THE UNIVERSITY OF TEXAS AT DALLAS

PyTorch Tutorial

CS 6384 Computer Vision
TA Shijian Deng
Department of Computer Science

Contents

PyTorch

1. Why PyTorch

A zoo of frameworks!

Caffe
(UC Berkeley) → Caffe2
(Facebook)
mostly features absorbed by PyTorch ↓

Torch
(NYU / Facebook) → PyTorch
(Facebook)

Theano
(U Montreal) → TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research infrastructure to PyTorch

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

JAX
(Google)

And others...

Source: CS231N by Fei-Fei Li, Ranjay Krishna, Danfei Xu

1. Why PyTorch (Cont'd)

Wanna build deep neural networks just like playing Lego?

PyTorch is all you need!

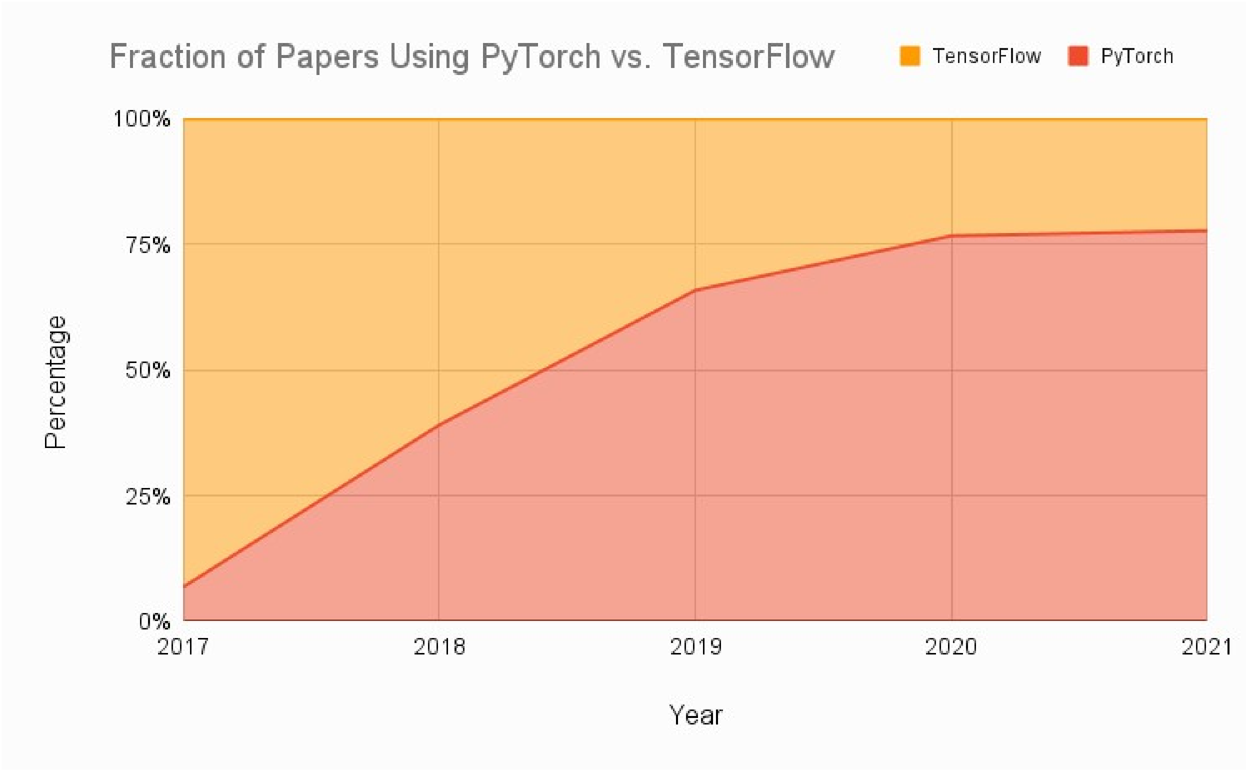
(1) Pythonic

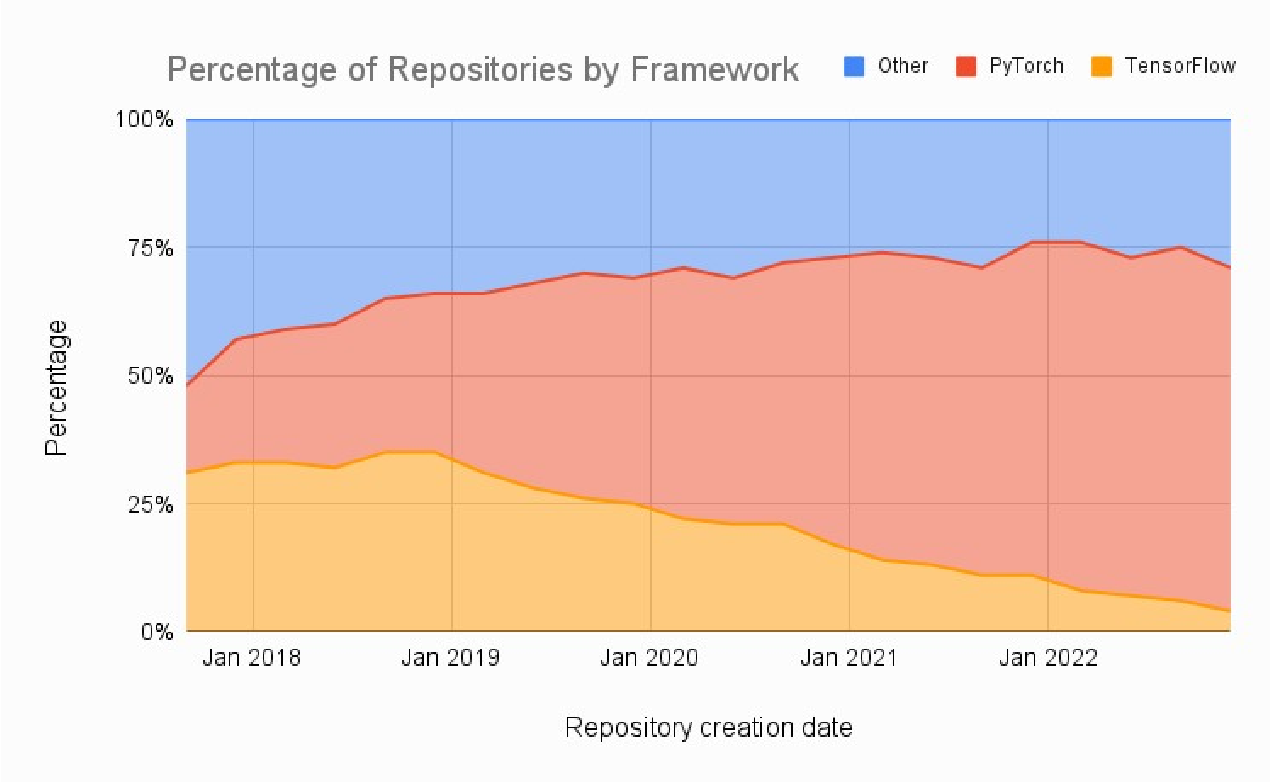
(2) Dynamic Graphs

(3) Ecosystems

Source: CS231N by Fei-Fei Li & Andrej Karpathy & Justin Johnson

1. Why PyTorch (Cont'd)

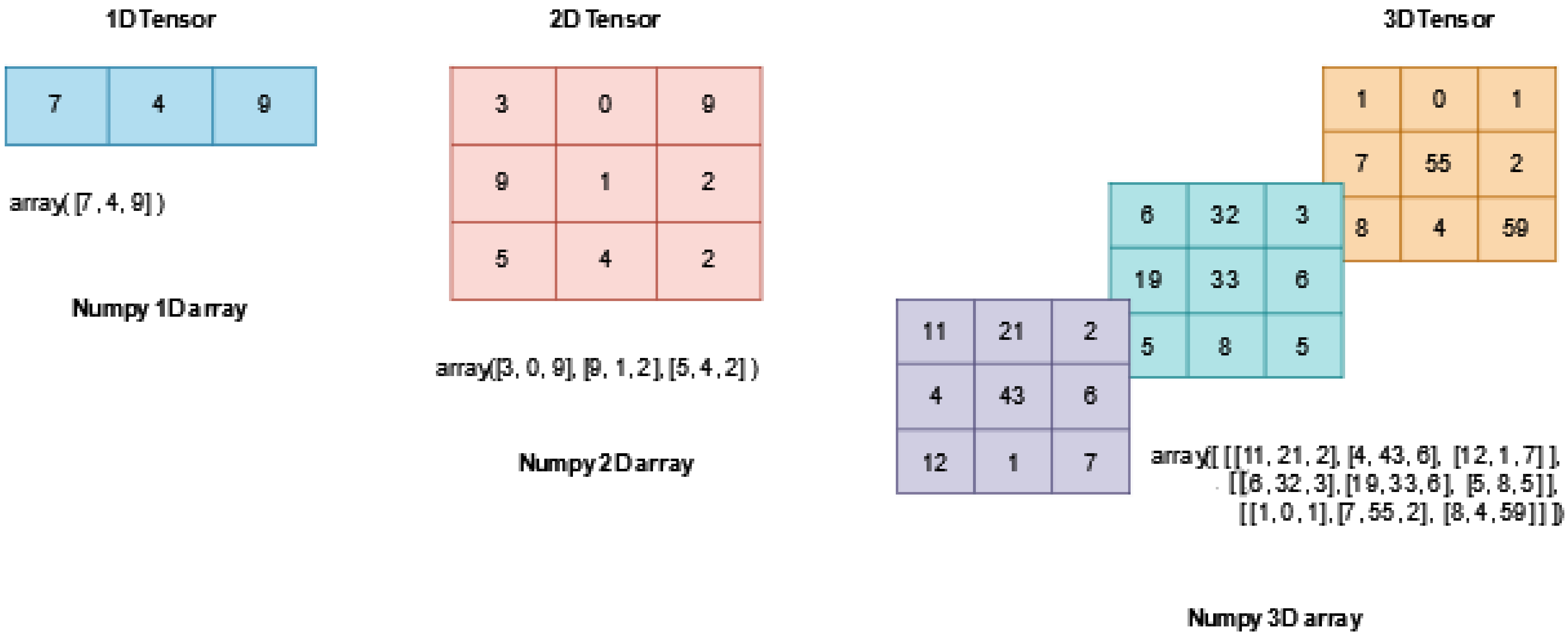




Source: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/

Tensors

2 Tensors



Source: Microsoft Learn

2.1 Initializing a Tensor

Directly from data

Tensors can be created directly from data. The data type is automatically inferred.

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

From a NumPy array

Tensors can be created from NumPy arrays (and vice versa - see Bridge with NumPy).

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

2.1 Initializing a Tensor (Cont'd)

From another tensor:

The new tensor retains the properties (shape, datatype) of the argument tensor, unless explicitly overridden.

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Out:

```
Ones Tensor:
 tensor([[1, 1],
        [1, 1]])

Random Tensor:
 tensor([[0.5788, 0.2201],
        [0.2905, 0.9572]])
```

2.2 Attributes of a Tensor

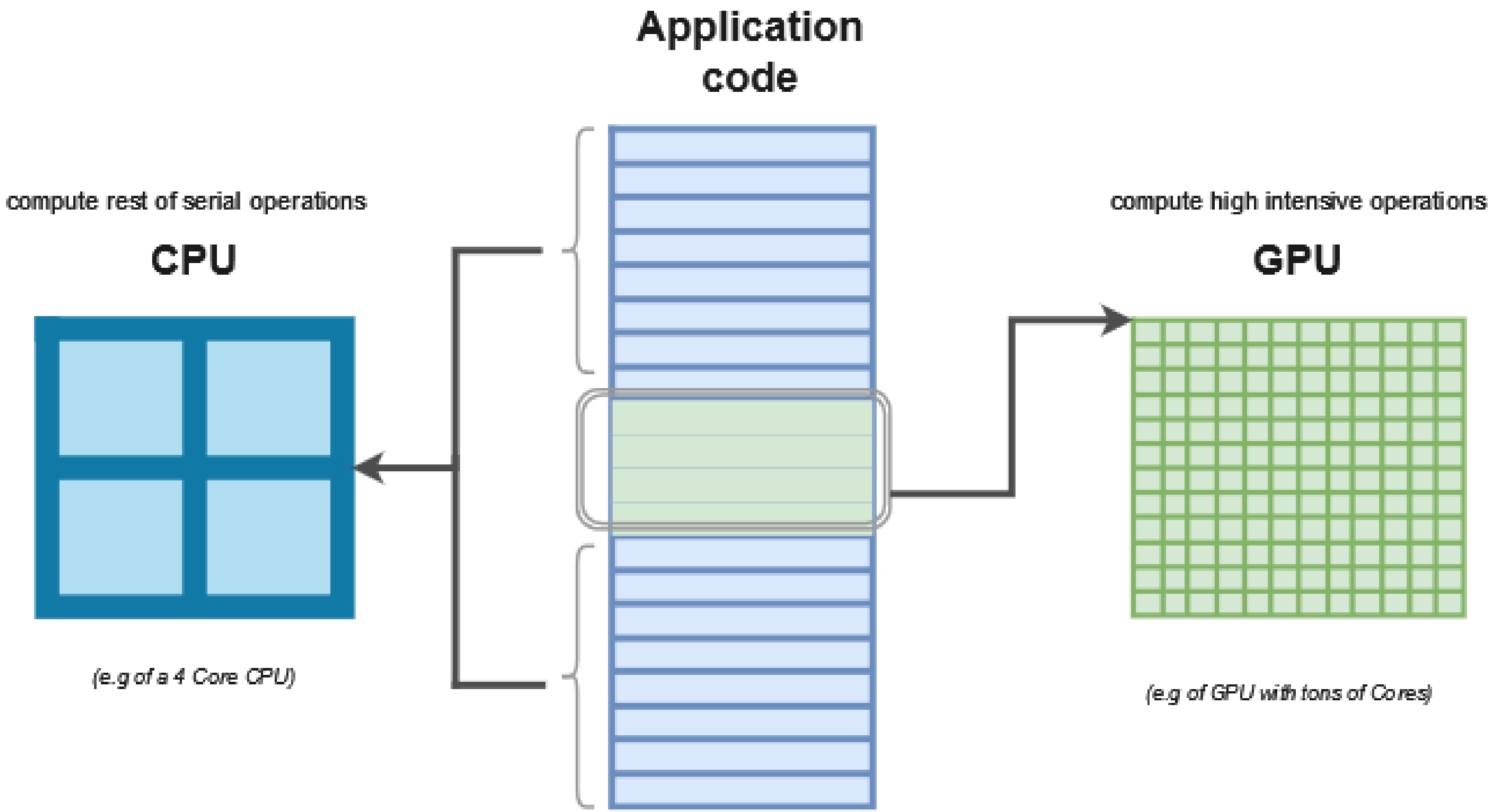
Tensor attributes describe their shape, datatype, and the device on which they are stored.

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Out:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

2.3 Operation on Tensors



Source: Microsoft Learn

2.3 Operation on Tensors (Cont'd)

By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using `.to` method (after checking for GPU availability). Keep in mind that copying large tensors across devices can be expensive in terms of time and memory!

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
```

2.3 Operation on Tensors (Cont'd)

Standard numpy-like indexing and slicing:

```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
print(tensor)
```

Out:

```
First row: tensor([1., 1., 1., 1.])
First column: tensor([1., 1., 1., 1.])
Last column: tensor([1., 1., 1., 1.])
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])
```

2.3 Operation on Tensors (Cont'd)

Joining tensors You can use `torch.cat` to concatenate a sequence of tensors along a given dimension. See also torch.stack, another tensor joining op that is subtly different from `torch.cat`.

```
t1 = torch.cat([tensor, tensor, tensor], dim=1)
print(t1)
```

Out:

```
tensor([[1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.],
        [1., 0., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1.]])
```

2.3 Operation on Tensors (Cont'd)

Arithmetic operations

```
# This computes the matrix multiplication between two tensors. y1, y2, y3 will
have the same value
# ``tensor.T`` returns the transpose of a tensor
y1 = tensor @ tensor.T
y2 = tensor.matmul(tensor.T)

y3 = torch.rand_like(y1)
torch.matmul(tensor, tensor.T, out=y3)


# This computes the element-wise product. z1, z2, z3 will have the same value
z1 = tensor * tensor
z2 = tensor.mul(tensor)

z3 = torch.rand_like(tensor)
torch.mul(tensor, tensor, out=z3)
```

2.3 Operation on Tensors (Cont'd)

Single-element tensors If you have a one-element tensor, for example by aggregating all values of a tensor into one value, you can convert it to a Python numerical value using `item()`:

```
agg = tensor.sum()
agg_item = agg.item()
print(agg_item, type(agg_item))
```

Out:

```
12.0 <class 'float'>
```

2.3 Operation on Tensors (Cont'd)

In-place operations Operations that store the result into the operand are called in-place. They are denoted by a `_` suffix. For example: `x.copy_(y)`, `x.t_()`, will change `x`.

```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)
```

Out:

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

2.3 Operation on Tensors (Cont'd)

Over 100 tensor operations, including:

arithmetic,
linear algebra,
matrix manipulation (transposing, indexing, slicing),
sampling
and more are comprehensively described here: https://pytorch.org/docs/stable/torch.html

Cheatsheet here: https://pytorch-for-numpy-users.wkentaro.com/

Datasets & DataLoaders

Fashion-MNIST

28 × 28 grayscale images

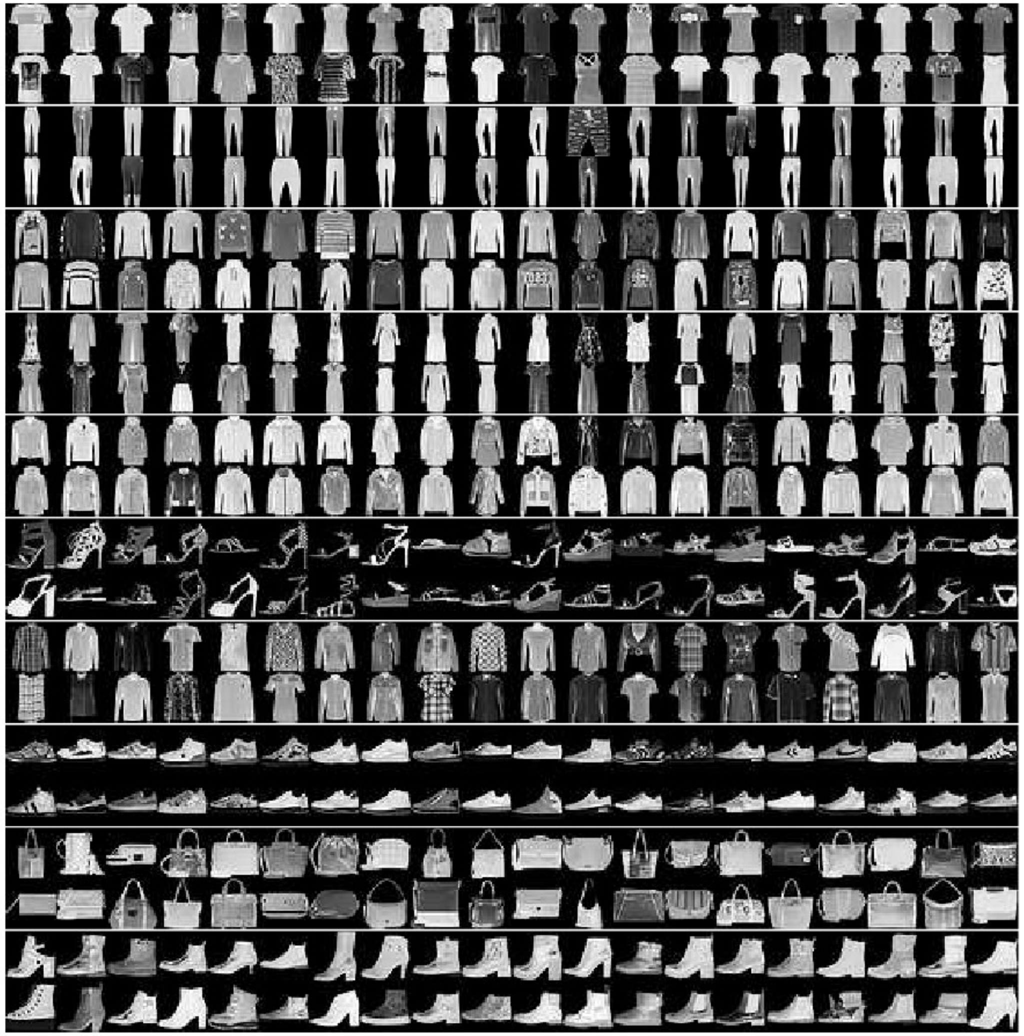
70, 000 fashion products

10 categories

7, 000 images per category

The training set has 60, 000 images and the test set has 10, 000 images.

Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	



Xiao et al. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. arXiv preprint arXiv:1708.07747.

3.1 Loading a Dataset

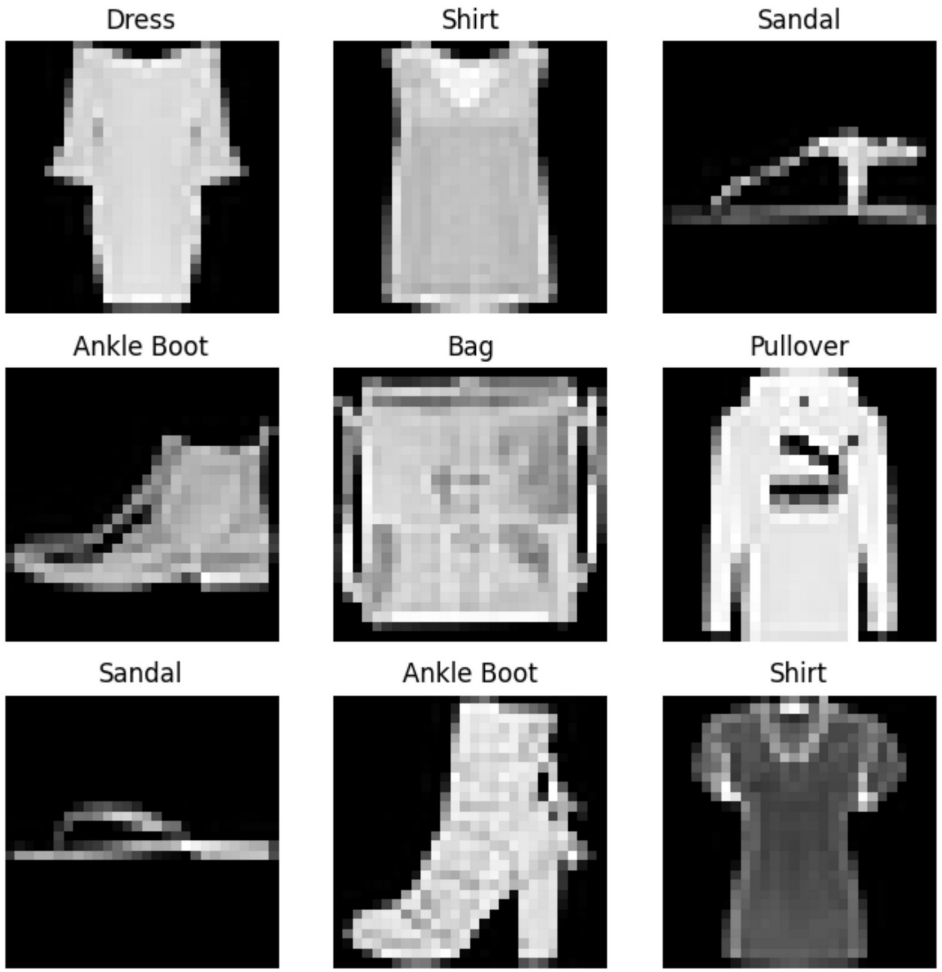
```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
```

3.2 Iterating and Visualizing the Dataset

We can index `Datasets` manually like a list: `training_data[index]`. We use `matplotlib` to visualize some samples in our training data.

```
labels_map = {
    0: "T-Shirt",
    1: "Trouser",
    2: "Pullover",
    3: "Dress",
    4: "Coat",
    5: "Sandal",
    6: "Shirt",
    7: "Sneaker",
    8: "Bag",
    9: "Ankle Boot",
}
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
for i in range(1, cols * rows + 1):
    sample_idx = torch.randint(len(training_data), size=(1,)).item()
    img, label = training_data[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(labels_map[label])
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()
```

3.2 Iterating and Visualizing the Dataset (Cont'd)



3.3 Creating a Custom Dataset for your files

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

3.4 Preparing your data for training with DataLoaders

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

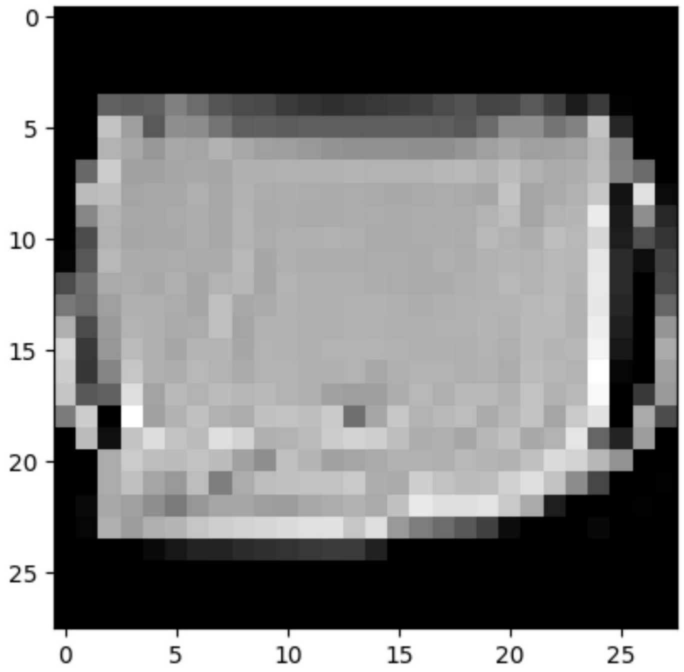
```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

3.5 Iterate through the DataLoader

We have loaded that dataset into the `DataLoader` and can iterate through the dataset as needed. Each iteration below returns a batch of `train_features` and `train_labels` (containing `batch_size=64` features and labels respectively). Because we specified `shuffle=True`, after we iterate over all batches the data is shuffled (for finer-grained control over the data loading order, take a look at Samplers).

```
# Display image and label.
train_features, train_labels = next(iter(train_dataloader))
print(f"Feature batch shape: {train_features.size()}")
print(f"Labels batch shape: {train_labels.size()}")
img = train_features[0].squeeze()
label = train_labels[0]
plt.imshow(img, cmap="gray")
plt.show()
print(f"Label: {label}")
```

3.5 Iterate through the DataLoader (Cont'd)



Out:

```
Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 8
```

Transforms

4 Transforms

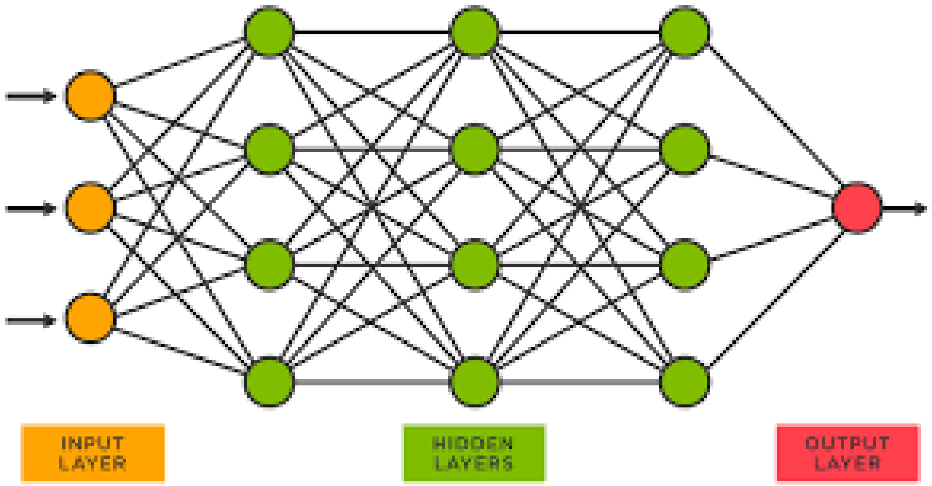
The FashionMNIST features are in PIL Image format, and the labels are integers. For training, we need the features as normalized tensors, and the labels as one-hot encoded tensors. To make these transformations, we use `ToTensor` and `Lambda`.

```
import torch
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda

ds = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
    target_transform=Lambda(lambda y: torch.zeros(10, dtype=torch.float).scatter_(0,
torch.tensor(y), value=1))
)
```

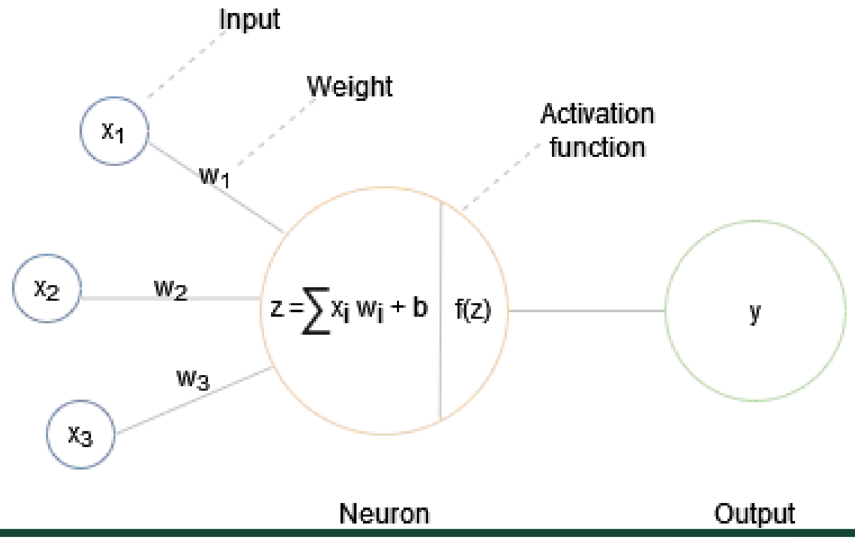
Build Model

5.1 torch.nn.Module
(The base class for all neural network modules)

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```



Source: Microsoft Learn



5.2 Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU, if it is available. Let's check to see if torch.cuda is available, else we continue to use the CPU.

```
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

Out:

```
Using cuda device
```

5.3 Define the Class

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

5.3 Define the Class (Cont'd)

We create an instance of `NeuralNetwork`, and move it to the `device`, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

5.3 Define the Class (Cont'd)

To use the model, we pass it the input data. This executes the model's `forward`, along with some background operations. Do not call `model.forward()` directly!

Calling the model on the input returns a 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([2], device='cuda:0')
```

Autograd

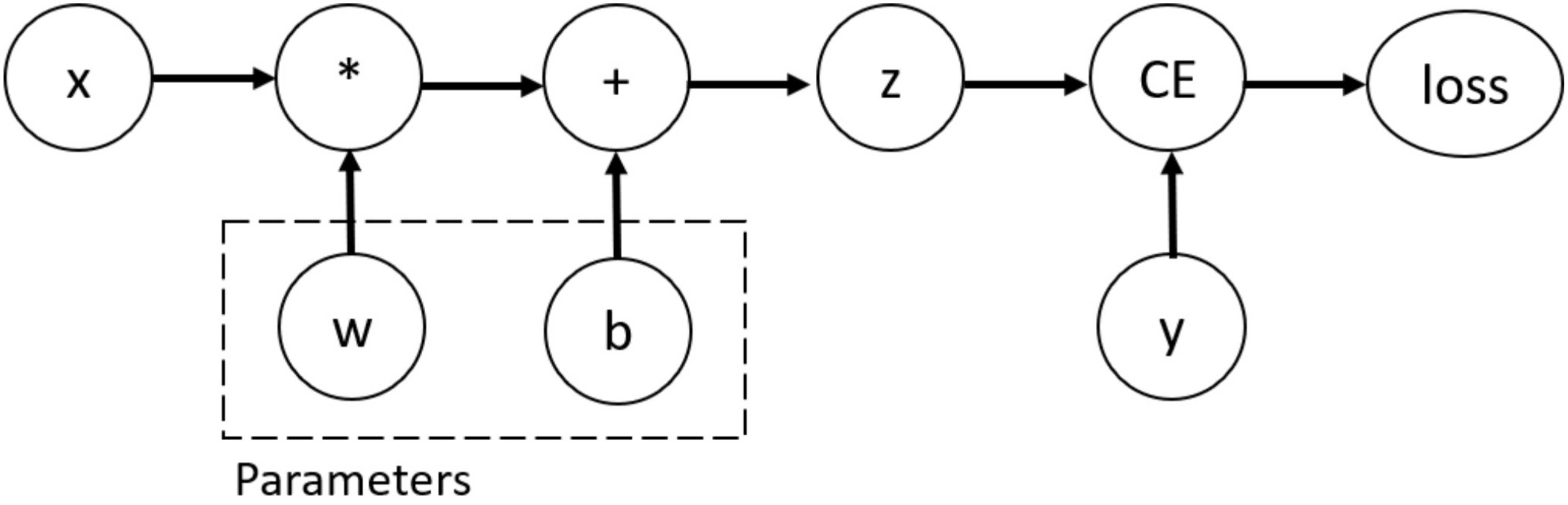
6 Autograd

Consider the simplest one-layer neural network, with input `x`, parameters `w` and `b`, and some loss function. It can be defined in PyTorch in the following manner:

```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

6.1 Tensors, Functions and Computational graph

This code defines the following **computational graph**:



In this network, `w` and `b` are **parameters**, which we need to optimize. Thus, we need to be able to compute the gradients of loss function with respect to those variables. In order to do that, we set the `requires_grad` property of those tensors.

6.2 Computing Gradients

To optimize weights of parameters in the neural network, we need to compute the derivatives of our loss function with respect to parameters, namely, we need $\frac{\partial loss}{\partial w}$ and $\frac{\partial loss}{\partial b}$ under some fixed values of `x` and `y`. To compute those derivatives, we call `loss.backward()`, and then retrieve the values from `w.grad` and `b.grad`:

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

Out:

```
tensor([[0.0286, 0.1466, 0.2703],
        [0.0286, 0.1466, 0.2703],
        [0.0286, 0.1466, 0.2703],
        [0.0286, 0.1466, 0.2703],
        [0.0286, 0.1466, 0.2703]])
tensor([0.0286, 0.1466, 0.2703])
```

6.3 Disabling Gradient Tracking

By default, all tensors with `requires_grad=True` are tracking their computational history and support gradient computation. However, there are some cases when we do not need to do that, for example, when we have trained the model and just want to apply it to some input data, i.e. we only want to do *forward* computations through the network. We can stop tracking computations by surrounding our computation code with `torch.no_grad()` block:

```
z = torch.matmul(x, w)+b
print(z.requires_grad)

with torch.no_grad():
    z = torch.matmul(x, w)+b
print(z.requires_grad)
```

Out:

```
True
False
```

6.3 Disabling Gradient Tracking (Cont'd)

Another way to achieve the same result is to use the `detach()` method on the tensor:

```
z = torch.matmul(x, w)+b
z_det = z.detach()
print(z_det.requires_grad)
```

Out:

```
False
```

Optimization

7.1 Loss Function

Common loss functions include nn.MSELoss (Mean Square Error) for regression tasks, and nn.NLLLoss (Negative Log Likelihood) for classification. nn.CrossEntropyLoss combines `nn.LogSoftmax` and `nn.NLLLoss`.

We pass our model's output logits to `nn.CrossEntropyLoss`, which will normalize the logits and compute the prediction error.

```
# Initialize the loss function
loss_fn = nn.CrossEntropyLoss()
```

7.2 Optimizer

We initialize the optimizer by registering the model's parameters that need to be trained, and passing in the learning rate hyperparameter.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

Inside the training loop, optimization happens in three steps:

- Call `optimizer.zero_grad()` to reset the gradients of model parameters. Gradients by default add up; to prevent double-counting, we explicitly zero them at each iteration.
- Backpropagate the prediction loss with a call to `loss.backward()`. PyTorch deposits the gradients of the loss w.r.t. each parameter.
- Once we have our gradients, we call `optimizer.step()` to adjust the parameters by the gradients collected in the backward pass.

7.3 Full Implementation – Train Loop

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

7.4 Full Implementation – Test Loop

```python
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n")
```

7.5 Full Implementation

```
loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

epochs = 10
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)
print("Done!")
```

7.5 Full Implementation (Cont'd)

Out:

```
Epoch 1
-------------------------------
loss: 2.296602  [    0/60000]
loss: 2.292679  [ 6400/60000]
loss: 2.274081  [12800/60000]
loss: 2.275222  [19200/60000]
loss: 2.255478  [25600/60000]
loss: 2.224747  [32000/60000]
loss: 2.242173  [38400/60000]
loss: 2.200988  [44800/60000]
loss: 2.185223  [51200/60000]
loss: 2.183314  [57600/60000]
Test Error:
 Accuracy: 41.7%, Avg loss: 2.162542

Epoch 2
-------------------------------
loss: 2.160909  [    0/60000]
loss: 2.156497  [ 6400/60000]
```

Save & Load Model

8.1 Saving and Loading Model Weights

PyTorch models store the learned parameters in an internal state dictionary, called `state_dict`. These can be persisted via the `torch.save` method:

```
model = models.vgg16(pretrained=True)
torch.save(model.state_dict(), 'model_weights.pth')
```

To load model weights, you need to create an instance of the same model first, and then load the parameters using `load_state_dict()` method.

```
model = models.vgg16() # we do not specify pretrained=True, i.e. do not load default weights
model.load_state_dict(torch.load('model_weights.pth'))
model.eval()
```

8.2 Saving and Loading Models with Shapes

When loading model weights, we needed to instantiate the model class first, because the class defines the structure of a network. We might want to save the structure of this class together with the model, in which case we can pass `model` (and not `model.state_dict()`) to the saving function:

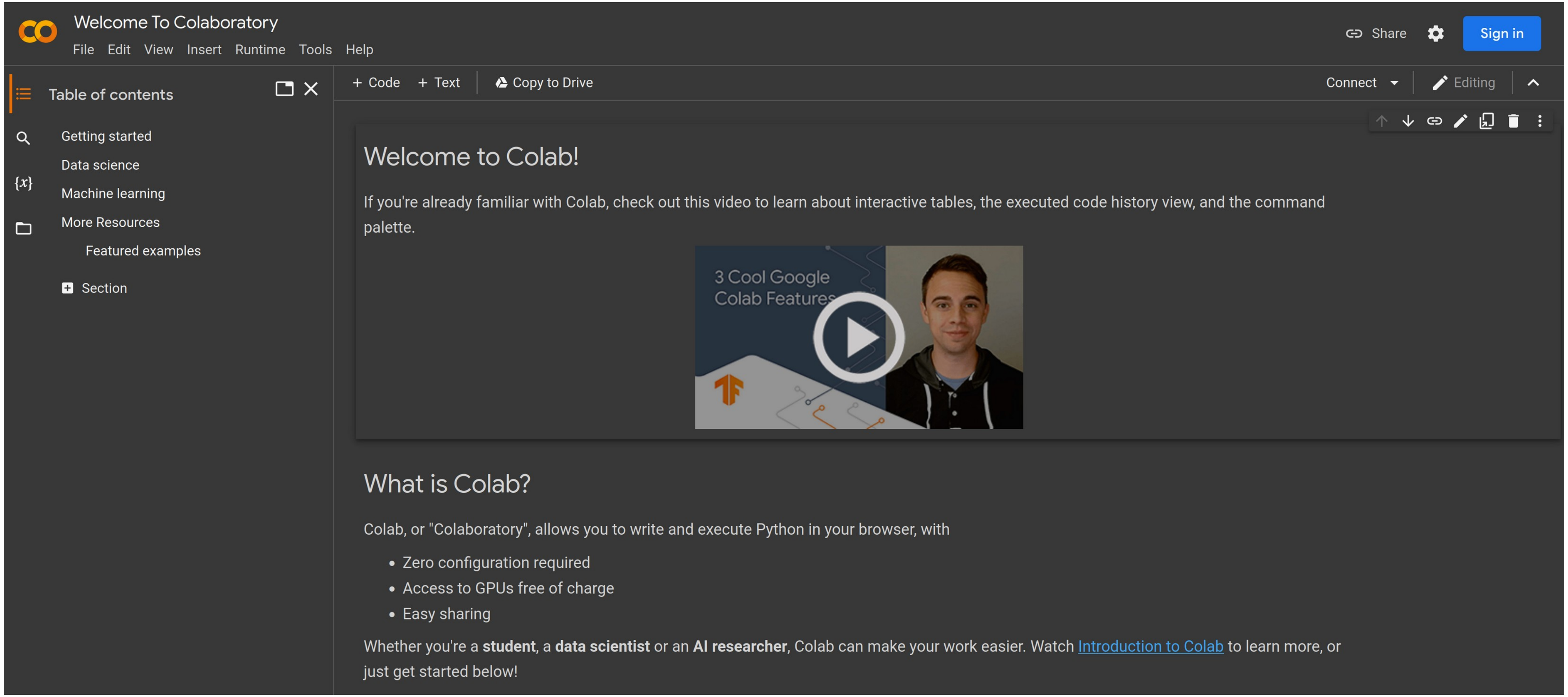
```
torch.save(model, 'model.pth')
```

We can then load the model like this:

```
model = torch.load('model.pth')
```

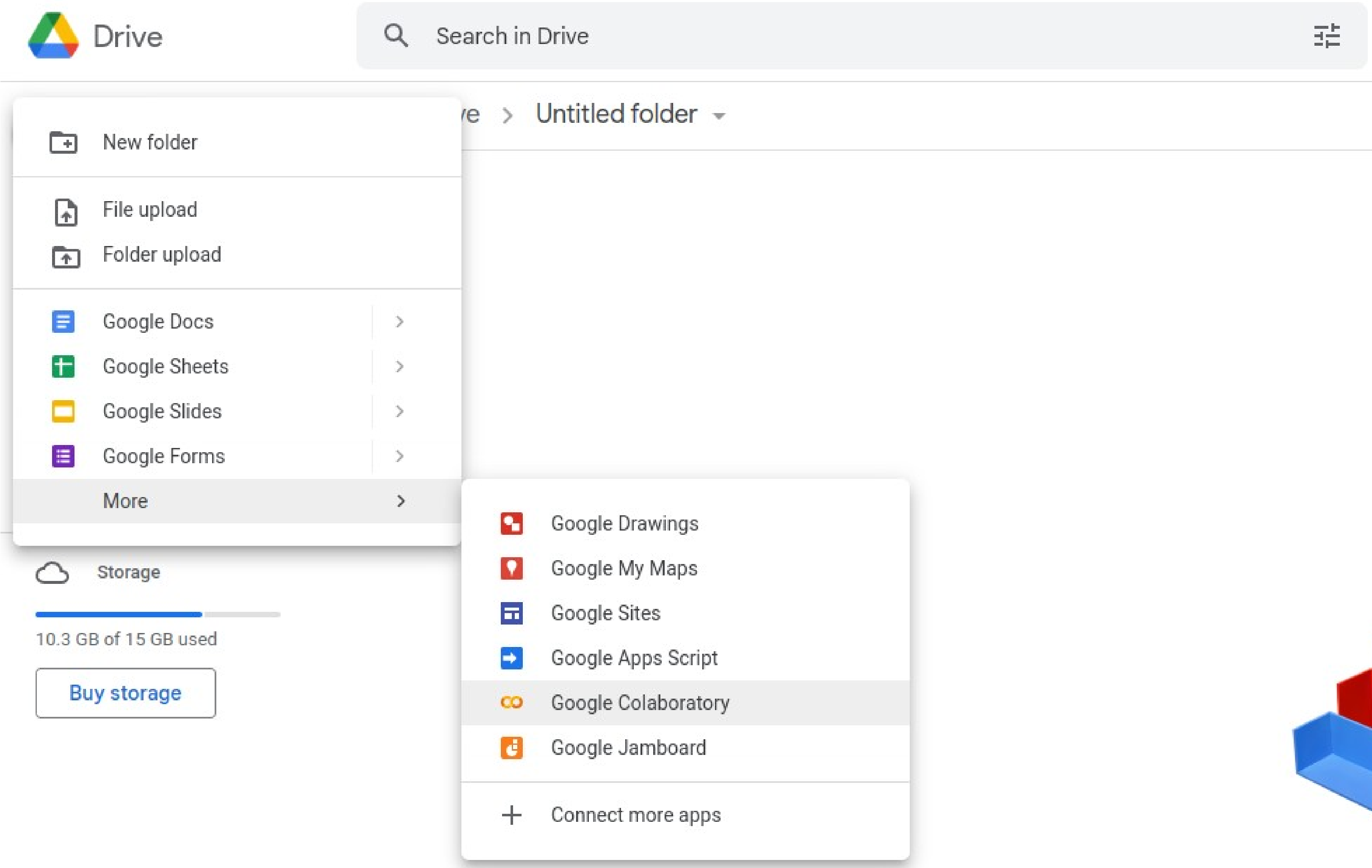
Google Colab

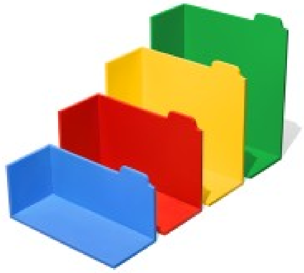
9.1 Free Online GPUs: Google Colab



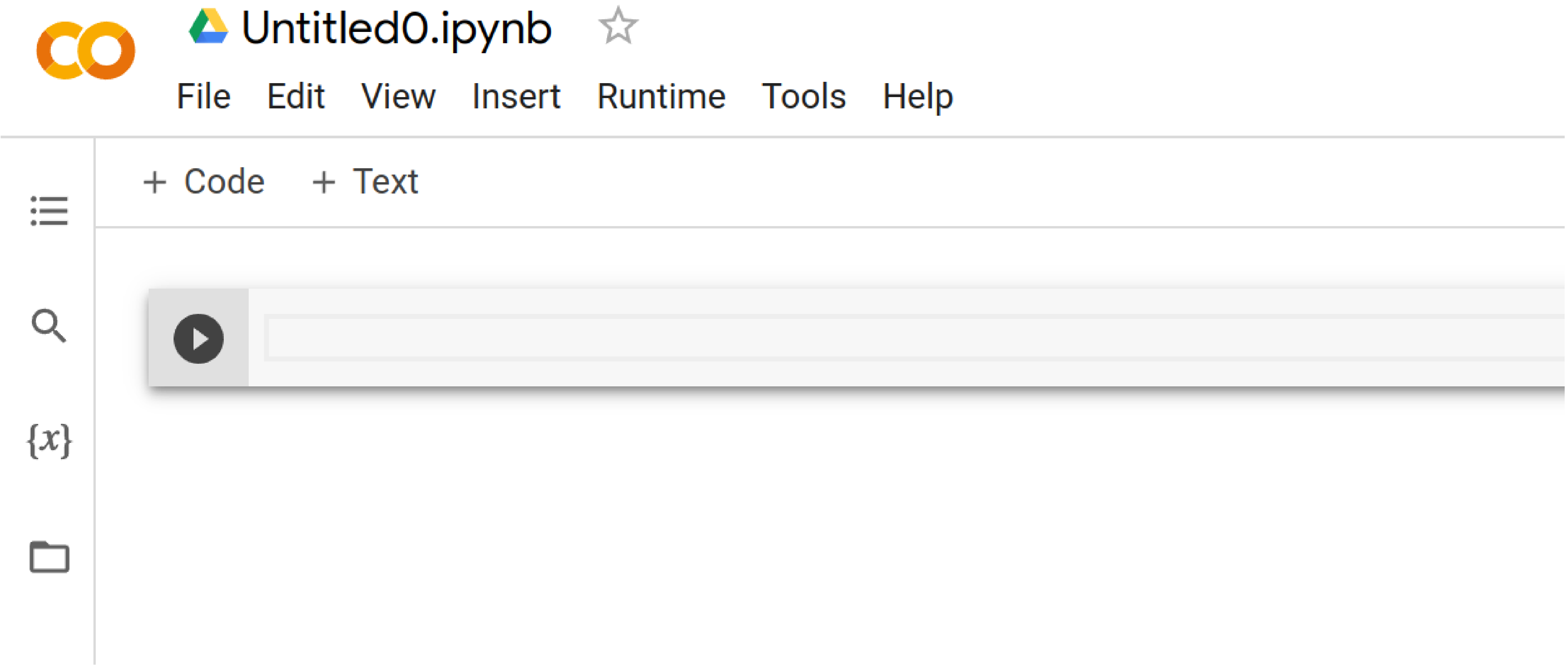
Source: https://colab.research.google.com/

9.2 Create a new Google Colab notebook

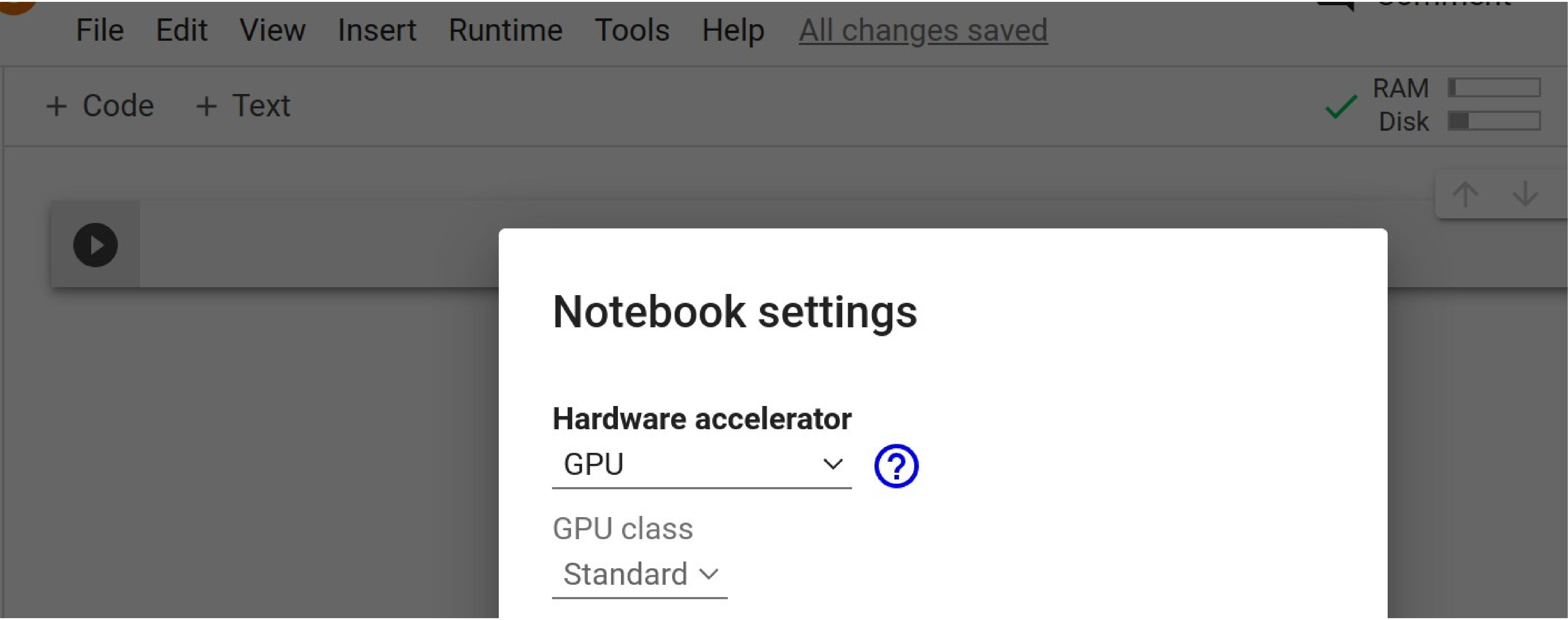




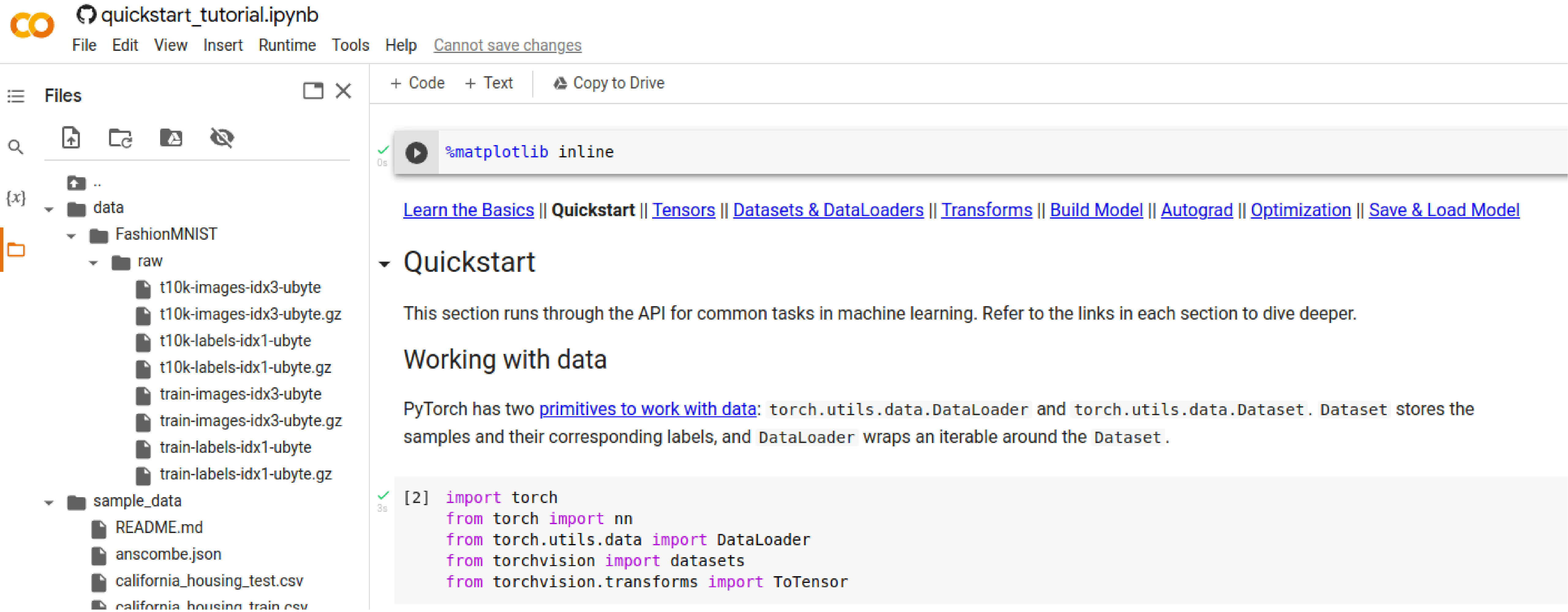
9.2 Create a new Google Colab notebook (Cont'd)



9.3 GPU mode

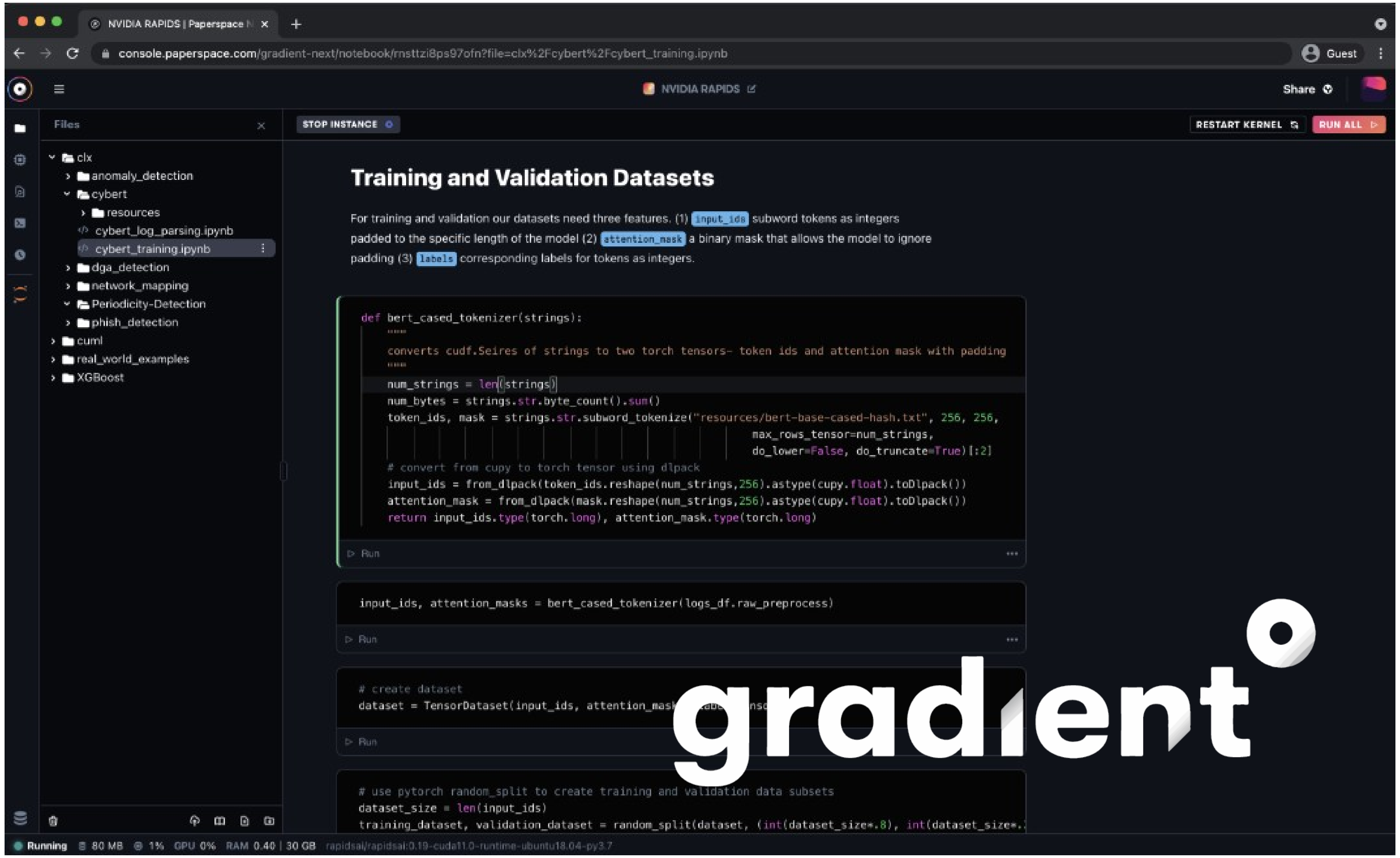

File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
RAM
Disk
Notebook settings
Hardware accelerator
GPU
GPU class
Standard

9.4 Run experiments on Colab



Source: https://colab.research.google.com/github/pytorch/tutorials/blob/gh-pages/_downloads/af0caf6d7af0dda755f4c9d7af9ccc2c/quickstart_tutorial.ipynb

9.5 Other online Free GPUs: Gradient



Source: https://www.paperspace.com/gradient

9.6 Install Conda at local machine

Latest Miniconda Installer Links

Latest - Conda 22.11.1 Python 3.10.8 released December 22, 2022

Platform	Name	SHA256 hash
Windows	Miniconda3 Windows 64-bit	`2e3086630fa3fae7636432a954be530c88d0705fce497120d56e0f5d865b0d51`
	Miniconda3 Windows 32-bit	`4fb64e6c9c28b88beab16994bfba4829110ea3145baa60bda5344174ab65d462`
macOS	Miniconda3 macOS Intel x86 64-bit bash	`7406579393427eaf9bc0e094dcd3c66d1e1b93ee9db4e7686d0a72ea5d7c0ce5`
	Miniconda3 macOS Intel x86 64-bit pkg	`9195ffba1a6984c81c69649ce976a38455ace5b474c24a4363e5ca65fc72e832`
	Miniconda3 macOS Apple M1 64-bit bash	`22eec9b7d3add25ac3f9b60621d8f3d8df3e63d4aa0ae5eb846b558d7ba68333`
	Miniconda3 macOS Apple M1 64-bit pkg	`fbb33c5770b10a0d5a0deef746e7499bfaf8ff840d0d517175036dd8449357f6`
Linux	Miniconda3 Linux 64-bit	`00938c3534750a0e4069499baf8f4e6dc1c2e471c86a59caa0dd03f4a9269db6`
	Miniconda3 Linux-aarch64 64-bit	`48a96df9ff56f7421b6dd7f9f71d548023847ba918c3826059918c08326c2017`
	Miniconda3 Linux-ppc64le 64-bit	`4c86c3383bb27b44f7059336c3a46c34922df42824577b93eadecefbf7423836`
	Miniconda3 Linux-s390x 64-bit	`a150511e7fd19d07b770f278fb5dd2df4bc24a8f55f06d6274774f209a36c766`

Source: https://docs.conda.io/en/latest/miniconda.html

9.7 Install PyTorch at local machine

PyTorch Build	Stable (1.13.1)		Preview (Nightly)	
Your OS	Linux	Mac	Windows	
Package	Conda	Pip	LibTorch	Source
Language	Python		C++ / Java	
Compute Platform	CUDA 11.6	CUDA 11.7	ROCm 5.2	CPU

Run this Command:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.6 -c pytorch
-c nvidia
```

Source: https://pytorch.org/get-started/locally/

9.8 Install previous versions of PyTorch

COMMANDS FOR VERSIONS < 1.0.0

Via conda

> *This should be used for most previous macOS version installs.*

To install a previous version of PyTorch via Anaconda or Miniconda, replace “0.4.1” in the following commands with the desired version (i.e., “0.2.0”).

Installing with CUDA 9

```
conda install pytorch=0.4.1 cuda90 -c pytorch
```

or

```
conda install pytorch=0.4.1 cuda92 -c pytorch
```

Source: https://pytorch.org/get-started/previous-versions/

9.9 New GPUs may not support old CUDA :(

Hardware Generation	Compute Capability	CTK Support	Latest Forward Comaptibility Package Support	Driver	
				Current Minimum Driver in Support	Maximum Driver Supported*
Hopper	9.x	11.8 - current	current	450.36.06+	latest
NVIDIA Ampere GPU Arch.	8.x	11.0 - current		450.36.06+	latest
Turing	7.5	10.0 - current		450.36.06+	latest
Volta	7.x	9.0 - current		450.36.06+	latest
Pascal	6.x	8.0 - current		450.36.06+	latest
Maxwell	5.x	6.5 - current		450.36.06+	latest

Source: https://docs.nvidia.com/deploy/cuda-compatibility/index.html

More Resources

Tutorial Code:
https://github.com/pytorch/tutorials/tree/master/beginner_source/basics

PyTorch Docs:

https://pytorch.org/docs/stable/index.html

Conda Docs:

https://docs.conda.io/projects/conda/en/stable/

Jupyter Docs:

https://jupyter-notebook.readthedocs.io/en/stable/

Acknowledgments

PyTorch Official Tutorial:

https://pytorch.org/tutorials/beginner/basics/intro.html

CS 231N PyTorch Tutorial
Drew Kaul - Stanford University