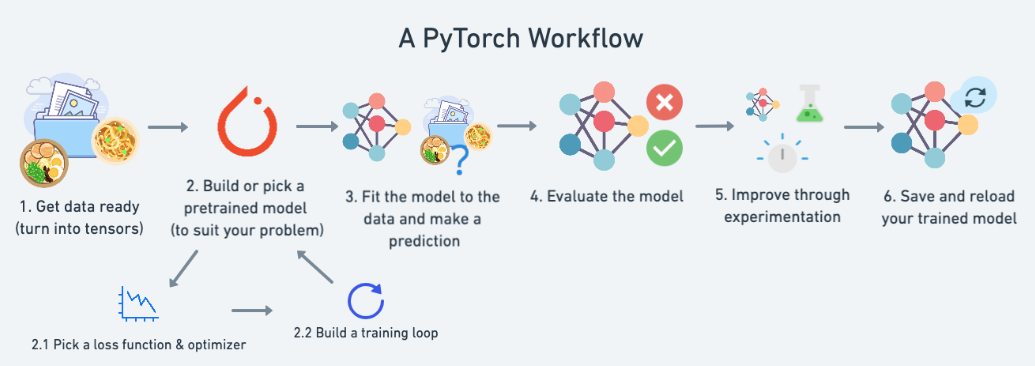
PyTorch Workflow Fundamentals



## **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 = mx + c

m = slope / regression coefficient(weight)

c = 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)*

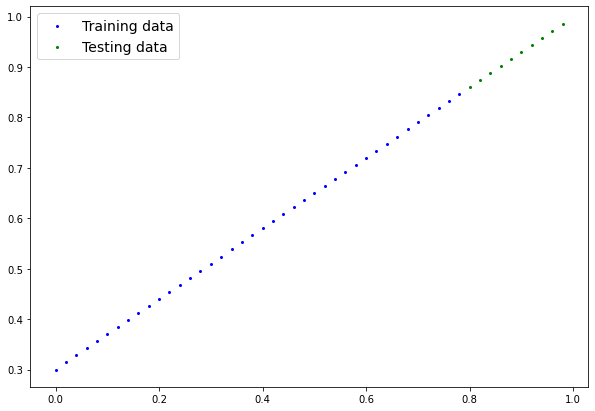
plt**.**scatter(test\_data, predictions, c**=**"r", s**=**4, label**=**"Predictions")

*# 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



# 2.a.Build Model

Use the blue dots to predict green dots

## Linear Regression Model

Y = (weight)x + Bias

weight = slope / regression coefficient

bias= constant/ random error in DL

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)

# b.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). |

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.

# 2.cMaking 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,

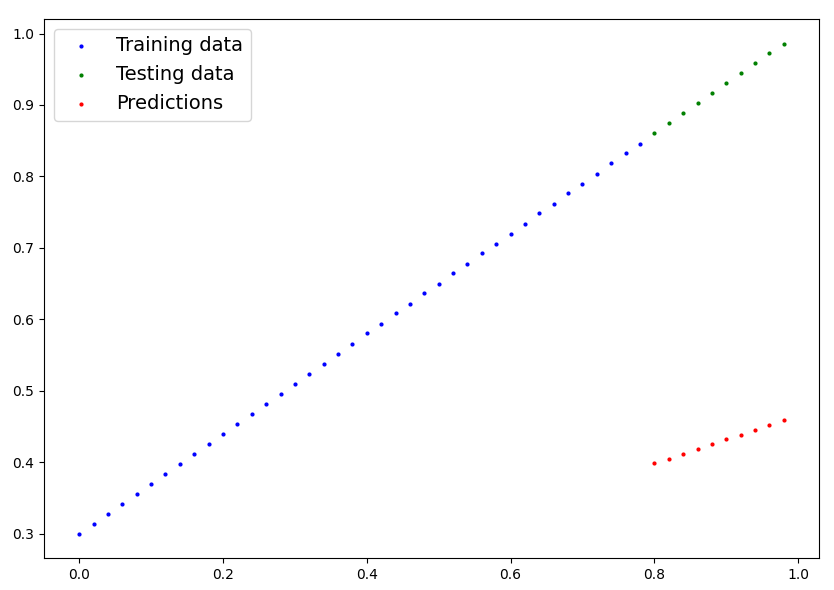
**Why its being used??**

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>

**Model prediction graph with random variables**

Goal here is 🡪 red dots needs to overlap the green dots or as close as green dots in the graph



# 2.d.Training Model with pytorch

Whole idea of training model is to move from \*unknown \* params( random) to \*known\* params. Or in poor repr of data to better repr of data ,if you hard code weight and bias to 0.7 |0.3, then red will overlap green

One way to measure how wrong / poor model predictions are to use LOSS function

## **Creating a loss function and optimizer in PyTorch**

**Doc 🡪** [**https://pytorch.org/docs/stable/nn.html#loss-functions**](https://pytorch.org/docs/stable/nn.html#loss-functions)

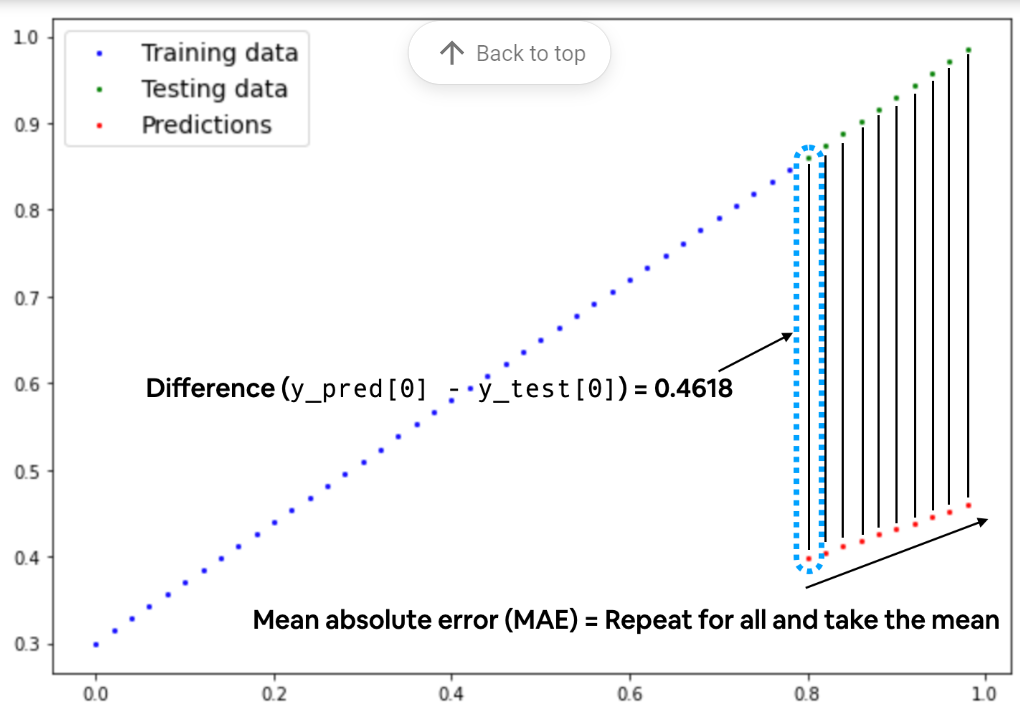
|  |  |  |  |
| --- | --- | --- | --- |
| **Function** | **What does it do?** | **Where does it live in PyTorch?** | **Common values** |
| **Loss function** | Measures how wrong your models predictions (e.g. y\_preds) are compared to the truth labels (e.g. y\_test). Lower the better. | [PyTorch has plenty of built-in loss functions in torch.nn.](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Fnn.html%23loss-functions) | Mean absolute error (MAE) for regression problems (torch.nn.L1Loss()). Binary cross entropy for binary classification problems (torch.nn.BCELoss()). |
| **Optimizer** | Tells your model how to update its internal parameters to best lower the loss. | [You can find various optimization function implementations in torch.optim.](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpytorch.org%2Fdocs%2Fstable%2Foptim.html) | Stochastic gradient descent (torch.optim.SGD()). Adam optimizer (torch.optim.Adam()). |

Loss function 🡪 measure the distance between predicted and actual values (can be used for simple 2 params here or for models where million of params are present)

Optimizer 🡪 takes into account the loss of model and adjusts the model params (eq : weight and bias)

Depending on what kind of problem you're working on will depend on what loss function and what optimizer you use.

However, there are some common values, that are known to work well such as the SGD (stochastic gradient descent) or Adam optimizer. And the MAE (mean absolute error) loss function for regression problems (predicting a number) or binary cross entropy loss function for classification problems (predicting one thing or another).



**What is MAE??**

1) MAE is Mean absolute error:

* Mean 🡪 average
* Absolute 🡪 modulus 🡪 |-ve or +ve|
* Error 🡪 deviation from what is expected

And we'll use SGD, torch.optim.SGD(params, lr) where:

* params is the target model parameters you'd like to optimize (e.g. the weights and bias values we randomly set before).
* lr is the **learning** rate you'd like the optimizer to update the parameters at, higher means the optimizer will try larger updates (these can sometimes be too large and the optimizer will fail to work), lower means the optimizer will try smaller updates (these can sometimes be too small and the optimizer will take too long to find the ideal values). The learning rate is considered a **hyperparameter** (because it's set by a machine learning engineer). Common starting values for the learning rate are 0.01, 0.001, 0.0001, however, these can also be adjusted over time (this is called [learning rate scheduling](https://pytorch.org/docs/stable/optim.html#how-to-adjust-learning-rate)).

**Setup of Loss Function and optimizer**

torch.optim 🡪 <https://pytorch.org/docs/stable/optim.html>

torch.nn.loss 🡪 <https://pytorch.org/docs/stable/nn.html#loss-functions>

# Create the loss function

loss\_fn = nn.L1Loss() # MAE loss is same as L1Loss

# Create the optimizer

optimizer = torch.optim.SGD(params=model\_0.parameters(), # parameters of target model to optimize

                            lr=0.01) # learning rate (how much the optimizer should change parameters at each step, higher=more (less stable), lower=less (might take a long time))

## Build a training / Optimization Loop in Pytorch

**##building a training and testing loop**

The training loop involves the model going through the training data and learning the relationships between the features and labels.

The testing loop involves going through the testing data and evaluating how good the patterns are that the model learned on the training data (the model never see's the testing data during training).

Each of these is called a "loop" because we want our model to look (loop through) at each sample in each dataset.

**Few things we perform in training loop**

0) Loop through the data 🡪 term epoch 🡪 An epoch in machine learning means one complete pass of the training dataset through the algorithm

1) forward pass (data moves from input layer 🡪 output layer (neural network) 🡪 forward propagation) 🡪 make predictions

2) calculate the Loss ( compare forward pass predictions with expected value | ground truth) 🡪 evaluate how the model is doing.

3) optimizer zero grad

4) Loss backwards 🡪 move backwards through the network to calculate the gradients of each of the params of our models w.r.t the loss ( \*\* back propagation\*\*)

5) optimizer step 🡪 use the optimizer to adjust our model’s parameters to try and improve the loss. ( \*\* gradient descent\*\* )

|  |  |  |  |
| --- | --- | --- | --- |
| **Number** | **Step name** | **What does it do?** | **Code example** |
| 1 | Forward pass | The model goes through all of the training data once, performing its forward() function calculations. | model(x\_train) |
| 2 | Calculate the loss | The model's outputs (predictions) are compared to the ground truth and evaluated to see how wrong they are. | loss = loss\_fn(y\_pred, y\_train) |
| 3 | Zero gradients | The optimizers gradients are set to zero (they are accumulated by default) so they can be recalculated for the specific training step. | optimizer.zero\_grad() |
| 4 | Perform backpropagation on the loss | Computes the gradient of the loss with respect for every model parameter to be updated (each parameter with requires grad=True). This is known as **backpropagation**, hence "backwards". | loss.backward() |
| 5 | Update the optimizer (**gradient descent**) | Update the parameters with requires\_grad=True with respect to the loss gradients in order to improve them. | optimizer.step() |

**What is a gradient??**

It is the slope of a function.

Higher gradient means 🡪 model will learn faster (high efficiency)

Lower gradient 🡪 (slow learning)

0 gradient 🡪 model stops learning

Y = **m**x + c

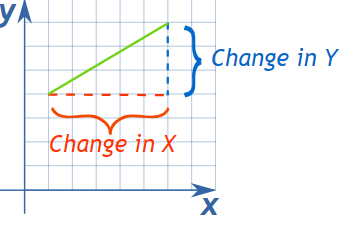
## **Example of Gradient Descent**

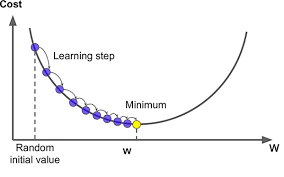
Let’s say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?

The best way is to observe the ground and find where the land descends. From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

**Loss here is the top of the hill (where we are initially and when we reach the ground loss is 0, which we are trying to predict, so we have travelled over the slope of the hill.**

**For simple cases, 2 variables , its easy to visualize but when we have more params ( higher dimension it becomes difficult). Pytorch calcs gradient behind the scenes by something called as pytorch autograd**



 as you see in this fig 🡪 initially the slope( change in Y / change in X) is too steep and then gradually becomes small , how the slope becomes small ?? this is because of optimizer and learning rate .

**For ex: LR can be high in the starting ( that is fall high in the initial loop) and then reduces gradually as we are reaching the bottom , to make sure we don’t cross the other side.**

