Fashion-MNIST Image Classification

Yuan Gao

UCLA Statistics & Data Science

Abstract

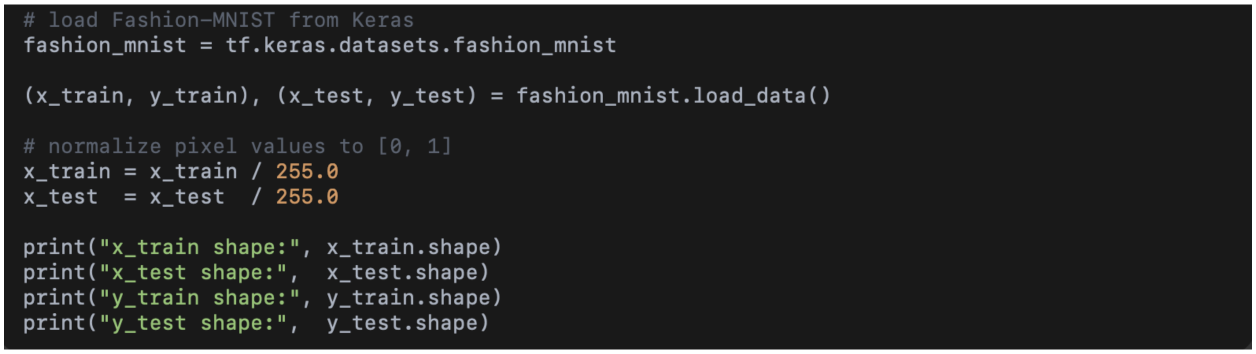
In this project, I compare two supervised learning models on the Fashion-MNIST dataset: an MLPClassifier and a Convolutional Neural Network (CNN). The MLP must flatten each image into a long vector, while the CNN keeps the 2-D structure. I train both models and test them on the same dataset to see how they perform. The results show that the MLP reaches 87.09% accuracy, and the CNN reaches 89.53% accuracy. This project helps me understand how different model structures affect image classification performance.

1 Introduction

The goal of this project is to classify clothing images from the Fashion-MNIST dataset. This dataset contains 70,000 grayscale images of size 28×28. There are ten categories, such as T-shirt/top, Trouser, Dress, Coat, Bag, and more. Each image has one correct label. In class, we learned about simple neural networks. Professor explained that MLPs flatten the image, which removes the spatial patterns inside it. CNNs keep the image shape and usually perform better on image tasks. Because of this, I train both models and compare their performance. This helps me understand why CNNs are more effective for image data.

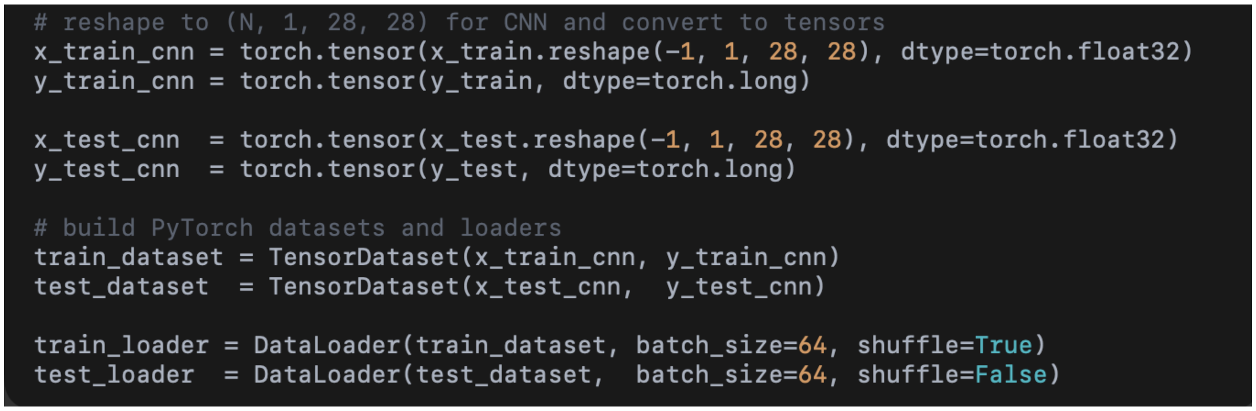
2 Data

This project uses the Fashion-MNIST dataset from Zalando. The dataset has 70,000 grayscale images of clothing items, each with size 28×28 pixels. There are 10 classes: T-shirt/top, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, and Ankle boot. Each image has one label from 0 to 9 that tells which clothing class it belongs to. To prepare the data for modeling, I first load the dataset and scale the pixel values to be between 0 and 1. This is done with TensorFlow, using code like:



```
# load Fashion-MNIST from Keras
fashion_mnist = tf.keras.datasets.fashion_mnist

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# normalize pixel values to [0, 1]
x_train = x_train / 255.0
x_test  = x_test  / 255.0

print("x_train shape:", x_train.shape)
print("x_test shape:",  x_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:",  y_test.shape)
```

The CNN part of the project is written in PyTorch, so I convert the NumPy arrays into PyTorch tensors and wrap them in TensorDataset and DataLoader. The main lines are:



```
# reshape to (N, 1, 28, 28) for CNN and convert to tensors
x_train_cnn = torch.tensor(x_train.reshape(-1, 1, 28, 28), dtype=torch.float32)
y_train_cnn = torch.tensor(y_train, dtype=torch.long)

x_test_cnn  = torch.tensor(x_test.reshape(-1, 1, 28, 28), dtype=torch.float32)
y_test_cnn  = torch.tensor(y_test, dtype=torch.long)

# build PyTorch datasets and loaders
train_dataset = TensorDataset(x_train_cnn, y_train_cnn)
test_dataset  = TensorDataset(x_test_cnn,  y_test_cnn)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader  = DataLoader(test_dataset,  batch_size=64, shuffle=False)
```

This lets the CNN read mini batches of images with shape (batch_size, 1, 28, 28), which is the standard format for convolutional layers.

3 Modeling

I trained two models: an MLPClassifier and a CNN. Both use supervised learning and predict one of ten classes.

Basic Neural Network (MLPClassifier) The MLP is a simple neural network. It cannot use the 2-D structure of the image, so each image is flattened into a 784-length vector.

```
# flatten images for scikit-learn MLP
x_train_mlp = x_train.reshape(len(x_train), -1)
x_test_mlp  = x_test.reshape(len(x_test), -1)

print("Flattened shape:", x_train_mlp.shape)
```

I define and train the MLP model. In my project I used one hidden layer with ReLU activation and a softmax output:

```
mlp = MLPClassifier(
    hidden_layer_sizes=(256,),
    activation="relu",
    solver="adam",
    max_iter=20,
    random_state=0
)

mlp.fit(x_train_mlp, y_train)
y_pred_mlp = mlp.predict(x_test_mlp)

mlp_accuracy = accuracy_score(y_test, y_pred_mlp)
print("MLP test accuracy:", mlp_accuracy)
```

This model is simple and fast to train. However, it treats every pixel as an independent input feature and does not know that nearby pixels belong to the same local pattern.

Convolutional Neural Network (CNN) The CNN keeps the 28×28 structure of the image and uses convolution layers to learn local patterns such as edges, corners, and textures.

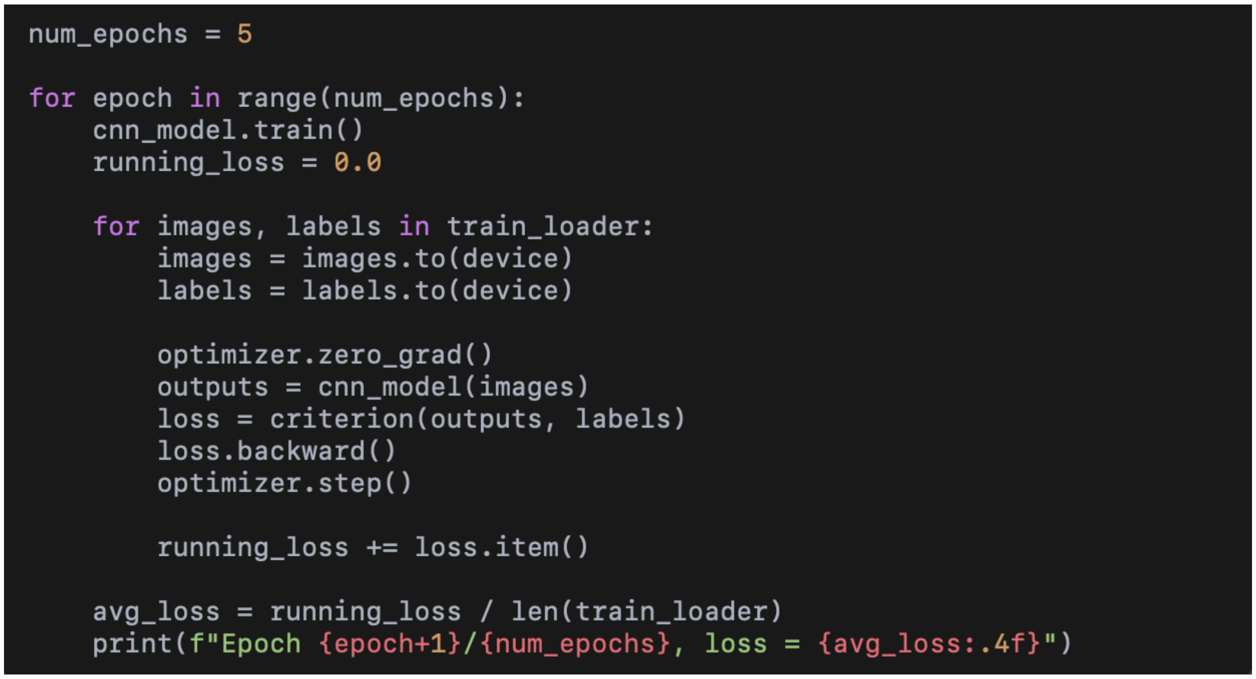
First, I define the model:

```
class FashionCNN(nn.Module):
    def __init__(self):
        super(FashionCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool  = nn.MaxPool2d(2, 2)
        self.fc1   = nn.Linear(64 * 7 * 7, 128)
        self.fc2   = nn.Linear(128, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))   # -> (32, 14, 14)
        x = self.pool(F.relu(self.conv2(x)))   # -> (64, 7, 7)
        x = x.view(x.size(0), -1)              # flatten
        x = F.relu(self.fc1(x))
        x = self.fc2(x)                        # logits for 10 classes
        return x

cnn_model = FashionCNN()
```

I use cross-entropy loss and the Adam optimizer:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
cnn_model.to(device)

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn_model.parameters(), lr=1e-3)
```

The training loop goes through the training set for 5 epochs:



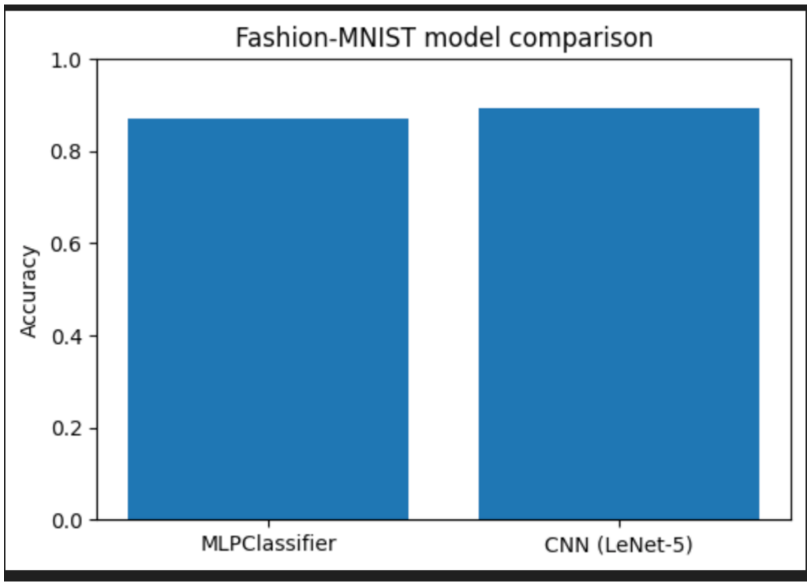
```python
num_epochs = 5

for epoch in range(num_epochs):
    cnn_model.train()
    running_loss = 0.0

    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
        outputs = cnn_model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

    avg_loss = running_loss / len(train_loader)
    print(f"Epoch {epoch+1}/{num_epochs}, loss = {avg_loss:.4f}")
```

Because the CNN keeps the 2-D structure of the image and learns local patterns through convolution, it achieves higher accuracy than the MLPClassifier. This supports Professor's comment: flattening the image loses spatial information, while CNNs can use that structure and work better on image classification tasks.

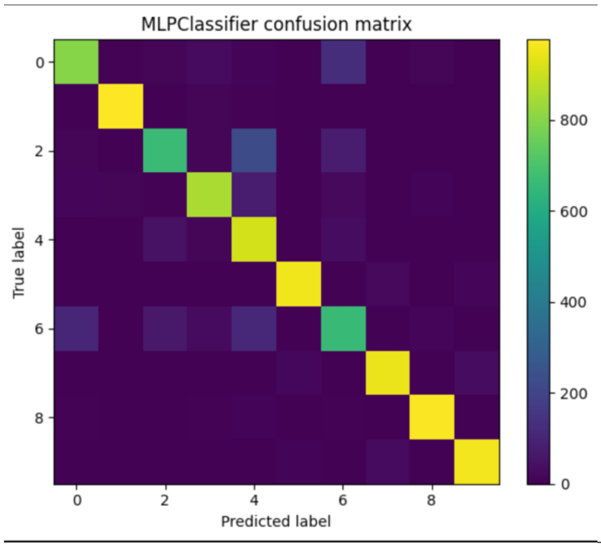
4 Results

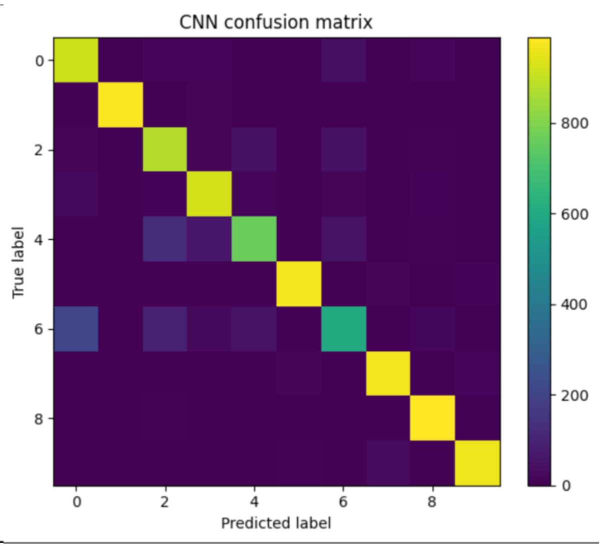
After training both models, I evaluated them on the 10,000-image test set.

Accuracy Results:



The MLP confusion matrix shows more errors for similar classes like: The CNN makes fewer mistakes because it can detect local image features.





In the MLP matrix, the medium-purple squares at (0,6) and (6,0) for T-shirt vs Shirt, and at (2,4) and (4,2) for Pullover vs Coat, show the model makes more mistakes, while the CNN matrix is darker in these spots, meaning the CNN separates these similar classes better.

5 Discussion

The results demonstrate that the CNN performs better than the MLP. This makes sense for me to understand their differences since the MLP flattens the image and loses the spatial layout, the MLP has trouble learning important visual patterns and edges. This causes mistakes when classes look similar. The CNN keeps the 2-D shape. Its filters learn local patterns. Because of this, the CNN achieves 89.53% accuracy, which is higher than the MLP's 87.09%. Training time also differs. The MLP trains very fast, while the CNN takes longer. But the accuracy gain from the CNN is meaningful, so the extra time is worth it for image tasks. These results support professor's explanation that simple feed-forward networks are not the best choice for images. Convolution helps the model understand shape and structure, which is important for visual data.

6 Conclusion

This project was my first step in comparing different models with real dataset, I learned how to build a basic MLP and a simple CNN, and how to train, test, and compare them in a simple way. In the future, I might use the same way of comparison on more advanced models. Using the same style as this project, I can measure how much each change really helps. This project gives me a basic template for my future work, define a clear task, build two or more reasonable models, and then compare them with simple metrics and plots. I can reuse this comparison framework when I study new architectures in machine learning, so I can understand not only which model is better, but why it is better and in what situations it should be used.