

Linear Regression with PyTorch

Part 2 of "PyTorch: Zero to GANs"

This post is the second in a series of tutorials on building deep learning models with PyTorch, an open source neural networks library developed and maintained by Facebook. Check out the full series:

1. [PyTorch Basics: Tensors & Gradients](#)
2. [Linear Regression & Gradient Descent](#)
3. [Image Classification using 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](#)

Continuing where the [previous tutorial](#) left off, we'll discuss one of the foundational algorithms of machine learning in this post: *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:

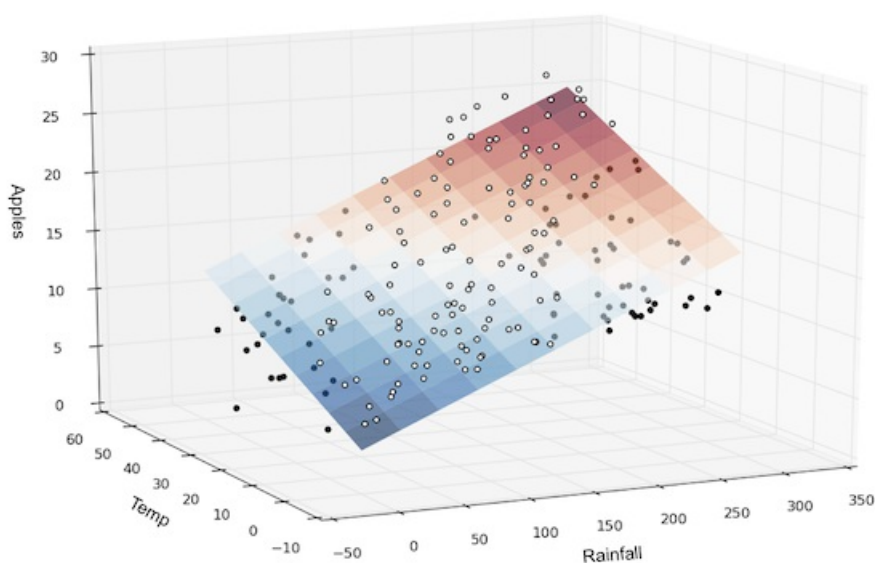
Region	Temp. (F)	Rainfall (mm)	Humidity (%)	Apples (ton)	Oranges (ton)
Kanto	73	67	43	56	70
Johto	91	88	64	81	101
Hoenn	87	134	58	119	133
Sinnoh	102	43	37	22	37
Unova	69	96	70	103	119

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 :

$$\text{yield_apple} = w_{11} * \text{temp} + w_{12} * \text{rainfall} + w_{13} * \text{humidity} + b_1$$

$$\text{yield_orange} = w_{21} * \text{temp} + w_{22} * \text{rainfall} + w_{23} * \text{humidity} + b_2$$

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 by looking at the training data, to make accurate predictions for new data (i.e. to predict the yields for apples and oranges in a new region using the average temperature, rainfall and humidity). This is done by adjusting the weights slightly many times to make better predictions, using an optimization technique called *gradient descent*.

System setup

This tutorial takes a code-first approach towards learning PyTorch, and you should try to follow along by running and experimenting with the code yourself. The easiest way to start executing this notebook is to click the **"Run"** button at the top of this page, and select **"Run on Binder"**. This will run the notebook on mybinder.org, a free online service for running Jupyter notebooks.

NOTE: If you're running this notebook on Binder, please skip ahead to the next section.

Running on your computer locally

You can clone this notebook hosted on [Jovian.ml](https://jovian.ml), install the required dependencies, and start Jupyter by running the following commands on the terminal:

```
pip install jovian --upgrade      # Install the jovian library
jovian clone aakashns/02-linear-regression  # Download notebook & dependencies
cd 02-linear-regression          # Enter the created directory
jovian install                   # Install the dependencies
conda activate 02-linear-regression # Activate virtual environment
jupyter notebook                 # Start Jupyter
```

On older versions of conda, you might need to run `source activate 02-linear-regression` to activate the environment. For a more detailed explanation of the above steps, check out the *System setup* section in the [previous notebook](#).

We begin by importing Numpy and PyTorch:

In []:

```
# Uncomment the command below if Numpy or PyTorch is not installed
# !conda install numpy pytorch cpuonly -c pytorch -y
```

In [1]:

```
import numpy as np
import torch
```

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 [2]:

```
# 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 [3]:

```
# 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 as follows:

In [4]:

```
# Convert inputs and targets to tensors
inputs = torch.from_numpy(inputs)
targets = torch.from_numpy(targets)
```

```

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 [5]:

```

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

```

```

tensor([[ -0.9876, -1.2698,  1.1006],
        [-0.4972,  0.6133,  0.4304]], requires_grad=True)
tensor([ 1.9560, -1.1333], 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).

$$\begin{matrix} X & \times & W^T & + & b \end{matrix}$$

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

We can define the model as follows:

In [6]:

```

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 [7]:

```

# Generate predictions
preds = model(inputs)

```

```
print(preds)
```

```
tensor([[[-107.8932,    22.1622],
         [-129.2238,    35.1282],
         [-190.2896,    62.7447],
         [-112.6626,   -9.5585],
         [-111.0506,    53.5561]], grad_fn=<AddBackward0>)
```

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

In [8]:

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

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

You can see that there's a huge difference between the predictions of our model, and the actual values of the target variables. Obviously, this is because we've initialized our model with random weights and biases, and we can't expect it 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).

In [19]:

```
# 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, and the `.numel` method returns the number of elements in a tensor. Let's compute the mean squared error for the current predictions of our model.

In [20]:

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

```
tensor(24868.0703, grad_fn=<DivBackward0>)
```

Here's how we can interpret the result: *On average, each element in the prediction differs from the actual target by about 145 (square root of the loss 20834)*. And that's pretty bad, considering the numbers we are trying to predict are themselves in the range 50–200. Also, the result 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 [21]:

```
# 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 [22]:

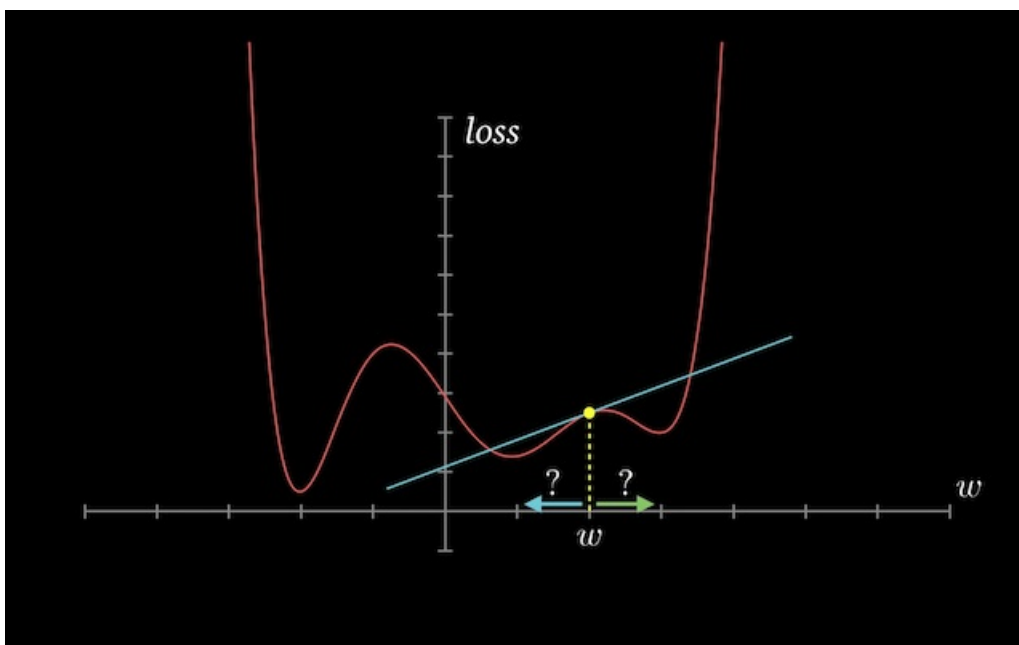
```
# Gradients for weights
print(w)
print(w.grad)
```

```
tensor([[ -0.9876, -1.2698,  1.1006],
        [-0.4972,  0.6133,  0.4304]], requires_grad=True)
tensor([[ -17301.5684, -19452.9375, -11681.3174],
        [ -4972.6597, -5340.1387, -3330.2732]])
```

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. 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.

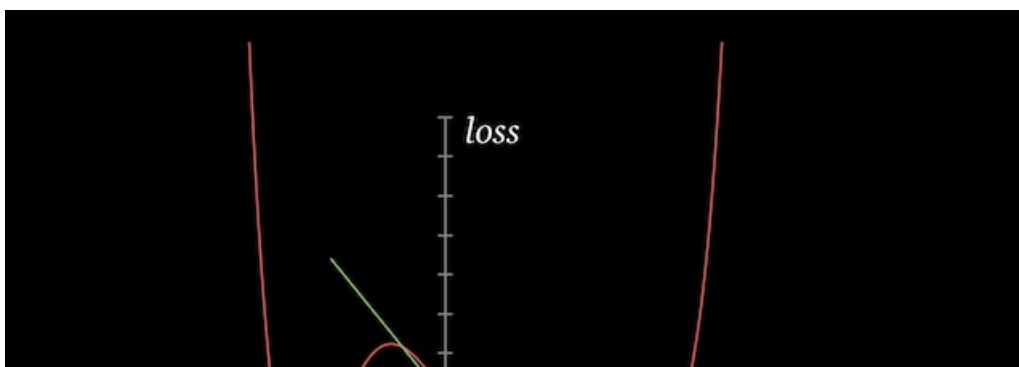
If a gradient element is **positive**:

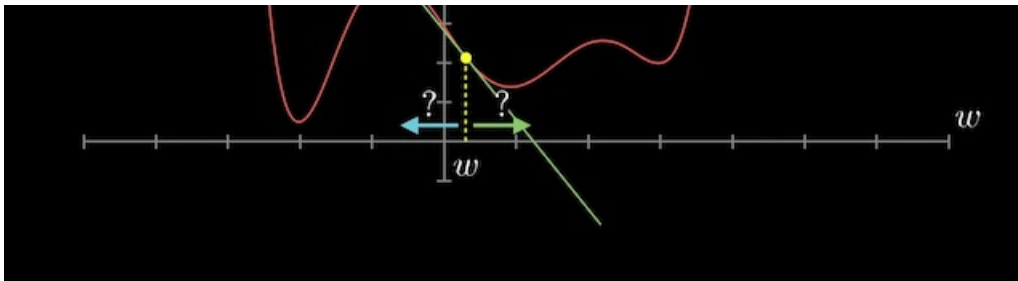
- **increasing** the element's value slightly will **increase** the loss.
- **decreasing** the element's value slightly will **decrease** the loss



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 in loss by changing a weight element is proportional to the value of the gradient of the loss w.r.t. that element. This forms the basis for the optimization algorithm that we'll use to improve our model.

Before we proceed, we reset the gradients to zero by calling `.zero_()` method. We need to do this, because PyTorch accumulates gradients i.e. the next time we call `.backward` on the loss, the new gradient values will get added to the existing gradient values, which may lead to unexpected results.

In [23]:

```
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 optimization 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

Let's implement the above step by step.

In [25]:

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

```
tensor([[ -107.8932,  22.1622],
        [-129.2238,  35.1282],
        [-190.2896,  62.7447],
        [-112.6626,  -9.5585],
        [-111.0506,  53.5561]], grad_fn=<AddBackward0>)
```

Note that the predictions are same as before, since we haven't made any changes to our model. The same holds true for the loss and gradients.

In [26]:

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

```
tensor(24868.0703, grad_fn=<DivBackward0>)
```

In [27]:

```
# Compute gradients
```

```
loss.backward()
print(w.grad)
print(b.grad)
```

```
tensor([[ -17301.5684, -19452.9375, -11681.3174],
        [ -4972.6597,  -5340.1387,  -3330.2732]])
tensor([-206.4240,  -59.1935])
```

Finally, we update the weights and biases using the gradients computed above.

In [28]:

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

A few things to note above:

- We use `torch.no_grad` to indicate to PyTorch that we shouldn't track, calculate or modify gradients while updating the weights and biases.
- We multiply the gradients with a really small number (`10-5` in this case), to ensure that we don't modify the weights by a really large amount, since we only want to take a small step in the downhill direction of the gradient. This number is called the *learning rate* of the algorithm.
- After we have updated the weights, we reset the gradients back to zero, to avoid affecting any future computations.

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

In [29]:

```
print(w)
print(b)

tensor([[ -0.8146, -1.0753,  1.2175],
        [ -0.4475,  0.6667,  0.4637]], requires_grad=True)
tensor([ 1.9581, -1.1327], requires_grad=True)
```

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

In [30]:

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

```
tensor(16868.5664, grad_fn=<DivBackward0>)
```

We have already achieved a significant reduction in the loss, simply 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.

In [31]:

```
# 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:

In [32]:

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

```
tensor(141.4334, grad_fn=<DivBackward0>)
```

As you can see, the loss is now much lower than what we started out with. Let's look at the model's predictions and compare them with the targets.

In [33]:

```
# Predictions
preds
```

Out[33]:

```
tensor([[ 60.3710,  70.3766],
        [ 92.8352,  98.7249],
        [ 89.3732, 137.2704],
        [ 38.6567,  38.3061],
        [110.5414, 114.7776]], grad_fn=<AddBackward0>)
```

In [34]:

```
# Targets
targets
```

Out[34]:

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

The prediction are now quite close to the target variables, and we can get even better results by training for a few more epochs.

At this point, we can save our notebook and upload it to [Jovian.ml](https://jovian.ml) for future reference and sharing.

In [35]:

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

In [36]:

```
import jovian
```

In [37]:

```
jovian.commit()
```

```
[jovian] Attempting to save notebook..
[jovian] Please enter your API key ( from https://jovian.ml/ ):
```

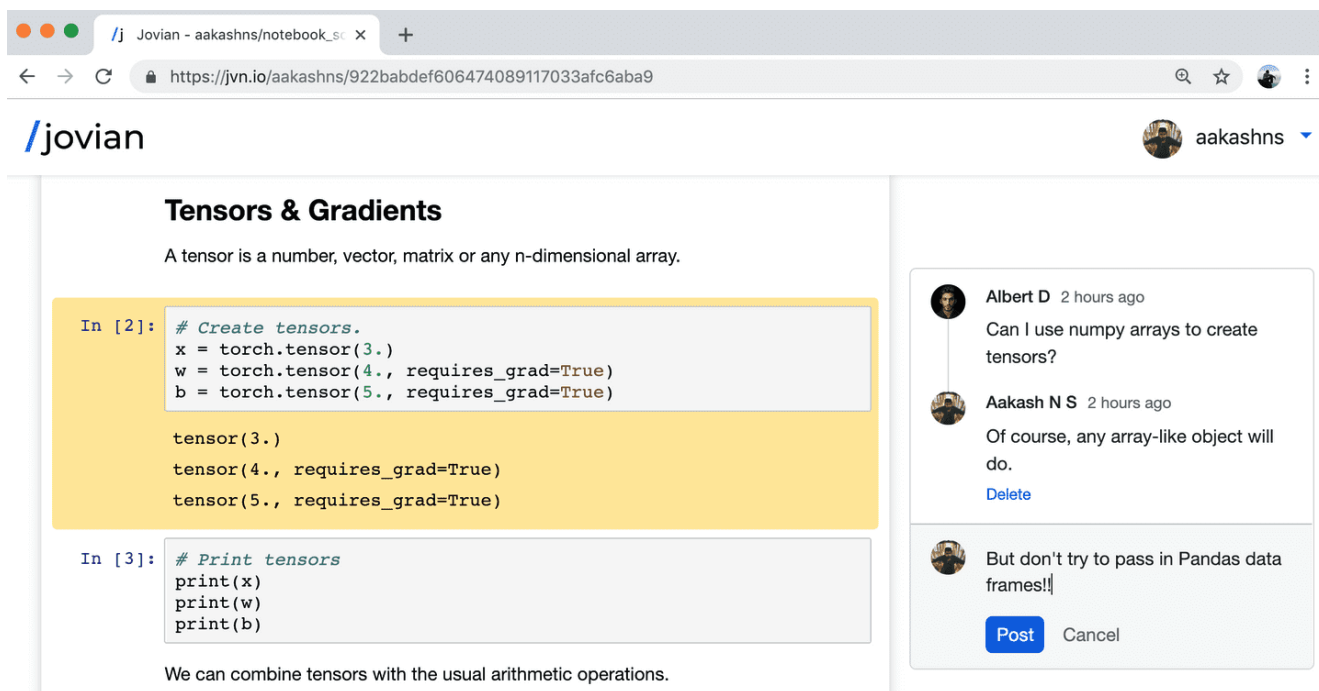


```
API KEY: .....
[jovian] Updating notebook "aakashns/02-linear-regression" on https://jovian.ml/
[jovian] Uploading notebook..
[jovian] Capturing environment..
[jovian] Committed successfully! https://jovian.ml/aakashns/02-linear-regression
```

Out[37]:

'https://jovian.ml/aakashns/02-linear-regression'

`jovian.commit` uploads the notebook to [Jovian.ml](https://jovian.ml), captures the Python environment and creates a sharable link for the notebook. You can use this link to share your work and let anyone reproduce it easily with the `jovian clone` command. Jovian also includes a powerful commenting interface, so you (and others) can discuss & comment on specific parts of your notebook:



The screenshot shows a web browser window with the Jovian interface. The notebook title is "Tensors & Gradients". The code cell contains two input blocks. The first block defines three tensors: `x = torch.tensor(3.)`, `w = torch.tensor(4., requires_grad=True)`, and `b = torch.tensor(5., requires_grad=True)`. The second block prints these tensors. The output shows the tensor values and their properties. A comment thread on the right shows a discussion about using numpy arrays and pandas data frames.

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

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

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

Albert D 2 hours ago
Can I use numpy arrays to create tensors?

Aakash N S 2 hours ago
Of course, any array-like object will do.
[Delete](#)

But don't try to pass in Pandas data frames!!
[Post](#) [Cancel](#)

Linear regression using PyTorch built-ins

The model and training process above were implemented using basic matrix operations. But since this such a common pattern , PyTorch has several built-in functions and classes to make it easy to create and train models.

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

In [38]:

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

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

In [39]:

```
# Input (temp, rainfall, humidity)
inputs = np.array([[73, 67, 43], [91, 88, 64], [87, 134, 58],
                  [102, 43, 37], [69, 96, 70], [73, 67, 43],
                  [91, 88, 64], [87, 134, 58], [102, 43, 37],
                  [69, 96, 70], [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], [56, 70],
                   [81, 101], [119, 133], [22, 37],
                   [103, 119], [56, 70], [81, 101],
                   [119, 133], [22, 37], [103, 119]],
                   dtype='float32')
```

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

In [40]:

```
inputs
```

Out[40]:

```
tensor([[ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 87., 134.,  58.],
        [102.,  43.,  37.],
        [ 69.,  96.,  70.],
        [ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 87., 134.,  58.],
        [102.,  43.,  37.],
        [ 69.,  96.,  70.],
        [ 73.,  67.,  43.],
        [ 91.,  88.,  64.],
        [ 87., 134.,  58.],
        [102.,  43.,  37.],
        [ 69.,  96.,  70.]])
```

We are using 15 training examples this time, 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 [41]:

```
from torch.utils.data import TensorDataset
```

In [42]:

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

Out[42]:

```
(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 (or pair), in which 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 [43]:

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

In [44]:

```
# Define data loader
batch_size = 5
train_dl = DataLoader(train_ds, batch_size, shuffle=True)
```

The data loader is typically used in a `for-in` loop. Let's look at an example.

In [46]:

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

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

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, which can lead to 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 [47]:

```
# Define model
model = nn.Linear(3, 2)
print(model.weight)
print(model.bias)
```

```
Parameter containing:
tensor([[ 0.5544,  0.0678, -0.5019],
        [-0.5014, -0.1209, -0.4819]], requires_grad=True)
Parameter containing:
tensor([-0.1549,  0.2659], 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 [48]:

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

Out[48]:

```
[Parameter containing:
 tensor([[ 0.5544,  0.0678, -0.5019],
        [-0.5014, -0.1209, -0.4819]], requires_grad=True),
 Parameter containing:
 tensor([-0.1549,  0.2659], requires_grad=True)]
```

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

In [49]:

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

```
preas
```

Out[49]:

```
tensor([[ 23.2769, -65.1601],
        [ 24.1400, -86.8446],
        [ 28.0528, -87.5095],
        [ 40.7381, -73.9080],
        [  9.4745, -79.6720],
        [ 23.2769, -65.1601],
        [ 24.1400, -86.8446],
        [ 28.0528, -87.5095],
        [ 40.7381, -73.9080],
        [  9.4745, -79.6720],
        [ 23.2769, -65.1601],
        [ 24.1400, -86.8446],
        [ 28.0528, -87.5095],
        [ 40.7381, -73.9080],
        [  9.4745, -79.6720]], grad_fn=<AddmmBackward>)
```

Loss Function

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

In [50]:

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

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

In [51]:

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

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

In [52]:

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

```
tensor(17562.2832, grad_fn=<MseLossBackward>)
```

Optimizer

Instead of manually manipulating the model's weights & biases using gradients, we can use the optimizer `optim.SGD`. SGD stands for `stochastic gradient descent`. It is called `stochastic` because samples are selected in batches (often with random shuffling) instead of as a single group.

In [53]:

```
# 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 which controls the amount by which the parameters are modified.

Train the model

We are now ready to train the model. We'll follow the exact 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` which trains the model for a given number of epochs.

In []:

```
# 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 which prints the loss from the last batch of data for every 10th epoch, to track the progress of training. `loss.item` returns the actual value stored in the loss tensor.

Let's train the model for 100 epochs.

In []:

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

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

In []:

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

In []:

```
# Compare with targets
targets
```

Indeed, the predictions are quite close to our targets, and now we have a fairly good model to predict crop yields for apples and oranges by looking at the average temperature, rainfall and humidity in a region.

changes by looking at the average temperature, rainfall and humidity in a region.

Commit and update the notebook

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

```
In [ ]:
```

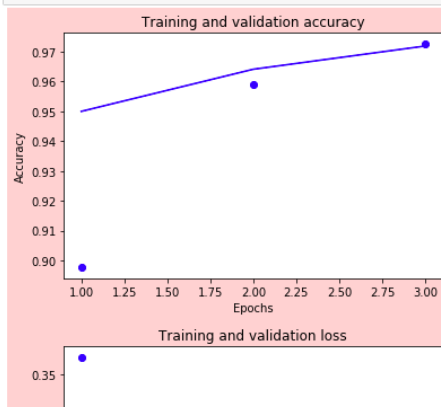
```
import jovian
```

```
In [ ]:
```

```
jovian.commit()
```

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:

```
In [15]: from utils import plot_history
plot_history(history)
```

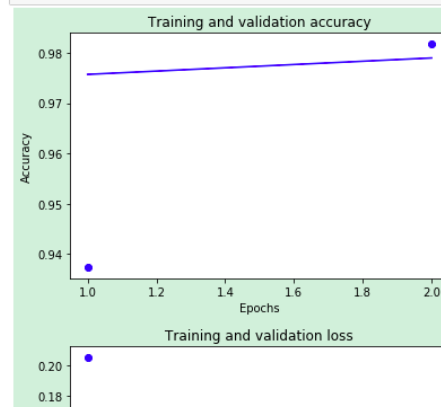


```
In [10]: test_loss, test_acc = model.evaluate(test_images, test_
10000/10000 [=====] - 1s
99us/step
```

```
In [11]: print('Test loss:', test_loss)
print('Test acc:', test_acc)

Test loss: 0.08638658923842013
Test acc: 0.9745
```

```
In [11]: from utils import plot_history
plot_history(history)
```



```
In [12]: test_loss, test_acc = model.evaluate(test_images, test_
10000/10000 [=====] - 2s
227us/step
```

```
In [13]: print('Test loss:', test_loss)
print('Test acc:', test_acc)

Test loss: 0.05926450476180762
Test acc: 0.9808
```

Further Reading

We've covered a lot of ground in this tutorial, including *linear regression* and the *gradient descent* optimization algorithm. Here are a few resources if you'd like to dig deeper into these topics:

- 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: *Logistic regression*.

```
In [ ]:
```