```python
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
```

1

```python
# Data Preprocessing
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize( mean: (0.5,), std: (0.5,))])

trainset = torchvision.datasets.FashionMNIST(
    root='./data', train=True, download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=32, shuffle=True)

testset = torchvision.datasets.FashionMNIST(
    root='./data', train=False, download=True, transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)
```

1.1 1.2 1.3 1.4 1.5

1. Data Preprocessing

1.1 Transformations

transforms.Compose: takes a list of transformations and applies sequentially to the data. In this case, the transformations are:

- transforms.ToTensor(): Converts PIL (Python Imaging Library) img or Numpy ndarray to a pytorch tensor.

Tensor shape (1, 28, 28)

- The number of **channels** (since it's grayscale, there's only 1 channel; RGB images would have 3 channels).
- 28: The **height** of the image.
- 28: The **width** of the image.

Tensor holding pixel intensity values

```
[
  [0, 0, 0, ..., 255, 0, 0, 0],
  [0, 0, 255, ..., 255, 255, 0, 0],
  [0, 255, 0, ..., 0, 0, 255, 0],
  ...,
  [0, 255, 255, ..., 255, 255, 0],
  [0, 0, 0, ..., 0, 255, 0, 0],
  [0, 0, 0, ..., 255, 0, 0, 0],
]
```

- transforms.Normalize((0.5),(0.5)): Scales pixel values in the tensor to range [-1, 1].

  normalization formula: $\text{Normalized value} = \frac{\text{Original Value} - \text{mean}}{\text{Standard deviation}}$

Since both mean and sd are set to 0.5, pixel values will be rescaled from a range [0, 1] to [-1, 1] (After being converted to tensors).

Why normalize?

- Speeds up convergence: normalized data helps optimizer find minimum of loss function quicker.
- Prevents exploding / vanishing gradients: values in small ranges keep gradients manageable.

1.1 Loading of FashionMNIST dataset

(1) torchvision.datasets.FashionMNIST : downloads and loads the FashionMNIST dataset, which consists of grayscale images of clothing items. Each img is 28 x 28, belongs to one of ten categories.

(2) PARAMETERS:

- ✯ root = './data' : specifies directory (./data) where dataset will be stored or loaded from.
- ✯ train = True : loads traing set. if false, loads testing set.
- ✯ download = True: if dataset not already in the specified root directory, it will be downloaded.
- ✯ transform = transform : applies the earlier defined transformation pipeline.

Fashion-MNIST is a more challenging version of MNIST for testing classification models, often being used as a benchmark dataset in research to develop/test new architectures.

1.3 Trainloader

1. `trainloader = torch.utils.data.DataLoader(trainset, batch_size=32, shuffle=True)`
This creates a DataLoader object which will load data from the trainset.

2. PARAMETERS

★ `torch.utils.data.DataLoader`: Pytorch utility that provides an efficient way to load datasets. It loads data in mini-batches, allowing for easier handling of large datasets, and optionally shuffles the data.

★ `batch_size=32`: defines size of each mini-batch. Loads 32 images at a time. This is crucial for efficient training, as it allows models to update weights based on an average of 32 samples rather than one, reducing noise and helping with more stable learning.

★ `shuffle=True`: shuffling the data at each epoch is important for preventing the model from learning order-specific patterns in the data. It ensures each mini-batch contains random samples, which helps with model generalization.

Why use DataLoader?

- Batching: loads multiple samples at once, makes training faster and less memory intensive.
- Shuffling: helps avoid overfitting by ensuring that the model does not rely on training data order.
- Efficiency: handles complexity of loading and preparing batches, especially in large datasets, and can use parallelism (multiple workers) to speed up data fetching.

1.4 Testset

1.
```
testset = torchvision.datasets.FashionMNIST(root='./data', train=False,
download=True, transform=transform)
```
Loads MNIST testing data.

2. PARAMETERS :

- ☆ **root='./data'** : directory where dataset will be stored (or loaded from)
- ☆ **train=False** : specifies loading of train set.
- ☆ **download=True** : downloads trainset (if not already)
- ☆ **transform=transform** : same transformations as in training set

By loading test data separately we evaluate the models performance on unseen data. Said data ensures the model generalizes well and isn't overfitting to the training data.

1.5 Testloader

1. `testloader = torch.utils.data.DataLoader(testset, batch_size=32, shuffle=False)`

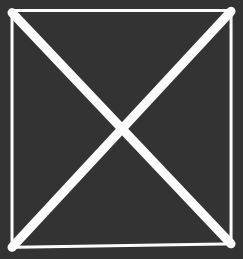
   Creates a Dataloader object to load batches of data from the test set.

2. PARAMETERS :

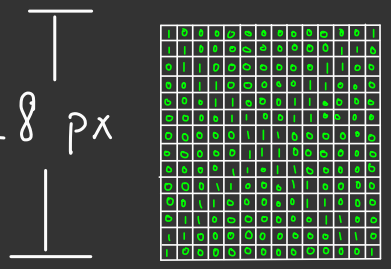
   - ☆ testset: Fashion-MNIST dataset
   - ☆ batch_size = 32: 32 images at a time
   - ☆ shuffle = False: shuffling is unnecessary for testing bc we want to eval the model on the test data in its natural order.

Network representation - Fully connected layer vs CNN.

Grayscale img: #1



Pixel representation: #2



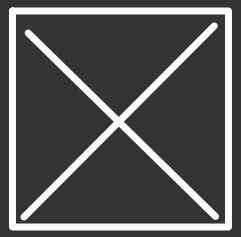
784 pixels

Fully connected network

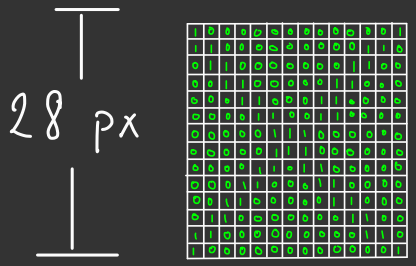
vs

CNN

Grayscale img: #1

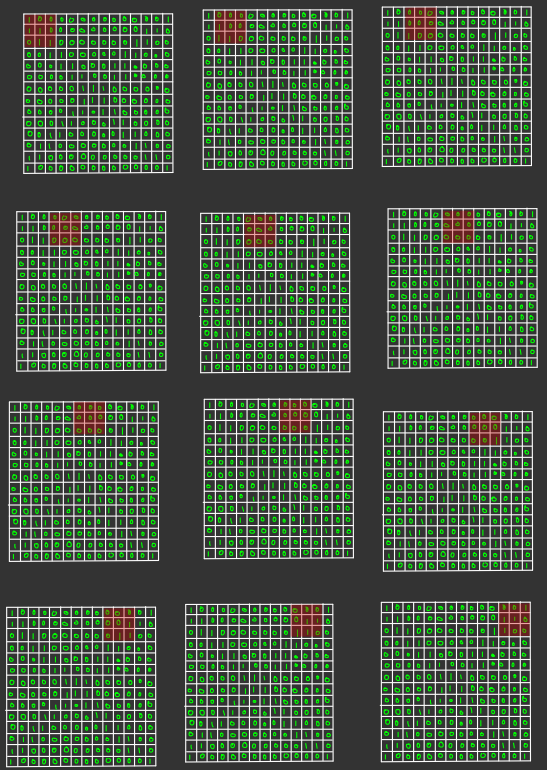


Pixel representation: #2



784 pixels

Also first input, img is treated as a 28x28 grid and passed directly to convolutional layer which processes img in its 2D form.

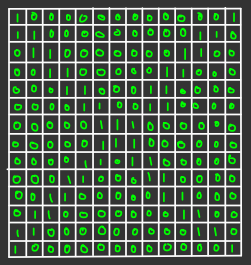


The amount the kernel moves each step depends on the stride size.

↳ The filter is initialized randomly, passing over every section of the input, at each point, the product point is calculated, producing a feature map with the results. The feature map is essentially a transformation of the input data, highlighting areas where certain features are detected, like edges. The filter values get refined through backpropagation.

Input img:

Pixel rep:



Filter (kernel):

0.3	1.1	-0.6
0.4	-0.6	-0.3
0.7	0.1	0 5

Weights = Raw value (Dot product)

Activation Function (Raw value) → Neuron output "0"

f(x)

Feature map (output):

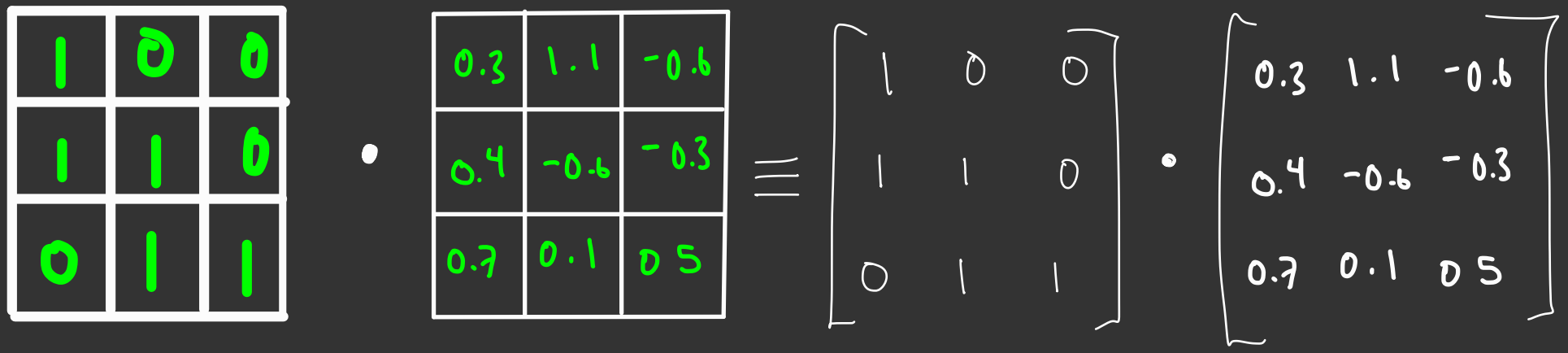
O_1	O_2	O_3
O_4	O_5	O_6
O_7	O_8	O_9

Feature map dimension formula:

$$\text{Output size} = \frac{\text{Input size} - \text{filter size} + 2 \times \text{Padding}}{\text{Stride}} + 1$$

*Different filters can detect different things like edges, textures or other features.

The process defined below is performed every single time a filter moves.



$= (1 \times 0.3) + (0 \times 1.1) + (0 \times -0.6) + (1 \times 0.4) + (1 \times -0.6) +$

$(0 \times -0.3) + (0 \times 0.7) + (1 \times 0.1) + (1 \times 0.5) = 0.7$

$0.7 \rightarrow$ Dot product

Activation function (Dot product) $\rightarrow$ ReLU

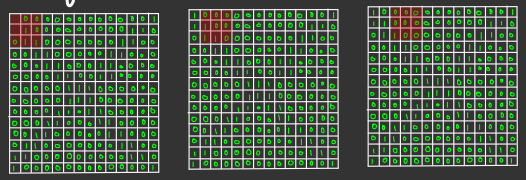
$$ReLU(x) \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

$ReLU(0.7) \rightarrow 0.7$

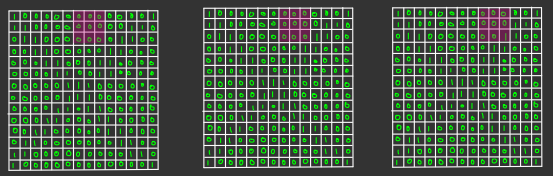
$ReLU(-0.7) \rightarrow 0$

Different filters

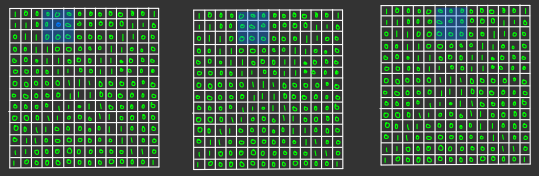
Edges



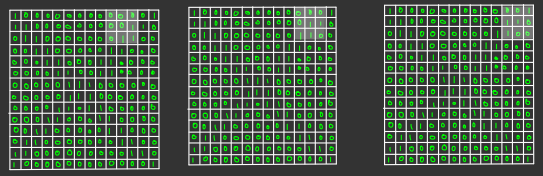
Textures



Corners



complex features



Summary of Filter Types:

- **Early layers** (close to the input): Focus on **low-level features** like edges and textures.
- **Mid layers**: Detect **combinations of these low-level features** to form shapes, like corners or curves.
- **Deeper layers**: Detect more **abstract and specific features**, such as object parts (e.g., eyes, or wheels of a car).

```python
# Define the CNN architecture
2 usages
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d( in_channels: 1,  out_channels: 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d( in_channels: 16,  out_channels: 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(32 * 7 * 7,  out_features 128)
        self.fc2 = nn.Linear( in_features: 128,  out_features 10)

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 32 * 7 * 7)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```python
# Initialize the CNN model
model = CNN()

# Define the loss function (CrossEntropyLoss) and the optimizer (Adam)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

7

```python
# Training loop
epochs = 10

for epoch in range(epochs):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {running_loss / len(trainloader)}")

print("Training Finished!")
```

```python
# Testing the trained model
correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")
```