PyTorch Notes

**00. PyTorch Fundamentals**

**Tensors**

Tensors are a fundamental building block of machine learning and represents data in a numerical way. For example, an image tensor is represented with the shape [3, 244, 244] corresponding to color channels, width, and height.

Creating Scalars

# Scalar

scalar = torch.tensor(7)

scalar.ndim

scalar.item()

Creating Vectors

# Vector

vector = torch.tensor([7, 7])

vector.ndim

vector.shape

Creating Matrices

# Matrix

MATRIX = torch.tensor([[7, 8], [9, 10]])

MATRIX.ndim

MATRIX.shape

Creating Tensors

# Tensor

TENSOR = torch.tensor([[[1, 2, 3],

[3, 6, 9],

[2, 4, 5]]])

TENSOR.ndim

TENSOR.shape

**Random Tensors**

When building machine learning models with PyTorch, it's rare you'll create tensors by hand (like what we've being doing).

Instead, a machine learning model often starts out with large random tensors of numbers and adjusts these random numbers as it works through data to better represent it.

In essence:

*Start with random numbers -> look at data -> update random numbers -> look at data -> update random numbers...*

Random Tensors

# Create random tensor of size (244, 244, 3)

random\_image\_size\_tensor = torch.rand(size=(244, 244, 3))

random\_image\_size\_tensor.shap, random\_image\_size\_tensor.ndim

**Zeros**

Sometimes we want to set this tensor to zeros and ones effectively masking which weighted parameters the model should not learn.

Zeros and Ones Tensors

zeros = torch.zeros(size=(3, 4))

zeros, zeros.dtype

ones = torch.ones(size=(3, 4))

ones, ones.dtype

Range Tensors

zeros\_to\_ten = torch.arange(start=0, end=10, step=1)

zeros\_to\_ten

Note: sometimes you’ll want one tensor of a certain type with the same shape as another tensor. To do so you can use torch.zeros\_like(input tensor) or torch.ones\_like(input tensor).

**Tensor Datatypes**

There are many different tensor datatypes and some are specific to CPU and some are better for GPU.

Types include: torch.cuda, torch.float32, torch.float, torch.float16, torch.half, torch.float64, torch.double

Of course, the higher the precision value, the more detail and data used to express a number. This matters in deep learning and numerical computing because of the shear volumes of operations. The more detail in each number, the more you have to compute. So, one would say to use lower precision datatypes since they are faster to compute. However, this sacrifices some performance on evaluation metrics like accuracy.

Range Tensors

# Default datatype for tensors is float32

float\_32\_tensor = torch.tensor([3.0, 6.0, 9.0],

dtype=None,

device=None,

requires\_grad=False)

float\_32\_tensor.shape, float\_32\_tensor.dtype, float\_32\_tensor.device

Tensor Properties

# Create a tensor

some\_tensor = torch.rand(3, 4)

# Find out details about it

print(some\_tensor)

print(f"Shape of tensor: {some\_tensor.shape}")

print(f"Datatype of tensor: {some\_tensor.dtype}")

print(f"Device tensor is stored on: {some\_tensor.device}") # will default to CPU

**Tensor Operations**

tensor = torch.tensor([1, 2, 3])

tensor + 10

tensor \* 10

torch.multiply(tensor, 10)

torch.matmul(tensor, tensor)

torch.mm(tensor, tensor)

# element wise multiplication

Tensor \* tensor

The torch.nn.Linear() module (we'll see this in action later on), also known as a feed-forward layer or fully connected layer, implements a matrix multiplication between an input x and a weights matrix A.

𝑦 = 𝑥 ⋅ 𝐴𝑇 + 𝑏

**Feed Forward Linear Layer**

# Since the linear layer starts with a random weight matrix, let's make it reproducible

torch.manual\_seed(42)

# This uses matrix multiplication

linear = torch.nn.Linear(in\_features=2, # in\_features = matches inner dimension of input

out\_features=6) # out\_features = describes outer value

x = tensor\_A

output = linear(x)

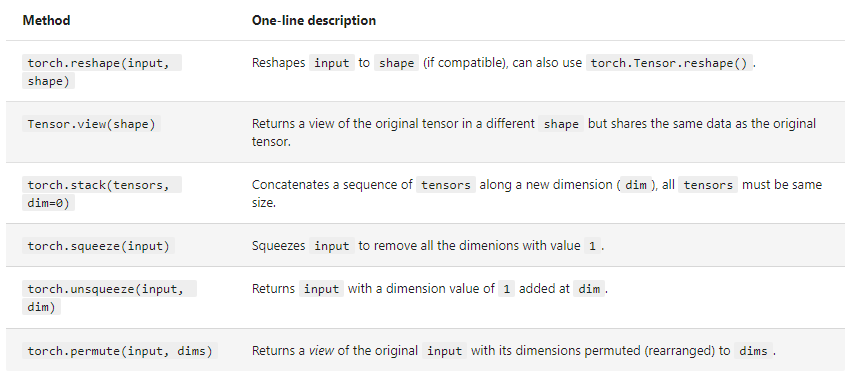
print(f"Input shape: {x.shape}\n")

print(f"Output:\n{output}\n\nOutput shape: {output.shape}")

**Converting Tensor Type**

tensor\_float16 = tensor.type(torch.float16)

**Reshape, Stacking, Squeezing, and Unsqueezing**

****

**PyTorch Tensors & NumPy**

# NumPy array to tensor

import torch

import numpy as np

array = np.arange(1.0, 8.0)

tensor = torch.from\_numpy(array)

array, tensor

# Tensor to NumPy array

tensor = torch.ones(7) # create a tensor of ones with dtype=float32

numpy\_tensor = tensor.numpy() # will be dtype=float32 unless changed

tensor, numpy\_tensor

**Reproducibility**

As you learn more about neural networks and machine learning, you'll start to discover how much randomness plays a part.

import torch

import random

# # Set the random seed

RANDOM\_SEED=42

torch.manual\_seed(seed=RANDOM\_SEED)

random\_tensor\_C = torch.rand(3, 4)

# Have to reset the seed every time a new rand() is called

# Without this, tensor\_D would be different to tensor\_C

torch.random.manual\_seed(seed=RANDOM\_SEED) # try commenting this line out and seeing what happens

random\_tensor\_D = torch.rand(3, 4)

print(f"Tensor C:\n{random\_tensor\_C}\n")

print(f"Tensor D:\n{random\_tensor\_D}\n")

print(f"Does Tensor C equal Tensor D? (anywhere)")

random\_tensor\_C == random\_tensor\_D

**Using Your GPU**

# Check for GPU

import torch

torch.cuda.is\_available()

device = “cuda” if torch.cuda.is\_available() else “cpu”

# Count number of devices

torch.cuda.device\_count()

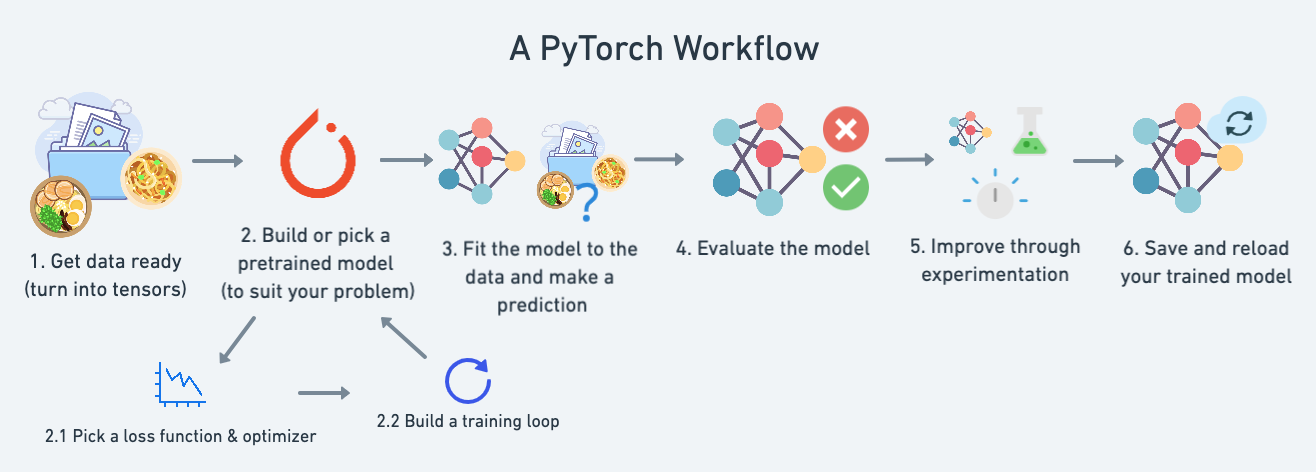
tensor = torch.tensor([1, 2, 3])

tensor\_on\_gpu = tensor.to(device)

# Note if tensor is on GPU, it can’t be transformed it to NumPy. To move back.

tensor\_back\_to\_cpu = torch\_on\_gpu.cpu()

**01. PyTorch Workflow Fundamentals**

C

**1. Creating Data**

# 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

We need to split the data up into training set, validation set, testing set.

# Create train/test split

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

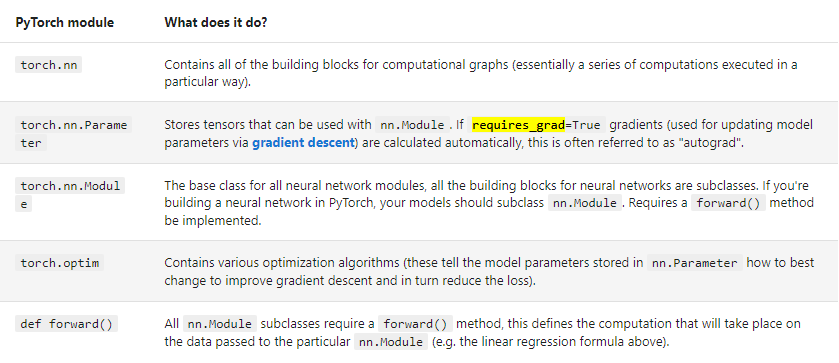
X\_train, y\_train = X[:train\_split], y[:train\_split]

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

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

**2. Building the Model**

*torch.nn*



# Create a Linear Regression model class

class LinearRegressionModel(nn.Module):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.weights = nn.Parameter(torch.randn(1, dtype=torch.float), requires\_grad=True)

self.bias = nn.Parameter(torch.randn(1, dtype=torch.float), requires\_grad=True)

def forward(self, x: torch.Tensor) -> torch.Tensor:

return self.weights \* x + self.bias # <- this is the linear regression formula (y = m\*x + b)

torch.manual\_seed(42)

model\_0 = LinearRegressionMode()

list(model\_0.parameters())

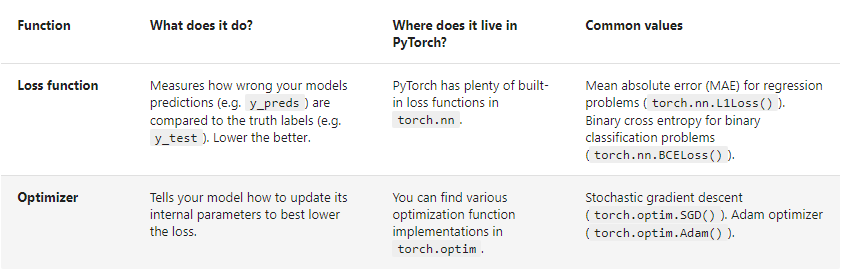
model\_0.state\_dict()

# Make predictions with model

with torch.inference\_mode():

y\_preds = model\_0(X\_test)

**3. Train Model**



Let’s use stochastic gradient descent as our optimizer and the mean absolute error as our loss function.

# 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(), 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))

*Training Loop*

Now that we defined our model, loss function, and optimizer, we now want to create a training loop to help the model determine the relationship between the features and labels.

Forward pass -> calculate loss -> zero gradients -> backpropagation on loss -> update optimizer

\*Note: Back propagation is the process or computing the gradient of the loss with respect to every model parameter to be updated.

*Testing Loop*

Forward pass -> calculate loss -> calculate evaluation metrics for personal analysis

Putting it all together:

torch.manual\_seed(42)

epochs = 100

train\_loss\_values = []

test\_loss\_values = []

epoch\_count = []

for epoch in range(epochs):

### Training

# Put model in training mode (this is the default state of a model)

model\_0.train()

# 1. Forward pass on train data using the forward() method inside

y\_pred = model\_0(X\_train)

# print(y\_pred)

# 2. Calculate the loss (how different are our models predictions to the ground truth)

loss = loss\_fn(y\_pred, y\_train)

optimizer.zero\_grad()

loss.backward()

optimizer.step()

### Testing

# Put the model in evaluation mode

model\_0.eval()

with torch.inference\_mode():

test\_pred = model\_0(X\_test)

test\_loss = loss\_fn(test\_pred, y\_test.type(torch.float))

if epoch % 10 == 0:

epoch\_count.append(epoch)

train\_loss\_values.append(loss.detach().numpy())

test\_loss\_values.append(test\_loss.detach().numpy())

print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test\_loss} ")

**4. Make Predictions**

model\_0.eval()

# 2. Setup the inference mode context manager

with torch.inference\_mode():

# 3. Make sure the calculations are done with the model and data on the same device

# in our case, we haven't setup device-agnostic code yet so our data and model are

# on the CPU by default.

# model\_0.to(device)

# X\_test = X\_test.to(device)

y\_preds = model\_0(X\_test)

**5. Saving and Loading a PyTorch Model**

If you’ve trained a PyTorch model, chances are you’ll want to save it and export it somewhere:

from pathlib import Path

MODEL\_PATH = Path("models")

MODEL\_PATH.mkdir(parents=True, exist\_ok=True)

MODEL\_NAME = "01\_pytorch\_workflow\_model\_0.pth"

MODEL\_SAVE\_PATH = MODEL\_PATH / MODEL\_NAME

torch.save(obj=model\_0.state\_dict(), f=MODEL\_SAVE\_PATH)

# Note we only saved the model’s parameters not the model itself

# Instantiate a new instance of our model (this will be instantiated with random weights)

loaded\_model\_0 = LinearRegressionModel()

# Load the state\_dict of our saved model (this will update the new instance of our model with trained weights)

loaded\_model\_0.load\_state\_dict(torch.load(f=MODEL\_SAVE\_PATH))