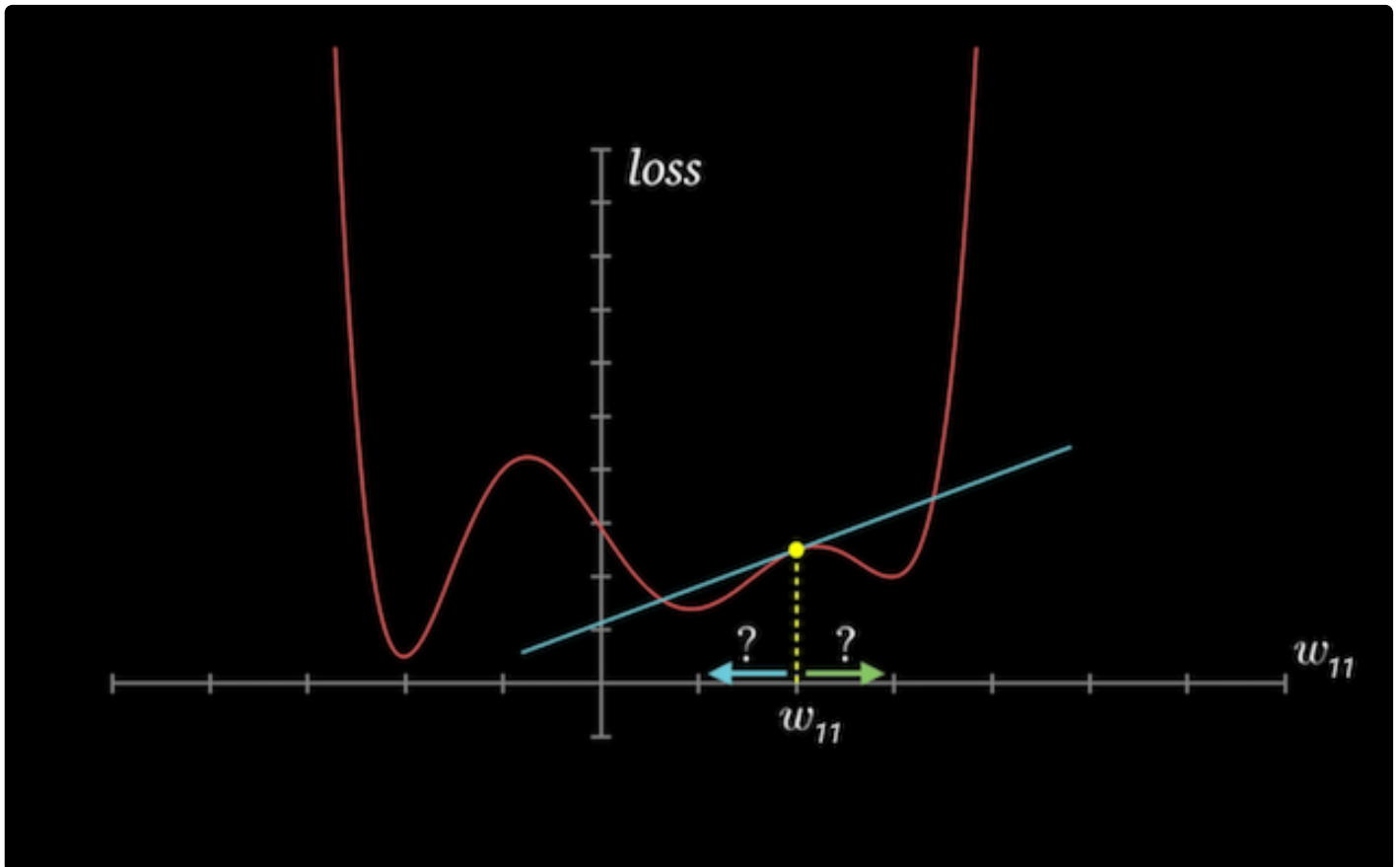
```
tensor([[ -0.2910,  -0.3450,   0.0305],
        [ -0.6528,   0.7386,  -0.5153]], requires_grad=True)
tensor([[ -10740.7393, -12376.3008,  -7471.2300],
        [ -9458.5078, -10033.8672,  -6344.1094]])
```

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.

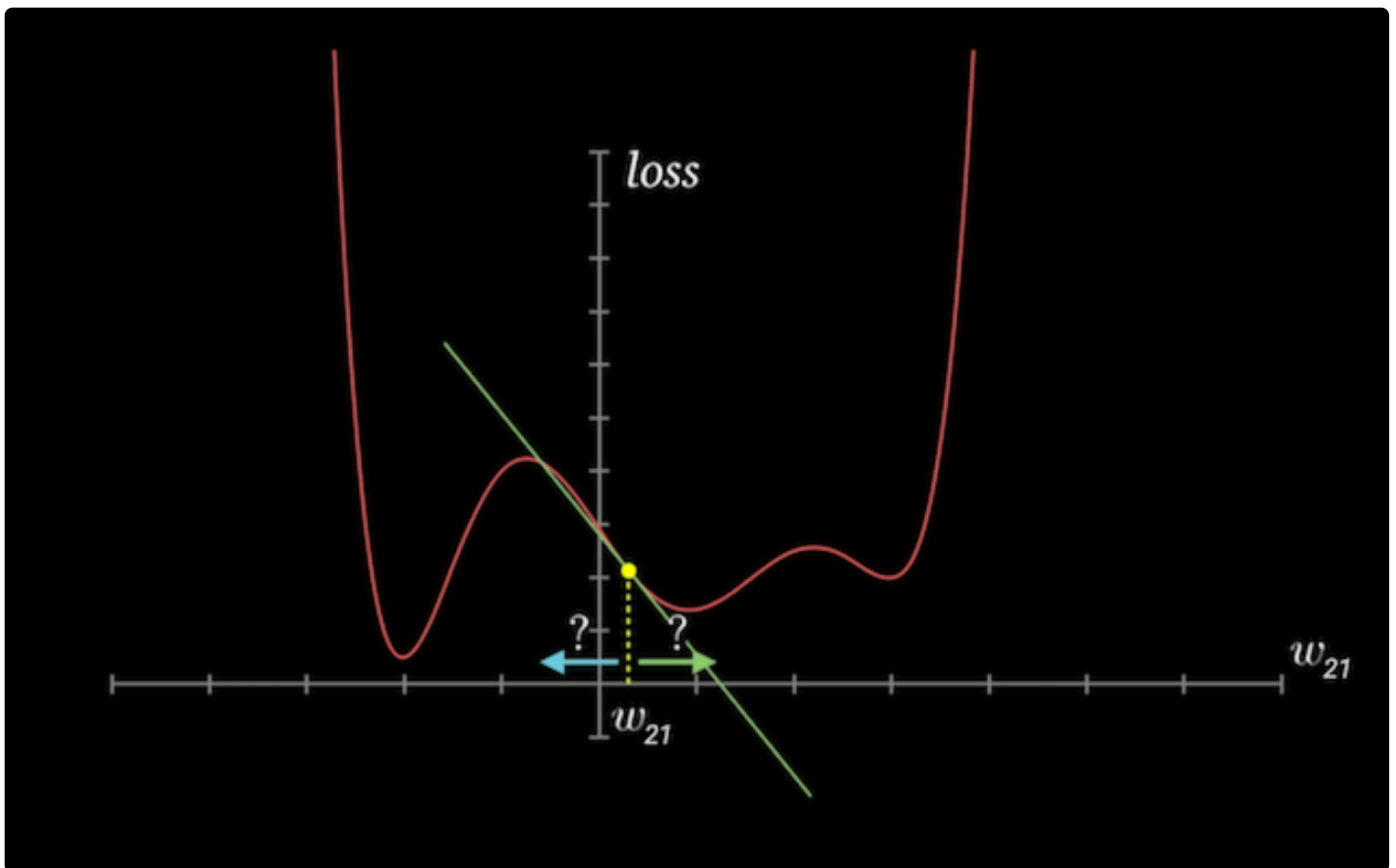
If a gradient element is **positive**:

- **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.

```
w
w.grad
```

```
tensor([[ -10740.7393,  -12376.3008,  -7471.2300],
        [ -9458.5078,  -10033.8672,  -6344.1094]])
```

```
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.

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



```
tensor(15813.8125, 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.

```
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.

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

```
tensor([[ -24.6101,  -4.7460],
        [-30.3494,  -6.6644],
        [-40.4230,  36.8703],
        [-25.2552, -38.3578],
        [-27.4497,   9.6170]], grad_fn=<AddBackward0>)
```

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

```
tensor(10762.5488, grad_fn=<DivBackward0>)
```

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

```
tensor([[ -8741.6377, -10223.4902,  -6143.8101],
        [ -7770.2256,  -8220.9961,  -5225.0342]])
tensor([-105.8175,  -92.6562])
```

Let's update the weights and biases using the gradients computed above.

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

Let's take a look at the new weights and biases.

```
print(w)
print(b)
```

```
tensor([[ -0.0961,  -0.1190,   0.1667],
        [ -0.4805,   0.9211,  -0.3996]], requires_grad=True)
tensor([-0.9137, -0.7759], 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(7357.4829, grad_fn=<DivBackward0>)
```

We have already achieved a significant reduction in the loss merely by adjusting the weights and biases slightly using gradient descent.

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.

```
# 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_()
```

Once again, let's verify that the loss is now lower:

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

```
tensor(130.3513, grad_fn=<DivBackward0>)
```

The loss is now much lower than its initial value. Let's look at the model's predictions and compare them with the targets.

```
# Predictions
preds
```

```
tensor([[ 60.8975,  70.5663],
        [ 83.9699,  92.9066],
        [108.6802, 150.1993],
        [ 43.5842,  38.4608],
        [ 91.6760, 104.6360]], grad_fn=<AddBackward0>)
```

```
# Targets
targets
```

```
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.

Save and upload your notebook

Whether you're running this Jupyter notebook online or on your computer, it's essential to save your work from time to time. You can continue working on a saved notebook later or share it with friends and colleagues to let them execute your code. [Jovian](#) offers an easy way of saving and sharing your Jupyter notebooks online.

First, you need to install the Jovian python library if it isn't already installed.

```
!pip install jovian --upgrade -q
```

```
import jovian
```

```
jovian.commit(project='02-linear-regression')
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/02-linear-regression" on https://jovian.ai/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ai/aakashns/02-linear-regression
'https://jovian.ai/aakashns/02-linear-regression'
```

`jovian.commit` uploads the notebook to your Jovian account, captures the Python environment, and creates a shareable link for your notebook, as shown above. You can use this link to share your work and let anyone (including you) run your notebooks and reproduce your work.

The first time you run `jovian.commit`, you may be asked to provide an *API Key* to securely upload the notebook to your Jovian account. You can get the API key from your [Jovian profile page](#) after logging in / signing up.

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.

```
import torch.nn as nn
```

As before, we represent the inputs and targets and matrices.

```
# 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)

```

inputs

```

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.

```

from torch.utils.data import TensorDataset

```

```

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

```

```

(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.

```
from torch.utils.data import DataLoader
```

```

# Define data loader
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.

```

for xb, yb in train_dl:
    print(xb)
    print(yb)
    break

```

```

tensor([[102.,  43.,  37.],
        [ 92.,  87.,  64.],
        [ 87., 134.,  58.],
        [ 69.,  96.,  70.],
        [101.,  44.,  37.]])
tensor([[ 22.,  37.],
        [ 82., 100.],
        [119., 133.],
        [103., 119.],
        [ 21.,  38.]])

```

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.

```

# Define model
model = nn.Linear(3, 2)

```

```
print(model.weight)
print(model.bias)
```

Parameter containing:

```
tensor([[ 0.1304, -0.1898,  0.2187],
        [ 0.2360,  0.4139, -0.4540]], requires_grad=True)
```

Parameter containing:

```
tensor([0.3457, 0.3883], 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.

```
# Parameters
list(model.parameters())
```

[Parameter containing:

```
tensor([[ 0.1304, -0.1898,  0.2187],
        [ 0.2360,  0.4139, -0.4540]], requires_grad=True),
```

Parameter containing:

```
tensor([0.3457, 0.3883], requires_grad=True)]
```

We can use the model to generate predictions in the same way as before.

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

```
tensor([[ 6.5493, 25.8226],
        [ 9.5025, 29.2272],
        [-1.0633, 50.0460],
        [13.5738, 25.4576],
        [ 6.4278, 24.6221],
        [ 6.8695, 25.6447],
        [ 9.9110, 28.3593],
        [-0.7142, 49.8280],
        [13.2536, 25.6355],
        [ 6.5161, 23.9321],
        [ 6.9578, 24.9546],
        [ 9.8227, 29.0494],
        [-1.4718, 50.9139],
        [13.4855, 26.1476],
        [ 6.1076, 24.8000]], grad_fn=<AddmmBackward>)
```

Loss Function

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

```
# Import nn.functional
import torch.nn.functional as F
```

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

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

Let's compute the loss for the current predictions of our model.

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

```
tensor(5427.9517, grad_fn=<MseLossBackward>)
```

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.

```
# 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.

```
# 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.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.

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

```

Epoch [10/100], Loss: 818.6476
Epoch [20/100], Loss: 335.3347
Epoch [30/100], Loss: 190.3544
Epoch [40/100], Loss: 131.6701
Epoch [50/100], Loss: 77.0783
Epoch [60/100], Loss: 151.5671
Epoch [70/100], Loss: 151.0817
Epoch [80/100], Loss: 67.6262
Epoch [90/100], Loss: 53.6205
Epoch [100/100], Loss: 33.4517

```

Let's generate predictions using our model and verify that they're close to our targets.

```

# Generate predictions
preds = model(inputs)

```

```
preds
```

```
tensor([[ 58.4229,  72.0145],
        [ 82.1525,  95.1376],
        [115.8955, 142.6296],
        [ 28.6805,  46.0115],
        [ 97.5243, 104.3522],
        [ 57.3792,  70.9543],
        [ 81.9342,  94.1737],
        [116.2036, 142.6871],
        [ 29.7242,  47.0717],
        [ 98.3498, 104.4486],
        [ 58.2047,  71.0507],
        [ 81.1088,  94.0774],
        [116.1137, 143.5935],
        [ 27.8550,  45.9152],
        [ 98.5680, 105.4124]], grad_fn=<AddmmBackward>)
```

```
# Compare with targets
```

```
targets
```

```
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 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.

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

```
tensor([[55.3323, 67.8895]], grad_fn=<AddmmBackward>)
```

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

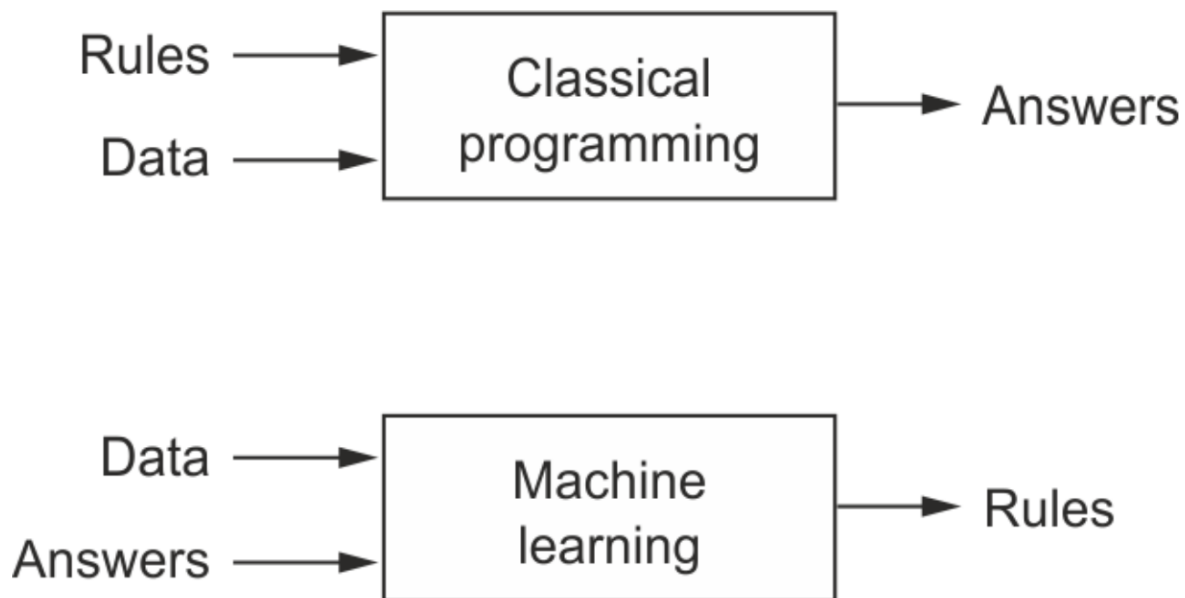
Machine Learning vs. Classical Programming

The approach we've taken in this tutorial is very different from programming as you might know it. Usually, we write programs that take some inputs, perform some operations, and return a result.

However, in this notebook, we've defined a "model" that assumes a specific relationship between the inputs and the outputs, expressed using some unknown parameters (weights & biases). We then show the model some known inputs and outputs and *train* the model to come up with good values for the unknown parameters. Once trained, the model can be used to compute the outputs for new inputs.

This paradigm of programming is known as *machine learning*, where we use data to figure out the relationship between inputs and outputs. *Deep learning* is a branch of machine learning that uses matrix operations, non-linear activation functions and gradient descent to build and train models. Andrej Karpathy, the director of AI at Tesla Motors, has written a great blog post on this topic, titled [Software 2.0](#).

This picture from book [Deep Learning with Python](#) by Francois Chollet captures the difference between classical programming and machine learning:



Keep this picture in mind as you work through the next few tutorials.

Commit and update the notebook

As a final step, we can record a new version of the notebook using the `jovian` library.

```
import jovian
```

```
jovian.commit(project='02-linear-regression')
```

```
[jovian] Attempting to save notebook..
```

```
[jovian] Updating notebook "aakashns/02-linear-regression" on https://jovian.ai/
```

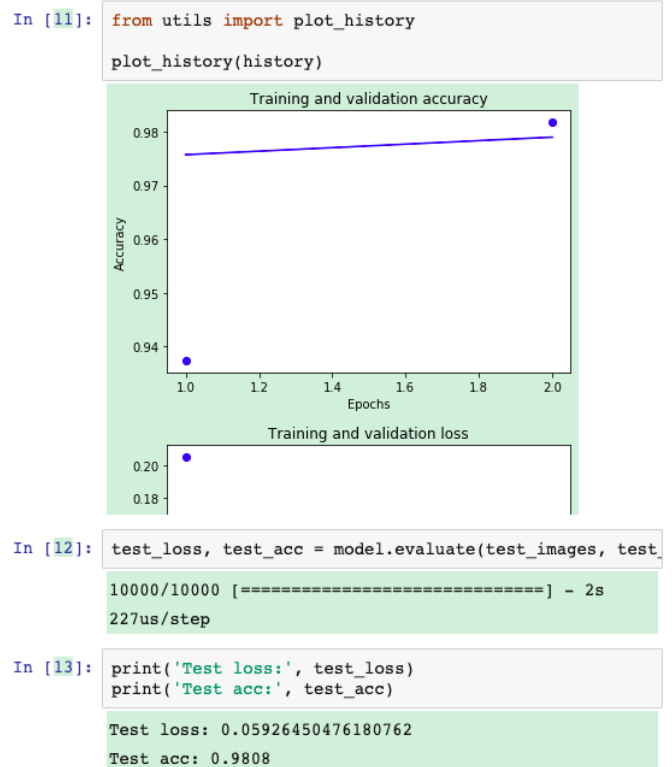
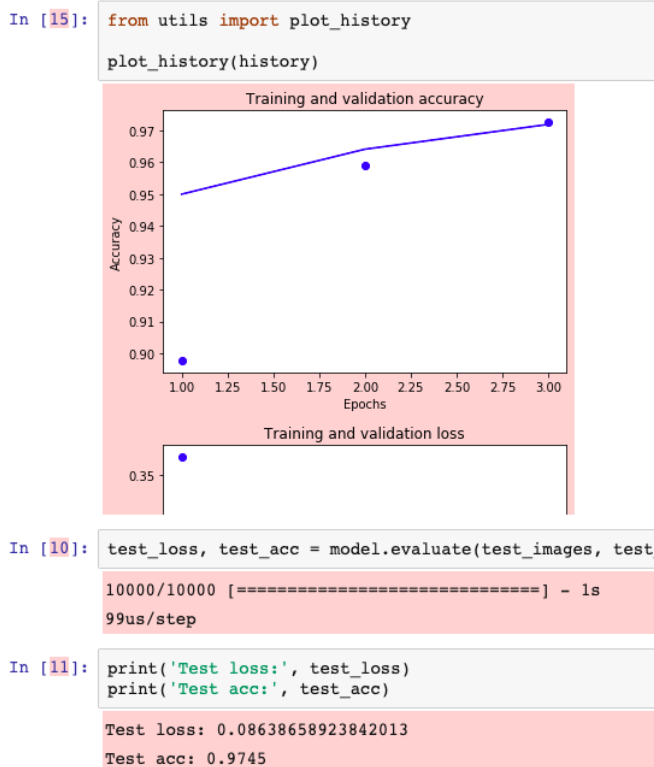
```
[jovian] Uploading notebook..
```

```
[jovian] Capturing environment..
```

```
[jovian] Committed successfully! https://jovian.ai/aakashns/02-linear-regression
```

'<https://jovian.ai/aakashns/02-linear-regression>'

Note that running `jovian.commit` a second time records a new version of your existing notebook. With Jovian.ml, you can avoid creating copies of your Jupyter notebooks and keep versions organized. Jovian also provides a visual diff ([example](#)) so you can inspect what has changed between different versions:



Exercises and Further Reading

We've covered the following topics in this tutorial:

- Introduction to linear regression and gradient descent
- Implementing a linear regression model using PyTorch tensors
- Training a linear regression model using the gradient descent algorithm
- Implementing gradient descent and linear regression using PyTorch built-in

Here are some resources for learning more about linear regression and gradient descent:

- An visual & animated explanation of gradient descent: <https://www.youtube.com/watch?v=IHZwWFHWa-w>
- For a more detailed explanation of derivatives and gradient descent, see [these notes from a Udacity course](#).
- For an animated visualization of how linear regression works, [see this post](#).
- For a more mathematical treatment of matrix calculus, linear regression and gradient descent, you should check out [Andrew Ng's excellent course notes](#) from CS229 at Stanford University.
- To practice and test your skills, you can participate in the [Boston Housing Price Prediction](#) competition on Kaggle, a website that hosts data science competitions.

With this, we complete our discussion of linear regression in PyTorch, and we're ready to move on to the next topic: [Working with Images & Logistic Regression](#).

Questions for Review

Try answering the following questions to test your understanding of the topics covered in this notebook:

1. What is a linear regression model? Give an example of a problem formulated as a linear regression model.
2. What are input and target variables in a dataset? Give an example.
3. What are weights and biases in a linear regression model?
4. How do you represent tabular data using PyTorch tensors?
5. Why do we create separate matrices for inputs and targets while training a linear regression model?
6. How do you determine the shape of the weights matrix & bias vector given some training data?
7. How do you create randomly initialized weights & biases with a given shape?
8. How is a linear regression model implemented using matrix operations? Explain with an example.
9. How do you generate predictions using a linear regression model?
10. Why are the predictions of a randomly initialized model different from the actual targets?
11. What is a loss function? What does the term "loss" signify?
12. What is mean squared error?
13. Write a function to calculate mean squared using model predictions and actual targets.
14. What happens when you invoke the `.backward` function on the result of the mean squared error loss function?
15. Why is the derivative of the loss w.r.t. the weights matrix itself a matrix? What do its elements represent?
16. How is the derivative of the loss w.r.t. a weight element useful for reducing the loss? Explain with an example.
17. Suppose the derivative of the loss w.r.t. a weight element is positive. Should you increase or decrease the element's value slightly to get a lower loss?
18. Suppose the derivative of the loss w.r.t. a weight element is negative. Should you increase or decrease the element's value slightly to get a lower loss?
19. How do you update the weights and biases of a model using their respective gradients to reduce the loss slightly?
20. What is the gradient descent optimization algorithm? Why is it called "gradient descent"?
21. Why do you subtract a "small quantity" proportional to the gradient from the weights & biases, not the actual gradient itself?
22. What is learning rate? Why is it important?
23. What is `torch.no_grad`?
24. Why do you reset gradients to zero after updating weights and biases?
25. What are the steps involved in training a linear regression model using gradient descent?
26. What is an epoch?
27. What is the benefit of training a model for multiple epochs?
28. How do you make predictions using a trained model?
29. What should you do if your model's loss doesn't decrease while training? Hint: learning rate.
30. What is `torch.nn`?
31. What is the purpose of the `TensorDataset` class in PyTorch? Give an example.
32. What is a data loader in PyTorch? Give an example.

33. How do you use a data loader to retrieve batches of data?
34. What are the benefits of shuffling the training data before creating batches?
35. What is the benefit of training in small batches instead of training with the entire dataset?
36. What is the purpose of the `nn.Linear` class in PyTorch? Give an example.
37. How do you see the weights and biases of a `nn.Linear` model?
38. What is the purpose of the `torch.nn.functional` module?
39. How do you compute mean squared error loss using a PyTorch built-in function?
40. What is an optimizer in PyTorch?
41. What is `torch.optim.SGD`? What does SGD stand for?
42. What are the inputs to a PyTorch optimizer?
43. Give an example of creating an optimizer for training a linear regression model.
44. Write a function to train a `nn.Linear` model in batches using gradient descent.
45. How do you use a linear regression model to make predictions on previously unseen data?