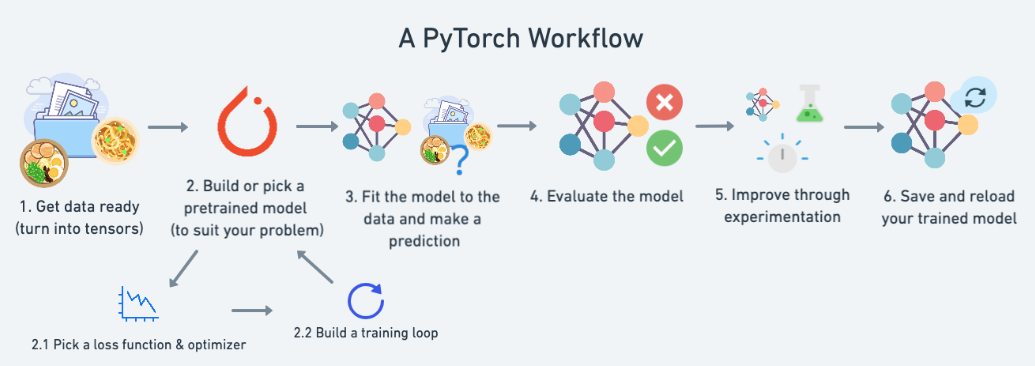
PyTorch Workflow Fundamentals



## **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)*

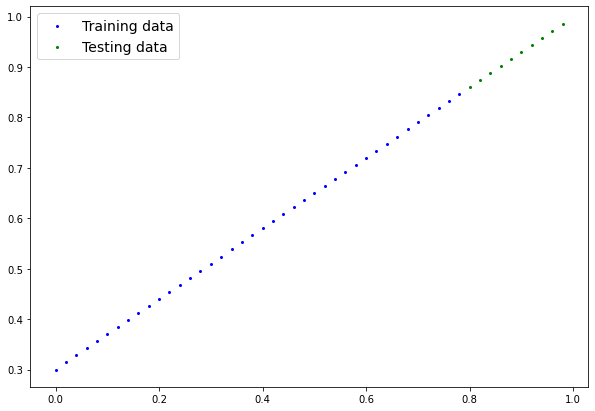
plt**.**scatter(test\_data, predictions, c**=**"r", s**=**4, label**=**"Predictions")

*# 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). We are just creating a sub class of neural networks ( our own neural network with weight and bias defined)

forward method 🡪 is needed for any subclass for neural network. This function is the operation which this model will do.

Doc Link 🡪 <https://pytorch.org/docs/stable/generated/torch.nn.Module.html>

# 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**)

**what the model is doing??**

1. Start with random values (weight and bias)
2. Look at the training data and adjust the random values to better repr( or close to) ideal values.( weight and bias values we used to create the data)
   1. How does it adjust itself??
      1. **Through 2 main algos**
         1. Gradient descent
         2. Back propagation (requires grad = True)

# Pytorch Model building essentials

**Pytorch cheat sheet**

<https://pytorch.org/tutorials/beginner/ptcheat.html>

|  |  |
| --- | --- |
| **PyTorch module** | **What does it do?** |
| [torch.nn](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fnn.html&link_redirector=1) | Contains all of the building blocks for computational graphs (essentially a series of computations executed in a particular way). |
| [torch.nn.Parameter](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.parameter.Parameter.html%23parameter&link_redirector=1) | [Stores tensors that can be used with nn.Module. If requires\_grad=True gradients (used for updating model parameters via gradient descent) are calculated automatically, this is often referred to as "autograd".](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fml-cheatsheet.readthedocs.io%2Fen%2Flatest%2Fgradient_descent.html&link_redirector=1) |
| [torch.nn.Module](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fgenerated%2Ftorch.nn.Module.html%23torch.nn.Module&link_redirector=1) | The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass nn.Module. Requires a forward() method be implemented. |
| [torch.optim](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Foptim.html&link_redirector=1) | Contains various optimization algorithms (these tell the model parameters stored in nn.Parameter how to best change to improve gradient descent and in turn reduce the loss). |
| def forward() | All nn.Module subclasses require a forward() method, this defines the computation that will take place on the data passed to the particular nn.Module (e.g. the linear regression formula above). |

5:21:21

What each step in model building , and which library is used

**Checking the contents of the model**

We use .parameters() function for checking the contents of the model.

Use random seed before instantiate the model.

**Why use random seed**?? 🡪 cos if we don’t use , computer / machine is going to give different results every time we run the model

If we use random seed , we are forcing the machine , to provide same random no. every time.

# Set manual seed since nn.Parameter are randomly initialzied

torch.manual\_seed(42)

# Create an instance of the model (this is a subclass of nn.Module that contains nn.Parameter(s))

model\_0 = LinearRegressionModel()

# Check the nn.Parameter(s) within the nn.Module subclass we created

list(model\_0.parameters())

# List named parameters

model\_0.state\_dict()

Essentially we want to start from random parameters and get the model to update them towards parameters that fit our data best (the hardcoded weight and bias values we set when creating our straight line data).

What it means is 🡪 goal of our model is , the above weight and bias ( random ones) need to match the hardcoded ones( 0.7 and 0.3)

Weight = 0.7

Bias = 0.3

So this is the principle of deep learning 🡪 start with random values and then move these as close to ideal values.

# Making predictions using torch.inference\_mode()

## making predictions using torch.inference\_mode()

To check this we can pass it the test data X\_test to see how closely it predicts y\_test.

When we pass data to our model, it'll go through the model's forward() method and produce a result using the computation we've defined.

Let's make some predictions.

# Make predictions with model

with torch.inference\_mode():

    y\_preds = model\_0(X\_test)

# Note: in older PyTorch code you might also see torch.no\_grad()

# with torch.no\_grad():

#   y\_preds = model\_0(X\_test)

Inference mode is used as a context manager,

**Why its being used??**

1. is used when using a model for inference (making predictions).
   1. Inference mode switch off lot of things like gradient tracking , which is necessary for training \*but\* not for inference.
2. Also when inference mode is used , data processes much faster , since the model doesn’t need to remember or track the gradients of different parameters when calculating values.
   1. Pytorch keeps track of these things automatically when model is executed.
3. Also when we are doing inference , we just need our model to predict , ( we are not training the model or updating the model)

Useful link 🡪 <https://twitter.com/PyTorch/status/1437838231505096708?lang=en>