# Fashion-MNIST

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>T-Shirt/Top</td>
<td><img src="image1" alt="T-Shirt/Top Examples" /></td>
</tr>
<tr>
<td>1</td>
<td>Trouser</td>
<td><img src="image2" alt="Trouser Examples" /></td>
</tr>
<tr>
<td>2</td>
<td>Pullover</td>
<td><img src="image3" alt="Pullover Examples" /></td>
</tr>
<tr>
<td>3</td>
<td>Dress</td>
<td><img src="image4" alt="Dress Examples" /></td>
</tr>
<tr>
<td>4</td>
<td>Coat</td>
<td><img src="image5" alt="Coat Examples" /></td>
</tr>
<tr>
<td>5</td>
<td>Sandals</td>
<td><img src="image6" alt="Sandals Examples" /></td>
</tr>
<tr>
<td>6</td>
<td>Shirt</td>
<td><img src="image7" alt="Shirt Examples" /></td>
</tr>
<tr>
<td>7</td>
<td>Sneaker</td>
<td><img src="image8" alt="Sneaker Examples" /></td>
</tr>
<tr>
<td>8</td>
<td>Bag</td>
<td><img src="image9" alt="Bag Examples" /></td>
</tr>
<tr>
<td>9</td>
<td>Ankle boots</td>
<td><img src="image10" alt="Ankle boots Examples" /></td>
</tr>
</tbody>
</table>

Tensors
Initializing a Tensor

**Directly from data**

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

```python
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](#)).

```python
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```
Initializing a Tensor

**From another tensor:**

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

```python
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([[0.9152, 0.2666],
        [0.0863, 0.9133]])
```
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
Operations on Tensors

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!

```python
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to('cuda')
```
Operations on Tensors

Standard numpy-like indexing and slicing:

tensor = torch.ones(4, 4)
print('First row: ', tensor[0])
print('First column: ', tensor[:, 0])
print('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.]])
Operations on Tensors

**Arithmetic operations**

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

y3 = torch.rand_like(tensor)
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)
```
Datasets & DataLoaders
Loading a Dataset

```python
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor, Lambda
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()
)
```
Iterating and Visualizing the Dataset

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

```python
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()
```
Iterating and Visualizing the Dataset
Creating a Custom Dataset for your files

```python
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)
        sample = {'image': image, 'label': label}
        return sample
```
Creating a Custom Dataset for your files

```python
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)
        sample = {'image': image, 'label': label}
        return sample
```
Creating a Custom Dataset for your files

```python
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)
        sample = {"image": image, "label": label}
        return sample
```
Creating a Custom Dataset for your files

```python
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)
        sample = {'image': image, 'label': label}
        return sample
```
Preparing your data for training with DataLoader

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.

```python
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)
```
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 a `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`).

```python
# 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}"
```
Iterate through the DataLoader

Out:

Feature batch shape: torch.Size([64, 1, 28, 28])
Labels batch shape: torch.Size([64])
Label: 8
Neural Network
torch.nn.Module

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

```python
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print('Using {} device'.format(device))
```

Out:

```
Using cuda device
```
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),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
Define the Class

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

```python
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): ReLU()
    )
)
```
Define the Class

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 10-dimensional tensor with raw predicted values for each class. We get the prediction probabilities by passing it through an instance of the `nn.Softmax` module.

```python
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')
```
Optimizing Model Params
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.

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

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

```python
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.backwards()`. 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.
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}]"
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    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 /= size
    correct /= size

    print(f"Test Error:  
Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} 
")
```python
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!")
```
Full Implementation

Out:

Epoch 1
-----------------------------------
loss: 2.299511 [ 0/60000]
loss: 2.301767 [ 6400/60000]
loss: 2.289777 [12800/60000]
loss: 2.291731 [19200/60000]
loss: 2.269755 [25600/60000]
loss: 2.261175 [32000/60000]
loss: 2.258553 [38400/60000]
loss: 2.240743 [44800/60000]
loss: 2.260818 [51200/60000]
loss: 2.243683 [57600/60000]
Test Error:
  Accuracy: 37.3%, Avg loss: 0.035121

Epoch 2
-----------------------------------
loss: 2.229830 [ 0/60000]
loss: 2.241497 [ 6400/60000]
loss: 2.221580 [12800/60000]
Save and Load the Model
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:

```python
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.

```python
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()
```
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:

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

We can then load the model like this:

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

• PyTorch Official Tutorial: https://pytorch.org/tutorials/beginner/basics/intro.html

• Feel free to check out the tutorial for more details!