Instructions:

1. Libraries allowed: **Python basic libraries, numpy, pandas, scikit-learn (only for data processing), and pytorch.**
2. Show all outputs.
3. Submit jupyter notebook and a pdf export of the notebook.
4. For practice examples, change variable name, shape, mathematical operations, nummerical values, tensor sizes etc. Be creative!

Data Collection and Pre-processing

Q1

a) Tensors

Complete the tutorial at https://docs.pytorch.org/tutorials/beginner/basics/tensorqs_tutorial.html . make changes to the tensor, variable name etc. to make your example completely different from tutorial. Failing to do so will earn you zero credit.

Add as many cells below as needed.

```
import torch
import numpy as np
```

1.1 Directly from data

```
data = [[1,3.14],
        [4,5],
        [6,7]]
```

```
data
```

```
[[1, 3.14], [4, 5], [6, 7]]
```

```
tensor_data = torch.tensor(data)
```

```
tensor_data
```

```
tensor([[1.0000, 3.1400],
        [4.0000, 5.0000],
        [6.0000, 7.0000]])
```

```
tensor_data.shape
```

```
torch.Size([3, 2])
```

1.2 From a NumPy array

```
np_data = np.array(data)
```

```
np_data
```

```
array([[1.  , 3.14],
       [4.  , 5.  ],
       [6.  , 7.  ]])
```

```
tensor_np_data=torch.from_numpy(np_data)
```

```
tensor_np_data
```

```
tensor([[1.0000, 3.1400],
        [4.0000, 5.0000],
        [6.0000, 7.0000]], dtype=torch.float64)
```

```
tensor_np_data.shape
```

```
torch.Size([3, 2])
```

1.3 From another tensor

```
x_ones = torch.ones_like(tensor_data)
```

```
print(f"Ones Tensor: \n {x_ones} \n")
```

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

```
x_rand = torch.rand_like(tensor_data, dtype=torch.float)
```

```
print(f"Random Tensor: \n {x_rand} \n")
```

```
Random Tensor:
 tensor([[0.7702, 0.7655],
        [0.1124, 0.8028],
        [0.1585, 0.9980]])
```

1.4 With random or constant values

```
shape=(4,2) # this is a tuple
```

```
rand = torch.rand(shape)
ones = torch.ones(shape)
zeros = torch.zeros(shape)
```

```
print(f"Random Tensor: \n {rand} \n")
print(f"Ones Tensor: \n {ones} \n")
print(f"Zeros Tensor: \n {zeros}")
```

```
Random Tensor:
 tensor([[0.2552, 0.1375],
        [0.3947, 0.0059],
        [0.7907, 0.5160],
        [0.4814, 0.0043]])

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

Zeros Tensor:
 tensor([[0., 0.],
        [0., 0.],
        [0., 0.],
        [0., 0.]])
```

1.5 Attributes of a Tensor

```
tensor = torch.rand(2,5)

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

```
Shape of tensor: torch.Size([2, 5])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

2 Operations on Tensors

```
# We move our tensor to the current accelerator if available
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())
```

2.1 Standard numpy-like indexing and slicing:

```
tensor = torch.ones(3, 3)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,2] = 0
print(tensor)
```

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

Joining tensors

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

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

```
t2 = torch.cat([tensor, tensor, tensor], dim=0)
print(t2)
```

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

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)
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

```
y1
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

```
y2
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

```
y3
```

```
tensor([[2., 2., 2.],
        [2., 2., 2.],
        [2., 2., 2.]])
```

```
# 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)
```

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

Single-element tensors

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

```
6.0 <class 'float'>
```

In-place operations

```
tensor = torch.ones(3,3)
print(f"{tensor} \n")
tensor.add_(3)
print(tensor)
```

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

tensor([[4., 4., 4.],
        [4., 4., 4.],
        [4., 4., 4.]])
```

3 Bridge with NumPy

3.1 Tensor to NumPy array

```
t = torch.ones(3)
print(f"t: {t}")
n = t.numpy()
print(f"n: {n}")
```

```
t: tensor([1., 1., 1.])
n: [1. 1. 1.]
```

```
t.add_(1)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2.])
n: [2. 2. 2.]
```

```
n = np.ones(5)
t = torch.from_numpy(n)
```

```
np.add(n, 1, out=n)
print(f"t: {t}")
print(f"n: {n}")
```

```
t: tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
n: [2. 2. 2. 2. 2.]
```

b) Autormatic differentiation

Complete the tutorial at https://pytorch.org/tutorials/beginner/basics/autogradqs_tutorial.html . Again, make changes to the tensor, variable name etc. to make your example completely different from tutorial. Failing to do so will earn you zero credit.

Add as many cells below as needed.

```
import torch

x = torch.ones(6)  # input tensor
y = torch.zeros(2)  # expected output
w = torch.randn(6, 2, requires_grad=True)
b = torch.randn(2, requires_grad=True)
z = x @ w +b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

```
print(f"Gradient function for z = {z.grad_fn}")
print(f"Gradient function for loss = {loss.grad_fn}")
```

```
Gradient function for z = <AddBackward0 object at 0x78eea2820910>
Gradient function for loss = <BinaryCrossEntropyWithLogitsBackward0 object at 0x78eea324b490>
```

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

```
tensor([[0.4262, 0.1864],
        [0.4262, 0.1864],
        [0.4262, 0.1864],
        [0.4262, 0.1864],
        [0.4262, 0.1864],
        [0.4262, 0.1864]])
tensor([0.4262, 0.1864])
```

Disabling Gradient Tracking

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

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

```
True
False
```

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

```
False
```

Jacobian Product

```
inp = torch.eye(4, 5, requires_grad=True)
```

```
inp
```

```
tensor([[1., 0., 0., 0., 0.],
        [0., 1., 0., 0., 0.],
        [0., 0., 1., 0., 0.],
        [0., 0., 0., 1., 0.]], requires_grad=True)
```

```
out = (inp+1).pow(2).t()
```

```
out
```

```
tensor([[4., 1., 1., 1.],
        [1., 4., 1., 1.],
        [1., 1., 4., 1.],
        [1., 1., 1., 4.],
        [1., 1., 1., 1.]], grad_fn=<TBackward0>)
```

```
out.backward(torch.ones_like(out), retain_graph=True)
print(f"First call\n{inp.grad}")
```

```
First call
tensor([[4., 2., 2., 2., 2.],
        [2., 4., 2., 2., 2.],
```

```
        [2., 2., 4., 2., 2.],
        [2., 2., 2., 4., 2.]])
```

```
out.backward(torch.ones_like(out), retain_graph=True)
print(f"\nSecond call\n{inp.grad}")
```

```
Second call
tensor([[8., 4., 4., 4., 4.],
        [4., 8., 4., 4., 4.],
        [4., 4., 8., 4., 4.],
        [4., 4., 4., 8., 4.]])
```

```
inp.grad.zero_()
out.backward(torch.ones_like(out), retain_graph=True)
print(f"\nCall after zeroing gradients\n{inp.grad}")
```

```
Call after zeroing gradients
tensor([[4., 2., 2., 2., 2.],
        [2., 4., 2., 2., 2.],
        [2., 2., 4., 2., 2.],
        [2., 2., 2., 4., 2.]])
```

c) Datasets and Dataloader

Download the "Hand Keypoint detection" dataset from https://www.kaggle.com/datasets/pablocumbrera/hand-keypoint-detection .

1. Define a pytorch Dataset class, HandKpDetection.
    1. Implement `__init__` method, in which gather the filepaths to all images in the dataset and store them in a list. Don't load the entire dataset or images in this method. Both image and label file share the same prefix, so just storing filepath for images will work.
    2. implement `__len__` method.
    3. Implement `__getitem__(self, index)` where you will load just one sample corresponding to the `index` in the method argument. Use the filepath information in the init methods to load both image and keypoints. Reshape the image to 32×32 and keypoints to 20×2

2. Implement Dataloaders with a minibatch size of 2. Convert both image and keypoints to `torch.tensor`.
3. Visualize the images and print the keypoints in the first minibatch

Add as many cells below as needed.

Download the dataset

```
import kagglehub

# Download latest version
path = kagglehub.dataset_download("pablocumbrera/hand-keypoint-detection")

print("Path to dataset files:", path)
```

```
Downloading from https://www.kaggle.com/api/v1/datasets/download/pablocumbrera/hand-keypoint-detection?dataset_version_nur
100%|██████████| 632M/632M [00:04<00:00, 154MB/s]
Extracting files...
Path to dataset files: /root/.cache/kagglehub/datasets/pablocumbrera/hand-keypoint-detection/versions/1
```

Define a pytorch Dataset class, HandKpDetection

```
import os, glob, json
import torch
from torch.utils.data import Dataset
from PIL import Image
import numpy as np
```

```
class HandKpDetection(Dataset):
    def __init__(self, root_dir, transform=None):
        """
        Args:
            root_dir (str): Dataset root, e.g.
                '/root/.cache/kagglehub/datasets/pablocumbrera/hand-keypoint-detection/versions/1/hand_lab
            transform: optional torchvision transform for images
```

```
        """
        self.root_dir = root_dir
        self.transform = transform

        # collect all .jpg files under synth1..synth4
        self.image_paths = []
        for folder in ["synth1", "synth2", "synth3", "synth4"]:
            self.image_paths.extend(sorted(glob.glob(os.path.join(root_dir, folder, "*.jpg"))))

        print("Found", len(self.image_paths), "images")

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

    def __getitem__(self, index):
      # --- load image ---
      img_path = self.image_paths[index]
      image = Image.open(img_path).convert("RGB")
      orig_w, orig_h = image.size   # original size

      # resize
      image = image.resize((32, 32))
      new_w, new_h = 32, 32

      # convert to tensor
      image = np.array(image, dtype=np.float32) / 255.0
      image = np.transpose(image, (2, 0, 1))
      image = torch.tensor(image)

      # --- load keypoints ---
      json_path = img_path.replace(".jpg", ".json")
      with open(json_path, "r") as f:
          ann = json.load(f)
      keypoints = np.array(ann["hand_pts"], dtype=np.float32)[:, :2]
      keypoints = keypoints.reshape(-1, 2)[:20]
```

```
        # scale keypoints to new size
        scale_x = new_w / orig_w
        scale_y = new_h / orig_h
        keypoints[:, 0] *= scale_x
        keypoints[:, 1] *= scale_y

        keypoints = torch.tensor(keypoints)

        return image, keypoints
```

Implement Dataloaders with a minibatch size of 2. Convert both image and keypoints to torch.tensor.

```
from torch.utils.data import DataLoader

root = "/root/.cache/kagglehub/datasets/pablocumbrera/hand-keypoint-detection/versions/1/hand_labels_synth
dataset = HandKpDetection(root)
dataloader = DataLoader(dataset, batch_size=2, shuffle=True)

images, keypoints = next(iter(dataloader))
print("Batch images:", images.shape)      # [2, 3, 32, 32]  [batch_size, channels, shape_x, shape_y]
print("Batch keypoints:", keypoints.shape) # [2, 20, 2]     [batch_size, shape_x, shape_y]
```

```
Found 14261 images
Batch images: torch.Size([2, 3, 32, 32])
Batch keypoints: torch.Size([2, 20, 2])
```

Visualize the images and print the keypoints in the first minibatch

```
import matplotlib.pyplot as plt

# get the first minibatch
images, keypoints = next(iter(dataloader))

print("First minibatch keypoints (tensor):")
```

```
print(keypoints)

# visualize both images in the minibatch
fig, axes = plt.subplots(1, 2, figsize=(6, 3))

for i in range(2):  # because batch_size = 2
    img = images[i].permute(1, 2, 0).numpy()   # CHW -> HWC
    kp = keypoints[i].numpy()

    axes[i].imshow(img)
    axes[i].scatter(kp[:, 0], kp[:, 1], c='red', s=10)  # plot keypoints
    axes[i].set_title(f"Sample {i}")
    axes[i].axis("off")

plt.show()
```

```
First minibatch keypoints (tensor):
tensor([[[15.2862, 13.7748],
         [16.5025, 14.4271],
         [17.8023, 15.0550],
         [18.5364, 15.7991],
         [ 0.0000,  0.0000],
         [16.7636, 15.2475],
         [16.8107, 16.3507],
         [16.8385, 17.3640],
         [ 0.0000,  0.0000],
         [15.8861, 15.3574],
         [16.0766, 16.5415],
         [16.2135, 17.6143],
         [ 0.0000,  0.0000],
         [15.1119, 15.4600],
         [15.4075, 16.5340],
         [15.6191, 17.5492],
         [ 0.0000,  0.0000],
         [14.2789, 15.5900],
         [14.7824, 16.7960],
         [14.9611, 17.3822]],
```

d) Build model

Complete the tutorial at https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html.

Add as many cells below as needed.

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

```
         [16.4958, 15.4813],
         [17.1234, 16.1749],
         [19.8924, 12.9190],
         [13.4585, 13.9200],
```

1 Get Device for Training

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

```
Using cpu device
```

```
         [11.6058, 13.7552],
         [ 9.2715, 15.6757],
         [10.5937, 18.0321],
         [17.9553, 17.8488],
         [13.1157, 19.2567],
         [11.4207, 21.1962]]])
```

1.1 Define the Class

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__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
```

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

```
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)
  )
)
```

```
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}")
```

```
Predicted class: tensor([5])
```

2 Model Layers

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

```
torch.Size([3, 28, 28])
```

2.1 nn.Flatten

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

```
torch.Size([3, 784])
```

2.2 nn.Linear

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

```
torch.Size([3, 20])
```

2.3 nn.ReLU

```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

```
Before ReLU: tensor([[ 0.4120,  0.1090,  0.3217,  0.6498,  0.0514, -0.1481, -0.2099, -0.1682,
          0.1685,  0.5080, -0.3803, -0.2483,  0.1397, -0.2026,  0.5672, -0.0646,
          0.5508,  0.4268,  0.2526,  0.0836],
```

```
          [ 0.4570,  0.0408,  0.3579,  0.3994,  0.2799,  0.1412, -0.2049, -0.3208,
            0.2147,  0.2715, -0.3543,  0.1022,  0.2976, -0.0227,  0.3972, -0.2048,
            0.5589,  0.7151,  0.3847, -0.3243],
          [ 0.1550,  0.0369,  0.1152,  0.2209,  0.0533,  0.2396,  0.2607, -0.1804,
            0.0997,  0.1888, -0.3111, -0.2393,  0.0185, -0.4501,  0.3856, -0.1492,
            0.9775,  0.7084,  0.2112,  0.0883]], grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.4120, 0.1090, 0.3217, 0.6498, 0.0514, 0.0000, 0.0000, 0.0000, 0.1685,
         0.5080, 0.0000, 0.0000, 0.1397, 0.0000, 0.5672, 0.0000, 0.5508, 0.4268,
         0.2526, 0.0836],
        [0.4570, 0.0408, 0.3579, 0.3994, 0.2799, 0.1412, 0.0000, 0.0000, 0.2147,
         0.2715, 0.0000, 0.1022, 0.2976, 0.0000, 0.3972, 0.0000, 0.5589, 0.7151,
         0.3847, 0.0000],
        [0.1550, 0.0369, 0.1152, 0.2209, 0.0533, 0.2396, 0.2607, 0.0000, 0.0997,
         0.1888, 0.0000, 0.0000, 0.0185, 0.0000, 0.3856, 0.0000, 0.9775, 0.7084,
         0.2112, 0.0883]], grad_fn=<ReluBackward0>)
```

2.4 nn.Sequential

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

2.5 nn.Softmax

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

3 Model Parameters

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values : {param[:2]} \n")
```

```
Model structure: 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)
  )
)


Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) | Values : tensor([[-0.0056, -0.0205, -0.0161,  ...,  0.0
        [-0.0218,  0.0271, -0.0139,  ...,  0.0201,  0.0002, -0.0295]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values : tensor([ 0.0233, -0.0174], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) | Values : tensor([[-0.0417,  0.0148,  0.0376,  ..., -0.0
        [ 0.0228,  0.0169, -0.0341,  ...,  0.0015,  0.0166,  0.0191]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values : tensor([ 0.0288, -0.0011], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) | Values : tensor([[-0.0196,  0.0141, -0.0236,  ...,  0.00
        [ 0.0201,  0.0274, -0.0346,  ...,  0.0156, -0.0084, -0.0153]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values : tensor([-0.0318,  0.0280], grad_fn=<SliceBackward0>)
```

e) Optimization loop

Complete the tutorial at https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html.

Add as many cells below as needed.

Prerequisite Code

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

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

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

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__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

    model = NeuralNetwork()
```

```
100%|██████████| 26.4M/26.4M [00:03<00:00, 7.19MB/s]
100%|██████████| 29.5k/29.5k [00:00<00:00, 233kB/s]
100%|██████████| 4.42M/4.42M [00:01<00:00, 4.27MB/s]
100%|██████████| 5.15k/5.15k [00:00<00:00, 10.4MB/s]
```

Hyperparameters

We define the following hyperparameters for training:

Number of Epochs - the number of times to iterate over the dataset

Batch Size- the number of data samples propagated through the network before the parameters are updated

Learning Rate - how much to update models parameters at each batch/epoch. Smaller values yield slow learning speed, while large values may result in unpredictable behavior during training.

```
learning_rate = 1e-3
batch_size = 64
epochs = 5
```

Optimization Loop

Each epoch consists of two main parts: The Train Loop - iterate over the training dataset and try to converge to optimal parameters.

The Validation/Test Loop - iterate over the test dataset to check if model performance is improving.

Loss Function

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

Optimizer

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

Full Implementation

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

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

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


def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
```

```
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for tensors with requires_g
    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")
```

```
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!")
```

```
Epoch 1
-------------------------------
loss: 2.308622  [   64/60000]
loss: 2.290988  [ 6464/60000]
loss: 2.271333  [12864/60000]
loss: 2.265017  [19264/60000]
loss: 2.245721  [25664/60000]
loss: 2.229995  [32064/60000]
loss: 2.230161  [38464/60000]
loss: 2.207123  [44864/60000]
loss: 2.197040  [51264/60000]
loss: 2.169623  [57664/60000]
Test Error:
```

```
 Accuracy: 47.9%, Avg loss: 2.162924

Epoch 2
-------------------------------
loss: 2.177693  [   64/60000]
loss: 2.169887  [ 6464/60000]
loss: 2.109456  [12864/60000]
loss: 2.123097  [19264/60000]
loss: 2.086841  [25664/60000]
loss: 2.026262  [32064/60000]
loss: 2.048617  [38464/60000]
loss: 1.981367  [44864/60000]
loss: 1.981413  [51264/60000]
loss: 1.908355  [57664/60000]
Test Error:
 Accuracy: 56.0%, Avg loss: 1.908405

Epoch 3
-------------------------------
loss: 1.944233  [   64/60000]
loss: 1.920965  [ 6464/60000]
loss: 1.795443  [12864/60000]
loss: 1.833873  [19264/60000]
loss: 1.739199  [25664/60000]
loss: 1.675555  [32064/60000]
loss: 1.691882  [38464/60000]
loss: 1.593845  [44864/60000]
loss: 1.614733  [51264/60000]
loss: 1.504018  [57664/60000]
Test Error:
 Accuracy: 58.9%, Avg loss: 1.527136

Epoch 4
-------------------------------
loss: 1.595046  [   64/60000]
loss: 1.565867  [ 6464/60000]
loss: 1.403391  [12864/60000]
loss: 1.479101  [19264/60000]
loss: 1.366183  [25664/60000]
loss: 1.351097  [32064/60000]
loss: 1.359906  [38464/60000]
loss: 1.281658  [44864/60000]
loss: 1.315342  [51264/60000]
loss: 1.214487  [57664/60000]
```

f) Save, load, and use model

Complete the tutorial at https://pytorch.org/tutorials/beginner/basics/saveloadrun_tutorial.html.

Add as many cells below as needed.

```
import torch
import torchvision.models as models
```

Saving and Loading Model Weights

```
model = models.vgg16(weights='IMAGENET1K_V1')
torch.save(model.state_dict(), 'model_weights.pth')
```

```
vnloading: "https://download.pytorch.org/models/vgg16-397923af.pth" to /root/.cache/torch/hub/checkpoints/vgg16-397923af.pt
0%|          | 528M/528M [00:07<00:00, 71.2MB/s]
```

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

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
```

```
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
      (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (25): ReLU(inplace=True)
      (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (27): ReLU(inplace=True)
      (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (29): ReLU(inplace=True)
      (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
    (classifier): Sequential(
      (0): Linear(in_features=25088, out_features=4096, bias=True)
      (1): ReLU(inplace=True)
      (2): Dropout(p=0.5, inplace=False)
      (3): Linear(in_features=4096, out_features=4096, bias=True)
      (4): ReLU(inplace=True)
      (5): Dropout(p=0.5, inplace=False)
      (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
  )
```

Saving and Loading Models with Shape

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

```
model = torch.load('model.pth', weights_only=False)
```

Exercise

Q2

a) Create a pytorch class for a neural network with 3 inputs, one hidden layer with 5 neurons and ReLU activatioin function, one output layer with 2 outputs. Use $torch.nn.init$ to initialize the biases to zeros and initialize the weights from a random distribution with vairiance 0.1 and mean 0.0.

```
# code
import torch
import torch.nn as nn
import math

class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(3, 5),
            nn.ReLU(),
            nn.Linear(5, 2),
        )

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

torch.nn.init.zeros_(model.linear_relu_stack[0].bias)
torch.nn.init.zeros_(model.linear_relu_stack[2].bias)
torch.nn.init.normal_(model.linear_relu_stack[0].weight, mean=0.0, std=math.sqrt(0.1))
torch.nn.init.normal_(model.linear_relu_stack[2].weight, mean=0.0, std=math.sqrt(0.1))
```

```
Parameter containing:
tensor([[-0.3566, -0.1552,  0.0519, -0.3366,  0.0977],
        [-0.1617,  0.2685, -0.3114,  0.5047,  0.1659]], requires_grad=True)
```

```
model
```

```
NeuralNetwork(
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=3, out_features=5, bias=True)
    (1): ReLU()
    (2): Linear(in_features=5, out_features=2, bias=True)
  )
)
```

b) Instantiate the class to create a model. Access the weights and biases and print them.

```
# code
model = NeuralNetwork()

for name, param in model.state_dict().items():
    print(f"{name}: {param}")
```

```
linear_relu_stack.0.weight: tensor([[ 0.3774,  0.2328,  0.3459],
        [ 0.0646, -0.2178,  0.0162],
        [-0.5591,  0.0329, -0.4415],
        [ 0.4316, -0.1069,  0.1645],
        [ 0.2732,  0.2034, -0.4304]])
linear_relu_stack.0.bias: tensor([ 0.0467,  0.5147,  0.0162, -0.4779,  0.4254])
linear_relu_stack.2.weight: tensor([[ 0.3728, -0.0880,  0.3002,  0.1626,  0.2000],
        [ 0.4351,  0.3026,  0.0636,  0.1249, -0.3836]])
linear_relu_stack.2.bias: tensor([0.0372, 0.1845])
```

c) Create a tensor X with 2 samples and 3 features randomly from a uniform distribution. Suppose the labels are 0 and 1, respectively for these two samples.

```
# code
X = torch.rand(2, 3)
y = torch.tensor([0, 1])
```

d) Instantiate crossentropy loss.

```
# code
CELoss = nn.CrossEntropyLoss()
```

e) Define an optimizer with stochastic gradient descent. Set the learning rate to 0.1.

```
# code
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

f) Set the gradients of the model to zero. Do backpropagation and find the jacobians of loss with respect to weights and biases in the first and second layers.

```
# code
optimizer.zero_grad()
output = model(X)
loss = CELoss(output, y)
loss.backward()

print("Gradients of first layer weights:")
print(model.linear_relu_stack[0].weight.grad)

print("Gradients of first layer bias:")
print(model.linear_relu_stack[0].bias.grad,"\n\n")

print("Gradients of second layer weights:")
print(model.linear_relu_stack[2].weight.grad)

print("Gradients of second layer bias:")
print(model.linear_relu_stack[2].bias.grad)
```

```
Gradients of first layer weights:
tensor([[ 0.0073,  0.0053,  0.0047],
        [ 0.0455,  0.0335,  0.0293],
        [ 0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000],
        [-0.0680, -0.0501, -0.0438]])
```

```
Gradients of first layer bias:
tensor([ 0.0023,  0.0147,  0.0000,  0.0000, -0.0219])


Gradients of second layer weights:
tensor([[-0.0917, -0.0094,  0.0000,  0.0000, -0.0329],
        [ 0.0917,  0.0094,  0.0000,  0.0000,  0.0329]])
Gradients of second layer bias:
tensor([-0.0376,  0.0376])
```

g) Update the parameters.

```
# code
optimizer.step()
```

h) Print the weight and biases in the first layer after update.

```
# code
for name, param in model.state_dict().items():
    print(f"{name}: {param}")
```

```
linear_relu_stack.0.weight: tensor([[ 0.3766,  0.2322,  0.3454],
        [ 0.0601, -0.2212,  0.0133],
        [-0.5591,  0.0329, -0.4415],
        [ 0.4316, -0.1069,  0.1645],
        [ 0.2800,  0.2084, -0.4260]])
linear_relu_stack.0.bias: tensor([ 0.0465,  0.5132,  0.0162, -0.4779,  0.4276])
linear_relu_stack.2.weight: tensor([[ 0.3819, -0.0871,  0.3002,  0.1626,  0.2033],
        [ 0.4259,  0.3017,  0.0636,  0.1249, -0.3869]])
linear_relu_stack.2.bias: tensor([0.0409, 0.1808])
```

i) save the model

```
# code
torch.save(model.state_dict(), 'model.pth')
```

j) instantiate the network in Q2(a) agian to create `model2`

```
model2 = NeuralNetwork()
```

k) Print the weight and biases in the first layer

```
# code
for name, param in model2.state_dict().items():
    print(f"{name}: {param}")
```

```
linear_relu_stack.0.weight: tensor([[-0.3500, -0.4872,  0.5173],
        [ 0.4641,  0.2517,  0.4589],
        [ 0.2395, -0.3973, -0.4249],
        [-0.2584,  0.0776,  0.0504],
        [ 0.1238, -0.0169,  0.0355]])
linear_relu_stack.0.bias: tensor([ 0.2996,  0.4627, -0.0290, -0.3258,  0.5329])
linear_relu_stack.2.weight: tensor([[ 0.2630,  0.0127,  0.3626, -0.2137,  0.4385],
        [ 0.2120, -0.2872,  0.1858,  0.3297, -0.4443]])
linear_relu_stack.2.bias: tensor([0.4379, 0.0590])
```

l) Now, load the saved model and print the first layer weight and biases again.

```
# code
model2.load_state_dict(torch.load('model.pth',weights_only=True))

for name, param in model2.state_dict().items():
    print(f"{name}: {param}")
```

```
linear_relu_stack.0.weight: tensor([[ 0.3766,  0.2322,  0.3454],
        [ 0.0601, -0.2212,  0.0133],
        [-0.5591,  0.0329, -0.4415],
        [ 0.4316, -0.1069,  0.1645],
        [ 0.2800,  0.2084, -0.4260]])
linear_relu_stack.0.bias: tensor([ 0.0465,  0.5132,  0.0162, -0.4779,  0.4276])
linear_relu_stack.2.weight: tensor([[ 0.3819, -0.0871,  0.3002,  0.1626,  0.2033],
        [ 0.4259,  0.3017,  0.0636,  0.1249, -0.3869]])
linear_relu_stack.2.bias: tensor([0.0409, 0.1808])
```

- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing
- padding for printing padding for printing