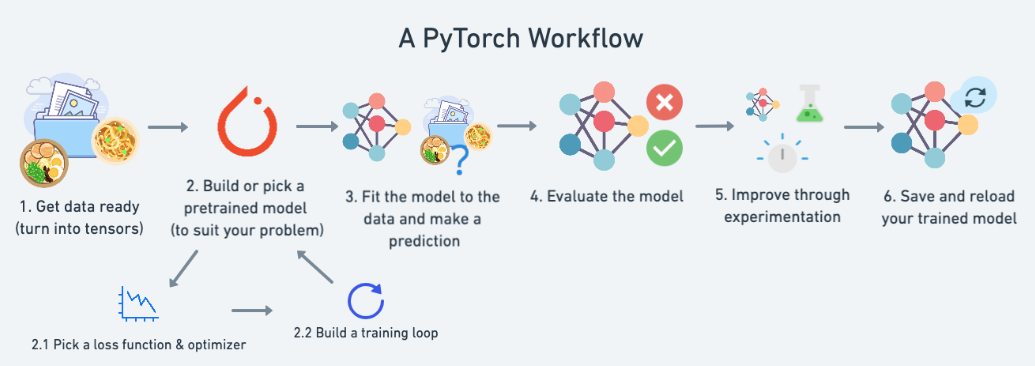
PyTorch Workflow Fundamentals



## **What we're going to cover**

|  |  |
| --- | --- |
| **Topic** | **Contents** |
| 1. Getting data ready | Data can be almost anything but to get started we're going to create a simple straight line |
| 2. Building a model | Here we'll create a model to learn patterns in the data, we'll also choose a loss function, optimizer and build a training loop. |
| 3. Fitting the model to data (training) | We've got data and a model, now let's let the model (try to) find patterns in the (training) data. |
| 4. Making predictions and evaluating a model (inference) | Our model's found patterns in the data, let's compare its findings to the actual (testing) data. |
| 5. Saving and loading a model | You may want to use your model elsewhere, or come back to it later, here we'll cover that. |
| 6. Putting it all together | Let's take all of the above and combine it. |

## **Libraries used for this session**

import torch

from torch import nn # nn contains all of PyTorch's building blocks for neural networks

import matplotlib.pyplot as plt

torch.nn 🡪 library for graphs 🡪 neural networks

link 🡪 <https://pytorch.org/docs/stable/nn.html>

# Data (preparing and loading)

Data can be anything:

1. Excel spread sheet
2. Images
3. Videos
4. Census
5. Text

Machine learning is a game of two parts:

1. Turn your data, whatever it is, into numbers (a representation).
2. Pick or build a model to learn the representation as best as possible.

## How to create data

For this session, we will use Linear Regression formula to prepare data( most common and popular DL/ML algo)

Linear regression algorithm shows a linear relationship between a dependent (y) and one or more independent (y) variables, hence called as linear regression. Since linear regression shows the linear relationship, which means it finds how the value of the dependent variable is changing according to the value of the independent variable.

The linear regression model provides a sloped straight line representing the relationship between the variables. Consider the below image:



General equation 🡪 y = bx + a

b = slope / regression coefficient(weight)

a = constant/ random error in DL (bias)

*# Create \*known\* parameters*

weight **=** 0.7

bias **=** 0.3

*# Create data*

start **=** 0

end **=** 1

step **=** 0.02

X **=** torch**.**arange(start, end, step)**.**unsqueeze(dim**=**1)

y **=** weight **\*** X **+** bias

X[:10], y[:10]

## Split data into training and test sets

One of most important steps in a machine learning project is creating a training and test set (and when required, a validation set).

Each split of the dataset serves a specific purpose:

|  |  |  |  |
| --- | --- | --- | --- |
| **Training set** | The model learns from this data (like the course materials you study during the semester). | ~60-80% | Always |
| **Validation set** | The model gets tuned on this data (like the practice exam you take before the final exam). | ~10-20% | Often but not always |
| **Testing set** | The model gets evaluated on this data to test what it has learned (like the final exam you take at the end of the semester). | ~10-20% |  |

# Create train/test split

train\_split = int(0.8 \* len(X)) # 80% of data used for training set, 20% for testing

X\_train, y\_train = X[:train\_split], y[:train\_split]

X\_test, y\_test = X[train\_split:], y[train\_split:]

len(X\_train), len(y\_train), len(X\_test), len(y\_test)

we've got 40 samples for training (X\_train & y\_train) and 10 samples for testing (X\_test & y\_test).

Model is going to use X\_train and Y\_Train and learn to predict the remaining samples

**Once we have the data sets ready , the next step is to visualize data. In the below function , we have one goal ,**

1. **Train the model using training data ( x\_train , y\_train) 🡪 ( train\_data , train\_labels)**
2. **And then for evaluation , we are going to make this model to predict y\_test using x\_test data set as input**
3. **And then we compare result of step2 , and evaluate how good model predictions are**
   1. **Ie. Prediction of y\_test( on step2 ) vs actual values of test data set**

## Prepare simple function to visualize data(use matplotlib)

**def** plot\_predictions(train\_data**=**X\_train,

train\_labels**=**y\_train,

test\_data**=**X\_test,

test\_labels**=**y\_test,

predictions**=None**):

"""

Plots training data, test data and compares predictions.

"""

plt**.**figure(figsize**=**(10, 7))

*# Plot training data in blue, c = color , s= size of array*

plt**.**scatter(train\_data, train\_labels, c**=**"b", s**=**4, label**=**"Training data")

*# Plot test data in green*

plt**.**scatter(test\_data, test\_labels, c**=**"g", s**=**4, label**=**"Testing data")

**if** predictions **is** **not** **None**:

*# Plot the predictions in red (predictions were made on the test data)*

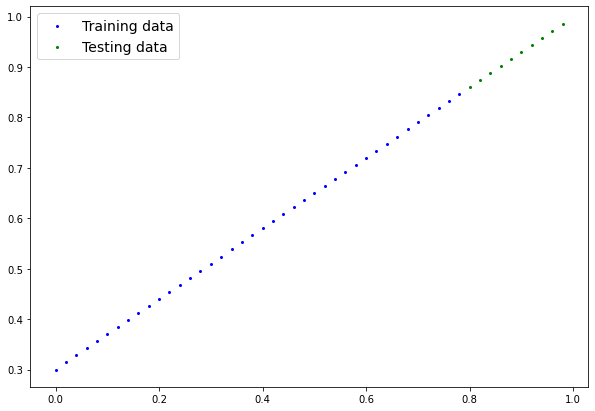
plt**.**scatter(test\_data, predictions, c**=**"r", s**=**4, label**=**"Predictions")

*# Show the legend*

plt**.**legend(prop**=**{"size": 14});

**output**

1. No predictions no red dot ( only blue as training data and green as test data)
2. For prediction , we are going to use the model to find out y( green values) for x( green value) and compare how accurate the model results are.
   1. Below is the ideal output
      1. Ie. If 100% accurate , red dots will overlap green dots below but if not there will be deviation



# Build Model

Use the blue dots to predict green dots

## Linear Regression Model

Y = (weight)x + Bias

b = slope / regression coefficient(weight)

a = constant/ random error in DL (bias)

nn.Module -> base class for neural networks ( building blocks for all models in pytorch)

5:01:08

# Create a Linear Regression model class

class LinearRegressionModel(nn.Module): # <- almost everything in PyTorch is a nn.Module (think of this as neural network building blocks)

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.weights = nn.Parameter(torch.randn(1, # <- start with random weights (this will get adjusted as the model learns)

                                                dtype=torch.float), # <- PyTorch float32 by default

                                   requires\_grad=True) # <- can we update this value with gradient descent?)

        self.bias = nn.Parameter(torch.randn(1, # <- start with random bias (this will get adjusted as the model learns)

                                            dtype=torch.float), # <- PyTorch float32 by default

                                requires\_grad=True) # <- can we update this value with gradient descent?))

    # Forward defines the computation in the model

    def forward(self, x: torch.Tensor) -> torch.Tensor: # <- "x" is the input data (e.g. training/testing features)

        return self.weights \* x + self.bias # <- this is the linear regression formula (**Y = (weight)x + Bias**)