Table of Contents:

```
!pip install -qU flaml

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset
from flaml import tune
import matplotlib.pyplot as plt
import seaborn as sns
```

Step 1: Brief description of the problem and data

This project will look at the Fashion MNIST dataset consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

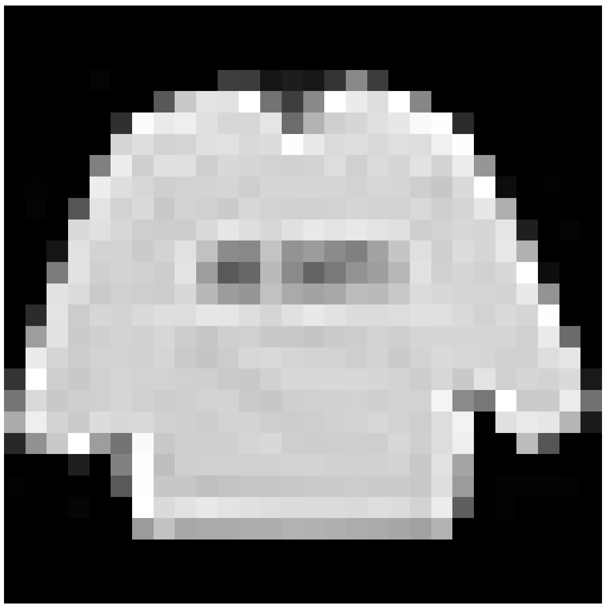
This project seeks to develop a convolutional neural network (CNN) to accurately determine what type of garment is shown in a particular image, such as `T-shirt/top`, `Trouser`, or `Pullover`. The full mapping of all labels is shown below. The next snippet of code below that displays a few examples from the training dataset along with the associated label.

My Github repository link

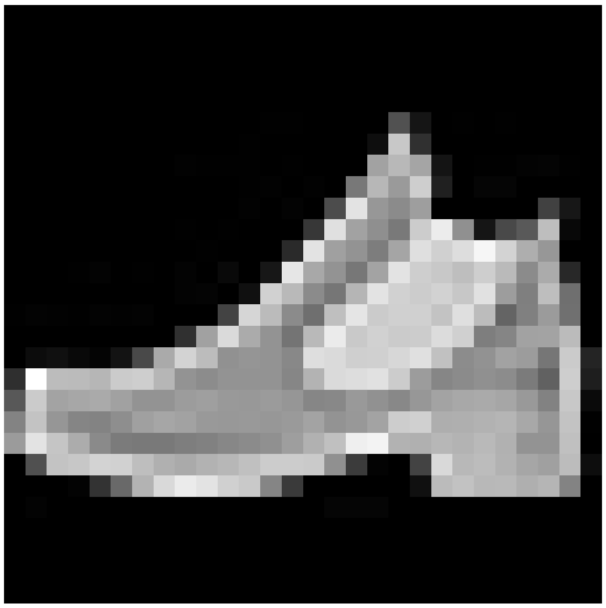
```
# Map the numerical labels (0, 1, 2...) to text labels (T-shirt/top,
Trouser, Pullover...)
label_to_text_map = {
    0: 'T-shirt/top',
    1: 'Trouser',
```

```python
    2: 'Pullover',
    3: 'Dress',
    4: 'Coat',
    5: 'Sandal',
    6: 'Shirt',
    7: 'Sneaker',
    8: 'Bag',
    9: 'Ankle boot'
}

data_loc = 'fashion-mnist-data'

# Load train and test data
train_data = pd.read_csv(data_loc + "/fashion-mnist_train.csv")
Y_train = train_data['label'].values
X_train = train_data.drop(columns=['label']).values / 255.0  # 
Normalize to [0, 1]
test_data = pd.read_csv(data_loc + "/fashion-mnist_test.csv")
Y_test = test_data['label'].values
X_test = test_data.drop(columns=['label']).values / 255.0  # Normalize 
to [0, 1]

# Convert to PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32).reshape(-1, 1, 
28, 28)
Y_train = torch.tensor(Y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float32).reshape(-1, 1, 28, 
28)
Y_test = torch.tensor(Y_test, dtype=torch.long)

# Create PyTorch datasets
train_dataset = TensorDataset(X_train, Y_train)
test_dataset = TensorDataset(X_test, Y_test)

def display_image(data, index=0, title="Label"):
    # The first column is the label and the rest are pixel values
    labels = data.iloc[:, 0]
    images = data.iloc[:, 1:].values

    label = labels[index]
    image = images[index]

    # Reshape the flat image array into a 28x28 grid since all fashion 
MNIST images are 28x28 pixels
    image = image.reshape(28, 28)

    # Display the image
    plt.figure(figsize=(4, 4))
    plt.imshow(image, cmap="gray")
    plt.title(f"{title}: {label_to_text_map[label]} ({label})")
    plt.axis("off")
    plt.show()

# Display a few example images
display_image(train_data, index=0)
display_image(train_data, index=1)
display_image(train_data, index=2)
```

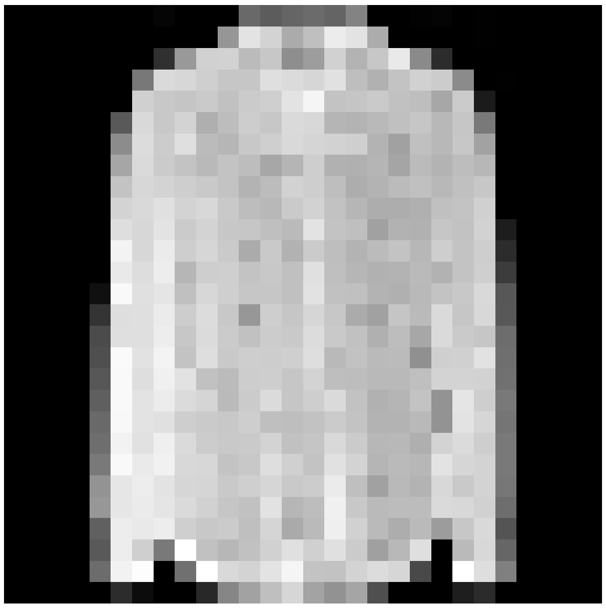
Label: Pullover (2)



Label: Ankle boot (9)



Label: Shirt (6)



I found this dataset interesting because it seeks to replace the well-known MNIST dataset as the benchmark to validate algorithms in the AI/ML/Data Science community. I plan to use the skills I am learning in this course in my future career, so knowing how to use this benchmark dataset along with the MNIST dataset will help me do benchmark testing on any ML models I develop later.

I also found the small size of this dataset appealing (the training dataset is 133MB) because it would allow me to iterate quickly on the model without needing to wait hours to train a single model. I found this especially appealing after week 5's GAN assignment, in which training took 1 or more hours!

Step 2: Exploratory Data Analysis (EDA) - Inspect, Visualize and Clean the Data

In this step, I will take a look at the data set, call out any areas of concern, and summarize my findings at the end.

Step 2.1: Confirmation of provided data

Kaggle states that the dataset contains a training set of 60,000 examples and a test set of 10,000 examples, which are all 28x28 grayscale images. 28x28 = 784, so I would expect the shape to by 785 (including 1 additional column for the data label).

Kaggle states that there are 10 unique data labels (0 through 9).

I trust Kaggle, but let's verify their claims just in case there is a mistake in the data description of the data itself.

```
print("Train data shape:", train_data.shape)
print("Test data shape:", test_data.shape)
print("Unique train labels:", np.unique(Y_train))
print("Unique test labels:", np.unique(Y_test))
```

```
Train data shape: (60000, 785)
Test data shape: (10000, 785)
Unique train labels: [0 1 2 3 4 5 6 7 8 9]
Unique test labels: [0 1 2 3 4 5 6 7 8 9]
```

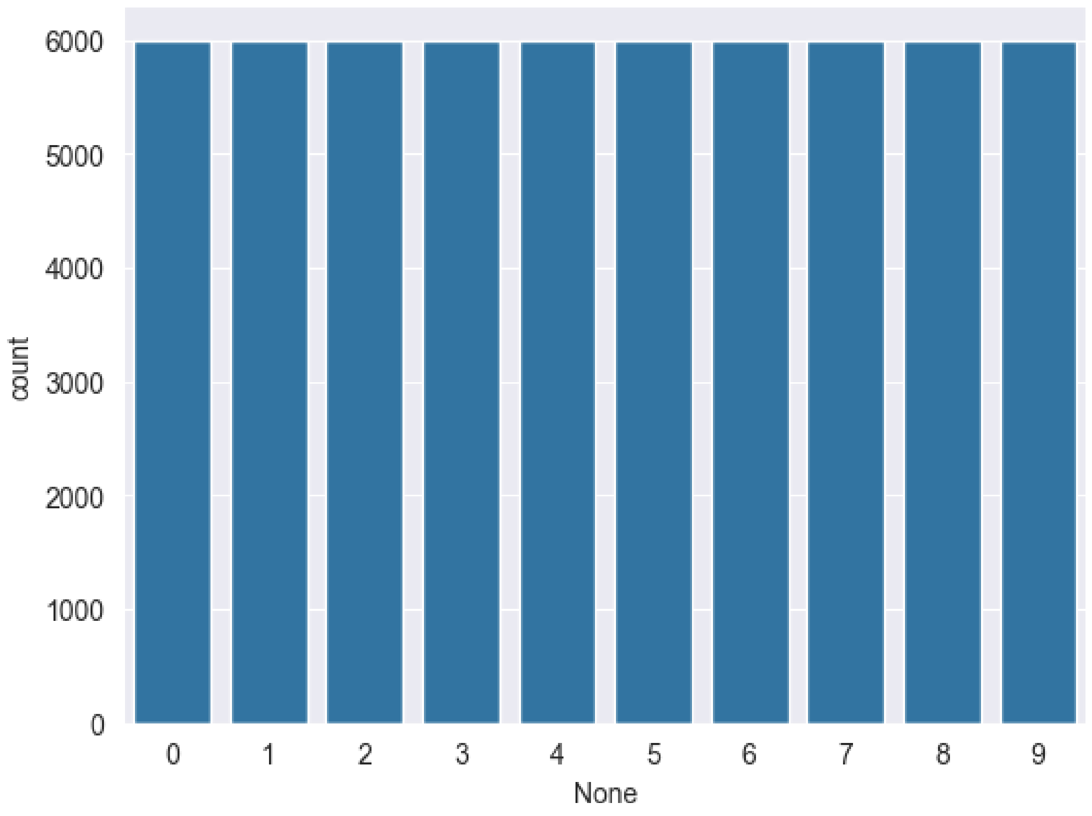
The data shape and number of unique labels matches the description of the data in Kaggle.

Step 2.2: Label distribution

Next I'll check the distribution of samples across the classes to ensure there is no significant imbalance. If there is, I may have to oversample the minority classes, undersample the majority classes, or perform data augmentation such as by applying transformations like rotating, flipping, or cropping.

```
sns.countplot(x=Y_train)
```

```
<Axes: xlabel='None', ylabel='count'>
```



The training data is perfectly balanced (!!) with 6,000 samples in each of 10 categories.

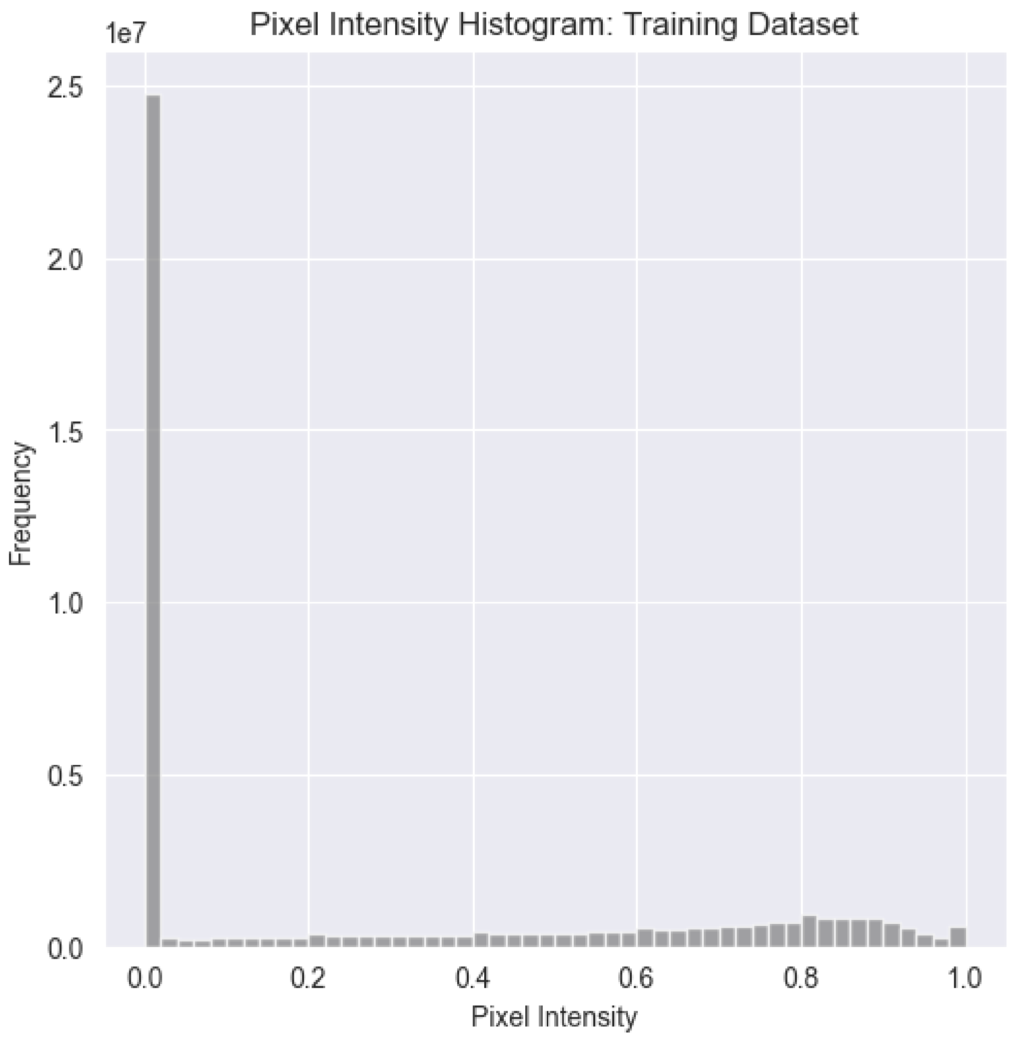
Step 2.3: Pixel intensity analysis

Here I'll check the range of pixel intensities. These are typically 0–255 for grayscale images, with 0 = white and 255 = black, but I've normalized this dataset above to fall between 0-1.

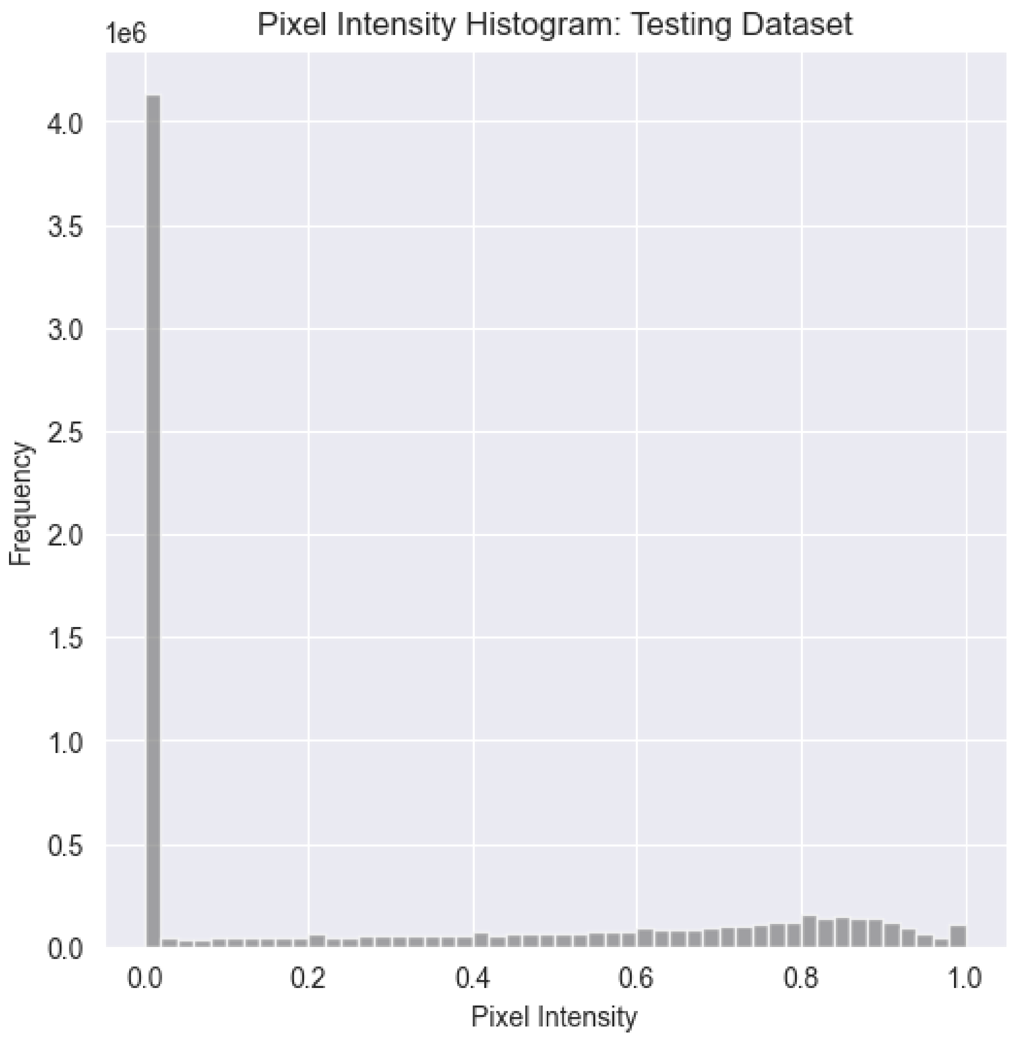
```
# Function to compute and plot pixel intensity analysis
def pixel_intensity_analysis(dataset, title="Dataset"):
    # Flatten the images to compute pixel-wise statistics
    images = dataset.tensors[0].view(-1, 28*28)  # Flatten to 1D
vector (28*28)

    # Compute pixel intensity statistics
    pixel_mean = images.mean().item()  # Mean pixel intensity
    pixel_std = images.std().item()    # Standard deviation
    pixel_min = images.min().item()    # Minimum pixel intensity
    pixel_max = images.max().item()    # Maximum pixel intensity

    # Print the statistics
    print(f"{title} Pixel Intensity Analysis:")
    print(f"Mean Pixel Intensity: {pixel_mean:.4f}")
    print(f"Standard Deviation: {pixel_std:.4f}")
    print(f"Min Pixel Intensity: {pixel_min:.4f}")
    print(f"Max Pixel Intensity: {pixel_max:.4f}")

    # Plot histogram of pixel intensities
    plt.figure(figsize=(6, 6))
    plt.hist(images.flatten().numpy(), bins=50, color='gray',
alpha=0.7)
    plt.title(f"Pixel Intensity Histogram: {title}")
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')
    plt.show()

# Perform pixel intensity analysis on training and testing datasets
pixel_intensity_analysis(train_dataset, "Training Dataset")
pixel_intensity_analysis(test_dataset, "Testing Dataset")

def display_image_with_histogram(dataset, index=0):
    # Get the image and label at the given index
    image = dataset.tensors[0][index].squeeze().numpy()  # (1, 28, 28)
-> (28, 28)
    label = dataset.tensors[1][index].item()

    # Plot the image and its pixel intensity histogram
    plt.figure(figsize=(12, 6))

    # Plot image
    plt.subplot(1, 2, 1)
    plt.imshow(image, cmap='gray')
    plt.title(f"Label: {label_to_text_map[label]} ({label})")
```

```python
    plt.axis('off')

    # Plot pixel intensity histogram
    plt.subplot(1, 2, 2)
    plt.hist(image.flatten(), bins=50, color='gray', alpha=0.7)
    plt.title(f"Pixel Intensity Histogram")
    plt.xlabel('Pixel Intensity')
    plt.ylabel('Frequency')

    plt.tight_layout()
    plt.show()

# Display a single example image with its pixel intensity histogram
from the training dataset
display_image_with_histogram(train_dataset, index=0)
```

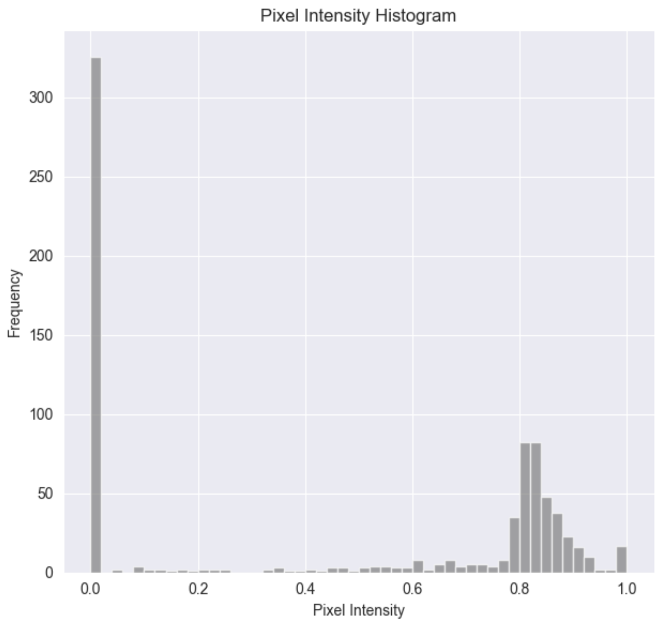
```
Training Dataset Pixel Intensity Analysis:
Mean Pixel Intensity: 0.2861
Standard Deviation: 0.3528
Min Pixel Intensity: 0.0000
Max Pixel Intensity: 1.0000
```



```
Testing Dataset Pixel Intensity Analysis:
Mean Pixel Intensity: 0.2869
Standard Deviation: 0.3540
Min Pixel Intensity: 0.0000
Max Pixel Intensity: 1.0000
```


Pixel Intensity Histogram: Testing Dataset
1e6
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Frequency
0.0
0.2
0.4
0.6
0.8
1.0
Pixel Intensity

Label: Pullover (2)

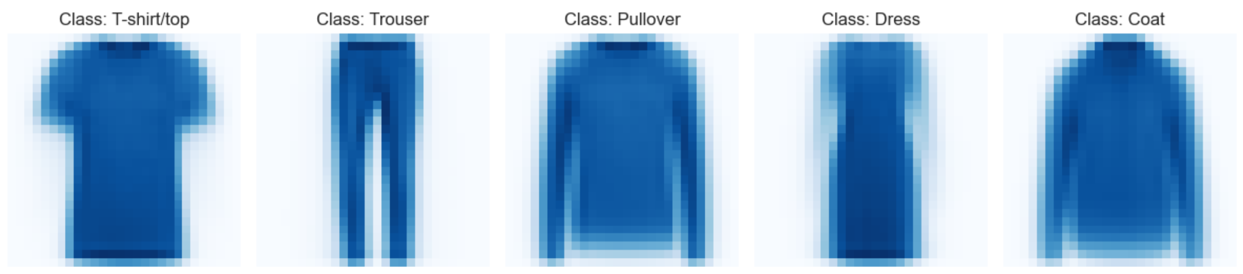

Pixel Intensity Histogram
300
250
200
150
100
50
0
Frequency
0.0
0.2
0.4
0.6
0.8
1.0
Pixel Intensity

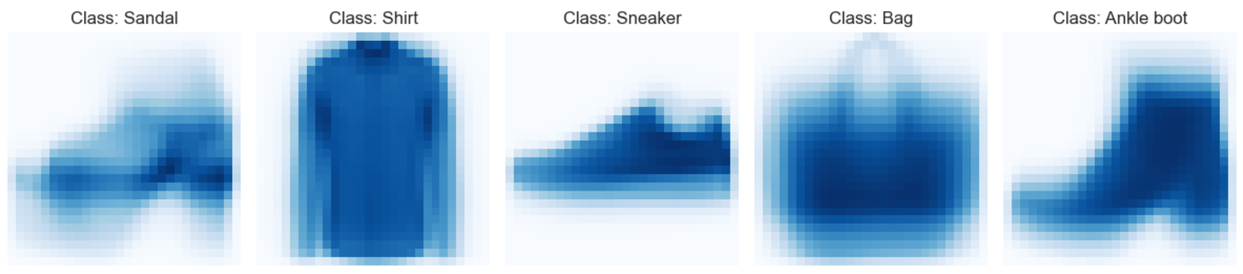
- Both the training and test datasets have a very large spike at the beginning of the pixel intensity distributions due to the white backgrounds used in the images.
- Both datasets have a small spike at the end of the pixel intensity distributions due to the black outline used in some images.
- The distribution of pixel intensities across the training dataset looks nearly identical to the distribution across the test dataset.
- If you ignore the white (0) and black (1) intensities, the pixel intensity distribution shows an average around 0.8, with the bell curve favoring the darker intensities between 0.6-0.9. There are much fewer pixels in the lower intensities 0.1-0.5.

Step 2.4: Class heatmaps

This section will generate and display heatmaps of averaged images for each class (i.e. "T-shirt/top", "Trouser")

```
def generate_class_heatmaps(dataset):
    # Initialize a list to store images for each class
    class_images = {i: [] for i in range(10)}

    # Loop through the dataset and group images by their labels
    for i in range(len(dataset)):
        image, label = dataset[i]
        class_images[label.item()].append(image.squeeze().numpy())  # 
Remove channel dimension for 2D image

    # Plot averaged images with heatmaps for each class
    plt.figure(figsize=(12, 12))

    for label, images in class_images.items():
        # Compute the average image for each class
        avg_image = np.mean(images, axis=0)

        # Plot the averaged image as a heatmap
        plt.subplot(2, 5, label + 1)
        sns.heatmap(avg_image, cmap='Blues', square=True, cbar=False, 
xticklabels=False, yticklabels=False)
        plt.title(f"Class: {label_to_text_map[label]}")

    plt.tight_layout()
    plt.show()

generate_class_heatmaps(train_dataset)
```





The clarity of the heatmaps above tells us a few interesting things.

1. The shapes in these images are easily recognizable to the human eye.
2. The training dataset likely does not contain many rotated or skewed images (perhaps it contains none), and instead contains images with the garment right side up with `Sandal`/`Sneaker`/`Ankle boot` with the toes pointed left.
3. The training dataset likely contains most/all images captured looking at the item from a consistent angle. For example, a photo of a shirt in real life may be taken with the shirt hanging on a hanger or balled up on the floor, but these heatmaps suggest that all `Shirt` images in this dataset are taken with the shirt hanging on a hanger as viewed from the angle where the shirt appears as if it's being worn by a person looking at the camera.
4. `Sandals` and `Bags` have the most "fuzziness" in the heatmaps, indicating that these categories are less consistent than the others. A bit of "fuzziness" can also be seen `Dress` and `Shirt` categories in the sleeves, which intuitively makes sense since some dresses and shirts have sleeves while others don't.

Based on the consistency of this dataset, I am optimistic that I will be able to generate a well-performing model. Let's see if I'm right!

Step 2.5: Blank images

```
def identify_blank_images(dataset):
    blank_images = []

    for i in range(len(dataset)):
        image, label = dataset[i]

        # Flatten the image to 1D for analysis
        flattened_image = image.squeeze().numpy().flatten()

        # Check if the image is blank (all pixels are zero)
        if np.all(flattened_image == 0):
            blank_images.append(i)

    print(f"Number of blank images: {len(blank_images)}")
    return blank_images

# Identify blank and noisy images in the training dataset
blank_images = identify_blank_images(train_dataset)

# Display some examples of blank images
def display_images(dataset, image_indices, title="Images"):
    plt.figure(figsize=(10, 5))
    for i, idx in enumerate(image_indices):
        image = dataset[idx][0].squeeze().numpy()  # Get image and
remove channel dimension
        plt.subplot(1, len(image_indices), i + 1)
        plt.imshow(image, cmap="gray")
        plt.title(f"Image {idx}")
        plt.axis("off")
    plt.suptitle(title)
    plt.tight_layout()
    plt.show()

# Display blank images
if blank_images:
    display_images(train_dataset, blank_images[:5], title="Blank
Images")

Number of blank images: 0
```

Step 2.6: EDA summary

In the EDA above I performed the following 5 tasks

1. **Confirmation of provided data**: The description of the data from Kaggle is accurate
2. **Label distribution**: The training data is perfectly evenly distributed (6,000 samples in each of 10 categories)
3. **Pixel intensity analysis**: White (pixel intensity = 0) is the most common due to the backgrounds. Black (pixel intensity = 255) is relatively common due to the black outline around some items. The distribution of intensities is roughly the same across the training and test datasets.
4. **Class heatmaps**: The images in the training dataset were captured at an angle that is highly consistent with no rotation or skew. For example, all 3 categories of shoes

(`Sandal`, `Sneaker`, and `Ankle boot`) have the base towards the bottom of the image and the toes pointed left.

5. **Blank images**: There are no blank images.

Put simply, this dataset is of very high quality!

Step 3: Model Architecture

I have chosen to use a CNN (convolutional neural network) to view the images in the test dataset and to predict their category (i.e. "T-shirt/top", "Trouser"). I chose a CNN because this type of model is widely used for image classification tasks due to its ability to automatically learn spatial hierarchies of features, which is especially useful for visual pattern recognition.

I will build a CNN with two convolutional layers (`conv1` and `conv2`), followed by batch normalization (`bn1`, `bn2`, `bn3`), pooling, and two fully connected layers (`fc1` and `fc2`).

The constructor accepts parameters for the number of output channels for the convolutional layers (`conv1_out`, `conv2_out`) and the number of neurons in the first fully connected layer (`fc1_out`), allowing some flexibility in the architecture and easy hyperparameter tuning later.

I've chosen to set `stride` to 1 due to the small dataset and expected quick training time. Larger datasets that take a long time to train could speed up training time by increasing the stride.

The training function includes a learning rate scheduler to adjust the learning rate during the training process, possibly leading to faster convergence and better overall performance.

```
# Check if MPS is available.  I am using a Mac M2, so the availability
of "MPS" indicates I can use the GPU to speed up training.
device = torch.device("mps" if torch.backends.mps.is_available() else
"cpu")
print(f"Using device: {device}")
```

```
Using device: mps
```

```
class SimpleCNN(nn.Module):
    def __init__(self, conv1_out=32, conv2_out=64, fc1_out=128):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, conv1_out, kernel_size=3, stride=1,
padding=1)
        self.bn1 = nn.BatchNorm2d(conv1_out)
        self.conv2 = nn.Conv2d(conv1_out, conv2_out, kernel_size=3,
stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(conv2_out)
        self.fc1 = nn.Linear(conv2_out * 7 * 7, fc1_out)
        self.bn3 = nn.BatchNorm1d(fc1_out)
        self.fc2 = nn.Linear(fc1_out, 10)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.max_pool2d(x, 2)
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.max_pool2d(x, 2)
        x = x.view(x.size(0), -1)
        x = F.relu(self.bn3(self.fc1(x)))
        x = self.fc2(x)
        return x

# Training function that accepts input config options including
conv1_out, conv2_out, fc1_out, batch_size, lr, and epochs
def train_fashion_mnist(config):
    train_loader = DataLoader(train_dataset,
batch_size=config['batch_size'], shuffle=True)
    test_loader = DataLoader(test_dataset,
batch_size=config['batch_size'], shuffle=False)
    model = SimpleCNN(conv1_out=config['conv1_out'],
conv2_out=config['conv2_out'], fc1_out=config['fc1_out']).to(device)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=config['lr'])
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=1,
gamma=0.7)  # Step every epoch, reduce by 30%
    train_losses = []
    test_losses = []

    # Training loop
    for epoch in range(config['epochs']):
        model.train()
        epoch_train_loss = 0
        for batch_idx, (data, target) in enumerate(train_loader):
            data, target = data.to(device), target.to(device)
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()
            epoch_train_loss += loss.item()

        # Step the scheduler at the end of each epoch
        scheduler.step()

        # Average training loss for the epoch
        epoch_train_loss /= len(train_loader)
        train_losses.append(epoch_train_loss)

        # Evaluate the model on the test set
        model.eval()
        epoch_test_loss = 0
        correct = 0
        total = 0
        with torch.no_grad():
            for data, target in test_loader:
                data, target = data.to(device), target.to(device)
                output = model(data)
                loss = criterion(output, target)
                epoch_test_loss += loss.item()
                _, predicted = torch.max(output.data, 1)
                total += target.size(0)
                correct += (predicted == target).sum().item()

        # Average test loss and accuracy for the epoch
        epoch_test_loss /= len(test_loader)
        test_losses.append(epoch_test_loss)

        print(f"Epoch {epoch + 1}/{config['epochs']}, "
              f"Train Loss: {epoch_train_loss:.4f}, "
              f"Test Loss: {epoch_test_loss:.4f}, "
              f"Test Accuracy: {correct / total:.4f}, "
              f"Learning Rate: {scheduler.get_last_lr()[0]:.6f}")

    accuracy = correct / total
    return {'accuracy': accuracy, 'model': model, 'train_losses':
train_losses, 'test_losses': test_losses}

baseline_config_space = {
    'conv1_out': 32,
    'conv2_out': 64,
    'fc1_out': 128,
    'batch_size': 64,
    'lr': 1e-2,
    'epochs': 5,
}

baseline_result = train_fashion_mnist(baseline_config_space)

print("Baseline configuration = {")
for key, value in baseline_config_space.items():
    print(f"    '{key}': {value},")
print("}")
print("Baseline accuracy =", baseline_result['accuracy'])
```

```
Epoch 1/5, Train Loss: 0.3539, Test Loss: 0.2528, Test Accuracy:
0.9064, Learning Rate: 0.007000
Epoch 2/5, Train Loss: 0.2277, Test Loss: 0.2180, Test Accuracy:
0.9177, Learning Rate: 0.004900
Epoch 3/5, Train Loss: 0.1796, Test Loss: 0.1920, Test Accuracy:
0.9298, Learning Rate: 0.003430
Epoch 4/5, Train Loss: 0.1372, Test Loss: 0.2182, Test Accuracy:
0.9231, Learning Rate: 0.002401
Epoch 5/5, Train Loss: 0.1025, Test Loss: 0.2002, Test Accuracy:
0.9327, Learning Rate: 0.001681
```

```
Baseline configuration = {
    'conv1_out': 32,
    'conv2_out': 64,
    'fc1_out': 128,
    'batch_size': 64,
    'lr': 0.01,
    'epochs': 5,
}
Baseline accuracy = 0.9327
```

Step 4: Results and Analysis

The baseline model performed decently with an accuracy of 93.27%. Let's see if we can improve that using hyperparameter tuning.

```
# Define the search space for hyperparameter tuning
config_space = {
    'conv1_out': tune.choice([16, 32, 64]),
    'conv2_out': tune.choice([32, 64, 128]),
    'fc1_out': tune.choice([64, 128, 256]),
    'batch_size': tune.choice([32, 64, 128]),
    'lr': tune.loguniform(1e-4, 1e-2),
    'epochs': tune.choice([5, 10, 15]),
}

# Use FLAML (Fast and Lightweight AutoML) to do hyperparameter tuning
analysis = tune.run(
    train_fashion_mnist,
    config=config_space,
    metric='accuracy',
    mode='max',
    num_samples=20,
    resources_per_trial={'cpu': 2, 'gpu': 1},  # Use GPU
    local_dir='./flaml_results',  # Directory to save results
)

Epoch 1/5, Train Loss: 0.4241, Test Loss: 0.2997, Test Accuracy:
0.8943, Learning Rate: 0.000372
Epoch 2/5, Train Loss: 0.2429, Test Loss: 0.2306, Test Accuracy:
0.9159, Learning Rate: 0.000261
Epoch 3/5, Train Loss: 0.2000, Test Loss: 0.2189, Test Accuracy:
0.9201, Learning Rate: 0.000182
Epoch 4/5, Train Loss: 0.1701, Test Loss: 0.2079, Test Accuracy:
0.9218, Learning Rate: 0.000128
Epoch 5/5, Train Loss: 0.1478, Test Loss: 0.2052, Test Accuracy:
0.9239, Learning Rate: 0.000089
Epoch 1/5, Train Loss: 0.3997, Test Loss: 0.2672, Test Accuracy:
```

```
0.9035, Learning Rate: 0.000522
Epoch 2/5, Train Loss: 0.2321, Test Loss: 0.2236, Test Accuracy:
0.9186, Learning Rate: 0.000366
Epoch 3/5, Train Loss: 0.1881, Test Loss: 0.2078, Test Accuracy:
0.9243, Learning Rate: 0.000256
Epoch 4/5, Train Loss: 0.1587, Test Loss: 0.2034, Test Accuracy:
0.9286, Learning Rate: 0.000179
Epoch 5/5, Train Loss: 0.1380, Test Loss: 0.1987, Test Accuracy:
0.9287, Learning Rate: 0.000125
Epoch 1/10, Train Loss: 0.4238, Test Loss: 0.2672, Test Accuracy:
0.9070, Learning Rate: 0.000372
Epoch 2/10, Train Loss: 0.2362, Test Loss: 0.2271, Test Accuracy:
0.9191, Learning Rate: 0.000261
Epoch 3/10, Train Loss: 0.1929, Test Loss: 0.2117, Test Accuracy:
0.9226, Learning Rate: 0.000182
Epoch 4/10, Train Loss: 0.1633, Test Loss: 0.2024, Test Accuracy:
0.9276, Learning Rate: 0.000128
Epoch 5/10, Train Loss: 0.1422, Test Loss: 0.2023, Test Accuracy:
0.9261, Learning Rate: 0.000089
Epoch 6/10, Train Loss: 0.1276, Test Loss: 0.2017, Test Accuracy:
0.9280, Learning Rate: 0.000063
Epoch 7/10, Train Loss: 0.1168, Test Loss: 0.2009, Test Accuracy:
0.9291, Learning Rate: 0.000044
Epoch 8/10, Train Loss: 0.1082, Test Loss: 0.1997, Test Accuracy:
0.9298, Learning Rate: 0.000031
Epoch 9/10, Train Loss: 0.1021, Test Loss: 0.2011, Test Accuracy:
0.9303, Learning Rate: 0.000021
Epoch 10/10, Train Loss: 0.0984, Test Loss: 0.2010, Test Accuracy:
0.9293, Learning Rate: 0.000015
Epoch 1/15, Train Loss: 0.4565, Test Loss: 0.2936, Test Accuracy:
0.8963, Learning Rate: 0.000259
Epoch 2/15, Train Loss: 0.2451, Test Loss: 0.2362, Test Accuracy:
0.9158, Learning Rate: 0.000181
Epoch 3/15, Train Loss: 0.2019, Test Loss: 0.2140, Test Accuracy:
0.9234, Learning Rate: 0.000127
Epoch 4/15, Train Loss: 0.1737, Test Loss: 0.2087, Test Accuracy:
0.9241, Learning Rate: 0.000089
Epoch 5/15, Train Loss: 0.1530, Test Loss: 0.2001, Test Accuracy:
0.9297, Learning Rate: 0.000062
Epoch 6/15, Train Loss: 0.1398, Test Loss: 0.1974, Test Accuracy:
0.9275, Learning Rate: 0.000044
Epoch 7/15, Train Loss: 0.1301, Test Loss: 0.1971, Test Accuracy:
0.9297, Learning Rate: 0.000030
Epoch 8/15, Train Loss: 0.1226, Test Loss: 0.2021, Test Accuracy:
0.9268, Learning Rate: 0.000021
Epoch 9/15, Train Loss: 0.1170, Test Loss: 0.1967, Test Accuracy:
0.9294, Learning Rate: 0.000015
Epoch 10/15, Train Loss: 0.1138, Test Loss: 0.1969, Test Accuracy:
0.9298, Learning Rate: 0.000010
```

```
Epoch 11/15, Train Loss: 0.1118, Test Loss: 0.1975, Test Accuracy:
0.9304, Learning Rate: 0.000007
Epoch 12/15, Train Loss: 0.1092, Test Loss: 0.1974, Test Accuracy:
0.9286, Learning Rate: 0.000005
Epoch 13/15, Train Loss: 0.1074, Test Loss: 0.1978, Test Accuracy:
0.9293, Learning Rate: 0.000004
Epoch 14/15, Train Loss: 0.1069, Test Loss: 0.1967, Test Accuracy:
0.9308, Learning Rate: 0.000003
Epoch 15/15, Train Loss: 0.1063, Test Loss: 0.1976, Test Accuracy:
0.9298, Learning Rate: 0.000002
Epoch 1/10, Train Loss: 0.4227, Test Loss: 0.2669, Test Accuracy:
0.9053, Learning Rate: 0.000372
Epoch 2/10, Train Loss: 0.2395, Test Loss: 0.2347, Test Accuracy:
0.9138, Learning Rate: 0.000261
Epoch 3/10, Train Loss: 0.1987, Test Loss: 0.2134, Test Accuracy:
0.9216, Learning Rate: 0.000182
Epoch 4/10, Train Loss: 0.1695, Test Loss: 0.2067, Test Accuracy:
0.9283, Learning Rate: 0.000128
Epoch 5/10, Train Loss: 0.1487, Test Loss: 0.2118, Test Accuracy:
0.9231, Learning Rate: 0.000089
Epoch 6/10, Train Loss: 0.1340, Test Loss: 0.1982, Test Accuracy:
0.9290, Learning Rate: 0.000063
Epoch 7/10, Train Loss: 0.1227, Test Loss: 0.2019, Test Accuracy:
0.9298, Learning Rate: 0.000044
Epoch 8/10, Train Loss: 0.1141, Test Loss: 0.2006, Test Accuracy:
0.9303, Learning Rate: 0.000031
Epoch 9/10, Train Loss: 0.1088, Test Loss: 0.2014, Test Accuracy:
0.9304, Learning Rate: 0.000021
Epoch 10/10, Train Loss: 0.1049, Test Loss: 0.2016, Test Accuracy:
0.9300, Learning Rate: 0.000015
Epoch 1/10, Train Loss: 0.4107, Test Loss: 0.2627, Test Accuracy:
0.9074, Learning Rate: 0.000409
Epoch 2/10, Train Loss: 0.2335, Test Loss: 0.2213, Test Accuracy:
0.9181, Learning Rate: 0.000286
Epoch 3/10, Train Loss: 0.1917, Test Loss: 0.2123, Test Accuracy:
0.9235, Learning Rate: 0.000200
Epoch 4/10, Train Loss: 0.1620, Test Loss: 0.1950, Test Accuracy:
0.9288, Learning Rate: 0.000140
Epoch 5/10, Train Loss: 0.1405, Test Loss: 0.1936, Test Accuracy:
0.9297, Learning Rate: 0.000098
Epoch 6/10, Train Loss: 0.1238, Test Loss: 0.1957, Test Accuracy:
0.9300, Learning Rate: 0.000069
Epoch 7/10, Train Loss: 0.1120, Test Loss: 0.1903, Test Accuracy:
0.9324, Learning Rate: 0.000048
Epoch 8/10, Train Loss: 0.1043, Test Loss: 0.1895, Test Accuracy:
0.9315, Learning Rate: 0.000034
Epoch 9/10, Train Loss: 0.0974, Test Loss: 0.1899, Test Accuracy:
0.9309, Learning Rate: 0.000024
Epoch 10/10, Train Loss: 0.0935, Test Loss: 0.1924, Test Accuracy:
```

```
0.9301, Learning Rate: 0.000016
Epoch 1/10, Train Loss: 0.4211, Test Loss: 0.2587, Test Accuracy:
0.9039, Learning Rate: 0.000372
Epoch 2/10, Train Loss: 0.2373, Test Loss: 0.2327, Test Accuracy:
0.9159, Learning Rate: 0.000261
Epoch 3/10, Train Loss: 0.1937, Test Loss: 0.2156, Test Accuracy:
0.9202, Learning Rate: 0.000182
Epoch 4/10, Train Loss: 0.1640, Test Loss: 0.2054, Test Accuracy:
0.9264, Learning Rate: 0.000128
Epoch 5/10, Train Loss: 0.1423, Test Loss: 0.2056, Test Accuracy:
0.9264, Learning Rate: 0.000089
Epoch 6/10, Train Loss: 0.1278, Test Loss: 0.2025, Test Accuracy:
0.9268, Learning Rate: 0.000063
Epoch 7/10, Train Loss: 0.1151, Test Loss: 0.2032, Test Accuracy:
0.9274, Learning Rate: 0.000044
Epoch 8/10, Train Loss: 0.1080, Test Loss: 0.2053, Test Accuracy:
0.9273, Learning Rate: 0.000031
Epoch 9/10, Train Loss: 0.1019, Test Loss: 0.2046, Test Accuracy:
0.9283, Learning Rate: 0.000021
Epoch 10/10, Train Loss: 0.0981, Test Loss: 0.2058, Test Accuracy:
0.9281, Learning Rate: 0.000015
Epoch 1/5, Train Loss: 0.4247, Test Loss: 0.2543, Test Accuracy:
0.9130, Learning Rate: 0.000340
Epoch 2/5, Train Loss: 0.2354, Test Loss: 0.2269, Test Accuracy:
0.9175, Learning Rate: 0.000238
Epoch 3/5, Train Loss: 0.1923, Test Loss: 0.2099, Test Accuracy:
0.9244, Learning Rate: 0.000167
Epoch 4/5, Train Loss: 0.1641, Test Loss: 0.2022, Test Accuracy:
0.9280, Learning Rate: 0.000117
Epoch 5/5, Train Loss: 0.1424, Test Loss: 0.1967, Test Accuracy:
0.9303, Learning Rate: 0.000082
Epoch 1/10, Train Loss: 0.4093, Test Loss: 0.2625, Test Accuracy:
0.9096, Learning Rate: 0.000409
Epoch 2/10, Train Loss: 0.2330, Test Loss: 0.2235, Test Accuracy:
0.9203, Learning Rate: 0.000286
Epoch 3/10, Train Loss: 0.1902, Test Loss: 0.2092, Test Accuracy:
0.9233, Learning Rate: 0.000200
Epoch 4/10, Train Loss: 0.1621, Test Loss: 0.1994, Test Accuracy:
0.9276, Learning Rate: 0.000140
Epoch 5/10, Train Loss: 0.1392, Test Loss: 0.1992, Test Accuracy:
0.9302, Learning Rate: 0.000098
Epoch 6/10, Train Loss: 0.1247, Test Loss: 0.1943, Test Accuracy:
0.9311, Learning Rate: 0.000069
Epoch 7/10, Train Loss: 0.1120, Test Loss: 0.1935, Test Accuracy:
0.9327, Learning Rate: 0.000048
Epoch 8/10, Train Loss: 0.1032, Test Loss: 0.1907, Test Accuracy:
0.9333, Learning Rate: 0.000034
Epoch 9/10, Train Loss: 0.0970, Test Loss: 0.1932, Test Accuracy:
0.9313, Learning Rate: 0.000024
```

```
Epoch 10/10, Train Loss: 0.0931, Test Loss: 0.1931, Test Accuracy:
0.9320, Learning Rate: 0.000016
Epoch 1/10, Train Loss: 0.4394, Test Loss: 0.3000, Test Accuracy:
0.8935, Learning Rate: 0.000315
Epoch 2/10, Train Loss: 0.2424, Test Loss: 0.2316, Test Accuracy:
0.9152, Learning Rate: 0.000221
Epoch 3/10, Train Loss: 0.2002, Test Loss: 0.2160, Test Accuracy:
0.9210, Learning Rate: 0.000154
Epoch 4/10, Train Loss: 0.1735, Test Loss: 0.2087, Test Accuracy:
0.9259, Learning Rate: 0.000108
Epoch 5/10, Train Loss: 0.1527, Test Loss: 0.2021, Test Accuracy:
0.9300, Learning Rate: 0.000076
Epoch 6/10, Train Loss: 0.1376, Test Loss: 0.2026, Test Accuracy:
0.9271, Learning Rate: 0.000053
Epoch 7/10, Train Loss: 0.1254, Test Loss: 0.2023, Test Accuracy:
0.9284, Learning Rate: 0.000037
Epoch 8/10, Train Loss: 0.1196, Test Loss: 0.2011, Test Accuracy:
0.9296, Learning Rate: 0.000026
Epoch 9/10, Train Loss: 0.1143, Test Loss: 0.2010, Test Accuracy:
0.9291, Learning Rate: 0.000018
Epoch 10/10, Train Loss: 0.1102, Test Loss: 0.1991, Test Accuracy:
0.9302, Learning Rate: 0.000013
Epoch 1/10, Train Loss: 0.3886, Test Loss: 0.2586, Test Accuracy:
0.9084, Learning Rate: 0.000530
Epoch 2/10, Train Loss: 0.2298, Test Loss: 0.2171, Test Accuracy:
0.9214, Learning Rate: 0.000371
Epoch 3/10, Train Loss: 0.1878, Test Loss: 0.2043, Test Accuracy:
0.9239, Learning Rate: 0.000259
Epoch 4/10, Train Loss: 0.1569, Test Loss: 0.2002, Test Accuracy:
0.9274, Learning Rate: 0.000182
Epoch 5/10, Train Loss: 0.1340, Test Loss: 0.1946, Test Accuracy:
0.9287, Learning Rate: 0.000127
Epoch 6/10, Train Loss: 0.1169, Test Loss: 0.1904, Test Accuracy:
0.9307, Learning Rate: 0.000089
Epoch 7/10, Train Loss: 0.1035, Test Loss: 0.1910, Test Accuracy:
0.9305, Learning Rate: 0.000062
Epoch 8/10, Train Loss: 0.0946, Test Loss: 0.1900, Test Accuracy:
0.9323, Learning Rate: 0.000044
Epoch 9/10, Train Loss: 0.0885, Test Loss: 0.1915, Test Accuracy:
0.9314, Learning Rate: 0.000031
Epoch 10/10, Train Loss: 0.0833, Test Loss: 0.1933, Test Accuracy:
0.9326, Learning Rate: 0.000021
Epoch 1/10, Train Loss: 0.3828, Test Loss: 0.2593, Test Accuracy:
0.9088, Learning Rate: 0.000609
Epoch 2/10, Train Loss: 0.2283, Test Loss: 0.2229, Test Accuracy:
0.9230, Learning Rate: 0.000426
Epoch 3/10, Train Loss: 0.1860, Test Loss: 0.2044, Test Accuracy:
0.9258, Learning Rate: 0.000298
Epoch 4/10, Train Loss: 0.1548, Test Loss: 0.1935, Test Accuracy:
```

```
0.9288, Learning Rate: 0.000209
Epoch 5/10, Train Loss: 0.1328, Test Loss: 0.1884, Test Accuracy:
0.9327, Learning Rate: 0.000146
Epoch 6/10, Train Loss: 0.1139, Test Loss: 0.1948, Test Accuracy:
0.9322, Learning Rate: 0.000102
Epoch 7/10, Train Loss: 0.1010, Test Loss: 0.1948, Test Accuracy:
0.9332, Learning Rate: 0.000072
Epoch 8/10, Train Loss: 0.0912, Test Loss: 0.1917, Test Accuracy:
0.9333, Learning Rate: 0.000050
Epoch 9/10, Train Loss: 0.0843, Test Loss: 0.1920, Test Accuracy:
0.9323, Learning Rate: 0.000035
Epoch 10/10, Train Loss: 0.0800, Test Loss: 0.1940, Test Accuracy:
0.9343, Learning Rate: 0.000025
Epoch 1/10, Train Loss: 0.3876, Test Loss: 0.2699, Test Accuracy:
0.8999, Learning Rate: 0.000530
Epoch 2/10, Train Loss: 0.2299, Test Loss: 0.2222, Test Accuracy:
0.9175, Learning Rate: 0.000371
Epoch 3/10, Train Loss: 0.1881, Test Loss: 0.2127, Test Accuracy:
0.9214, Learning Rate: 0.000259
Epoch 4/10, Train Loss: 0.1588, Test Loss: 0.1919, Test Accuracy:
0.9293, Learning Rate: 0.000182
Epoch 5/10, Train Loss: 0.1368, Test Loss: 0.1927, Test Accuracy:
0.9282, Learning Rate: 0.000127
Epoch 6/10, Train Loss: 0.1200, Test Loss: 0.1866, Test Accuracy:
0.9308, Learning Rate: 0.000089
Epoch 7/10, Train Loss: 0.1065, Test Loss: 0.1841, Test Accuracy:
0.9335, Learning Rate: 0.000062
Epoch 8/10, Train Loss: 0.0974, Test Loss: 0.1844, Test Accuracy:
0.9331, Learning Rate: 0.000044
Epoch 9/10, Train Loss: 0.0908, Test Loss: 0.1832, Test Accuracy:
0.9360, Learning Rate: 0.000031
Epoch 10/10, Train Loss: 0.0869, Test Loss: 0.1846, Test Accuracy:
0.9338, Learning Rate: 0.000021
Epoch 1/10, Train Loss: 0.3612, Test Loss: 0.2536, Test Accuracy:
0.9076, Learning Rate: 0.000832
Epoch 2/10, Train Loss: 0.2182, Test Loss: 0.2128, Test Accuracy:
0.9224, Learning Rate: 0.000582
Epoch 3/10, Train Loss: 0.1713, Test Loss: 0.1956, Test Accuracy:
0.9274, Learning Rate: 0.000408
Epoch 4/10, Train Loss: 0.1373, Test Loss: 0.1939, Test Accuracy:
0.9295, Learning Rate: 0.000285
Epoch 5/10, Train Loss: 0.1102, Test Loss: 0.1887, Test Accuracy:
0.9346, Learning Rate: 0.000200
Epoch 6/10, Train Loss: 0.0880, Test Loss: 0.1836, Test Accuracy:
0.9368, Learning Rate: 0.000140
Epoch 7/10, Train Loss: 0.0715, Test Loss: 0.1895, Test Accuracy:
0.9367, Learning Rate: 0.000098
Epoch 8/10, Train Loss: 0.0598, Test Loss: 0.1940, Test Accuracy:
0.9353, Learning Rate: 0.000068
```

```
Epoch 9/10, Train Loss: 0.0522, Test Loss: 0.1951, Test Accuracy:
0.9359, Learning Rate: 0.000048
Epoch 10/10, Train Loss: 0.0468, Test Loss: 0.2010, Test Accuracy:
0.9353, Learning Rate: 0.000034
Epoch 1/10, Train Loss: 0.3828, Test Loss: 0.2659, Test Accuracy:
0.9034, Learning Rate: 0.000609
Epoch 2/10, Train Loss: 0.2276, Test Loss: 0.2323, Test Accuracy:
0.9131, Learning Rate: 0.000426
Epoch 3/10, Train Loss: 0.1874, Test Loss: 0.2249, Test Accuracy:
0.9184, Learning Rate: 0.000298
Epoch 4/10, Train Loss: 0.1555, Test Loss: 0.2054, Test Accuracy:
0.9276, Learning Rate: 0.000209
Epoch 5/10, Train Loss: 0.1325, Test Loss: 0.1921, Test Accuracy:
0.9295, Learning Rate: 0.000146
Epoch 6/10, Train Loss: 0.1160, Test Loss: 0.1975, Test Accuracy:
0.9278, Learning Rate: 0.000102
Epoch 7/10, Train Loss: 0.1023, Test Loss: 0.1926, Test Accuracy:
0.9316, Learning Rate: 0.000072
Epoch 8/10, Train Loss: 0.0928, Test Loss: 0.1942, Test Accuracy:
0.9313, Learning Rate: 0.000050
Epoch 9/10, Train Loss: 0.0860, Test Loss: 0.1927, Test Accuracy:
0.9321, Learning Rate: 0.000035
Epoch 10/10, Train Loss: 0.0823, Test Loss: 0.1950, Test Accuracy:
0.9315, Learning Rate: 0.000025
Epoch 1/10, Train Loss: 0.3579, Test Loss: 0.2501, Test Accuracy:
0.9094, Learning Rate: 0.000729
Epoch 2/10, Train Loss: 0.2102, Test Loss: 0.2046, Test Accuracy:
0.9249, Learning Rate: 0.000510
Epoch 3/10, Train Loss: 0.1626, Test Loss: 0.1971, Test Accuracy:
0.9268, Learning Rate: 0.000357
Epoch 4/10, Train Loss: 0.1240, Test Loss: 0.1884, Test Accuracy:
0.9302, Learning Rate: 0.000250
Epoch 5/10, Train Loss: 0.0946, Test Loss: 0.1796, Test Accuracy:
0.9374, Learning Rate: 0.000175
Epoch 6/10, Train Loss: 0.0709, Test Loss: 0.1817, Test Accuracy:
0.9377, Learning Rate: 0.000122
Epoch 7/10, Train Loss: 0.0547, Test Loss: 0.1882, Test Accuracy:
0.9360, Learning Rate: 0.000086
Epoch 8/10, Train Loss: 0.0432, Test Loss: 0.1880, Test Accuracy:
0.9375, Learning Rate: 0.000060
Epoch 9/10, Train Loss: 0.0353, Test Loss: 0.1936, Test Accuracy:
0.9373, Learning Rate: 0.000042
Epoch 10/10, Train Loss: 0.0302, Test Loss: 0.1954, Test Accuracy:
0.9387, Learning Rate: 0.000029
Epoch 1/10, Train Loss: 0.3534, Test Loss: 0.2486, Test Accuracy:
0.9083, Learning Rate: 0.001017
Epoch 2/10, Train Loss: 0.2162, Test Loss: 0.2021, Test Accuracy:
0.9259, Learning Rate: 0.000712
Epoch 3/10, Train Loss: 0.1705, Test Loss: 0.1906, Test Accuracy:
```

```
0.9284, Learning Rate: 0.000499
Epoch 4/10, Train Loss: 0.1362, Test Loss: 0.2014, Test Accuracy:
0.9273, Learning Rate: 0.000349
Epoch 5/10, Train Loss: 0.1070, Test Loss: 0.1854, Test Accuracy:
0.9332, Learning Rate: 0.000244
Epoch 6/10, Train Loss: 0.0863, Test Loss: 0.1843, Test Accuracy:
0.9340, Learning Rate: 0.000171
Epoch 7/10, Train Loss: 0.0694, Test Loss: 0.1906, Test Accuracy:
0.9339, Learning Rate: 0.000120
Epoch 8/10, Train Loss: 0.0571, Test Loss: 0.1906, Test Accuracy:
0.9362, Learning Rate: 0.000084
Epoch 9/10, Train Loss: 0.0494, Test Loss: 0.1922, Test Accuracy:
0.9374, Learning Rate: 0.000059
Epoch 10/10, Train Loss: 0.0434, Test Loss: 0.1964, Test Accuracy:
0.9366, Learning Rate: 0.000041
Epoch 1/10, Train Loss: 0.3690, Test Loss: 0.2424, Test Accuracy:
0.9100, Learning Rate: 0.000522
Epoch 2/10, Train Loss: 0.2126, Test Loss: 0.2046, Test Accuracy:
0.9244, Learning Rate: 0.000365
Epoch 3/10, Train Loss: 0.1621, Test Loss: 0.1921, Test Accuracy:
0.9286, Learning Rate: 0.000256
Epoch 4/10, Train Loss: 0.1242, Test Loss: 0.1948, Test Accuracy:
0.9301, Learning Rate: 0.000179
Epoch 5/10, Train Loss: 0.0948, Test Loss: 0.1951, Test Accuracy:
0.9311, Learning Rate: 0.000125
Epoch 6/10, Train Loss: 0.0729, Test Loss: 0.1864, Test Accuracy:
0.9347, Learning Rate: 0.000088
Epoch 7/10, Train Loss: 0.0563, Test Loss: 0.1902, Test Accuracy:
0.9348, Learning Rate: 0.000061
Epoch 8/10, Train Loss: 0.0457, Test Loss: 0.1938, Test Accuracy:
0.9366, Learning Rate: 0.000043
Epoch 9/10, Train Loss: 0.0382, Test Loss: 0.1945, Test Accuracy:
0.9358, Learning Rate: 0.000030
Epoch 10/10, Train Loss: 0.0330, Test Loss: 0.1991, Test Accuracy:
0.9351, Learning Rate: 0.000021
Epoch 1/10, Train Loss: 0.3630, Test Loss: 0.2505, Test Accuracy:
0.9090, Learning Rate: 0.000646
Epoch 2/10, Train Loss: 0.2134, Test Loss: 0.2017, Test Accuracy:
0.9271, Learning Rate: 0.000452
Epoch 3/10, Train Loss: 0.1645, Test Loss: 0.1880, Test Accuracy:
0.9316, Learning Rate: 0.000316
Epoch 4/10, Train Loss: 0.1254, Test Loss: 0.1841, Test Accuracy:
0.9338, Learning Rate: 0.000221
Epoch 5/10, Train Loss: 0.0954, Test Loss: 0.1933, Test Accuracy:
0.9327, Learning Rate: 0.000155
Epoch 6/10, Train Loss: 0.0729, Test Loss: 0.1859, Test Accuracy:
0.9371, Learning Rate: 0.000108
Epoch 7/10, Train Loss: 0.0559, Test Loss: 0.1869, Test Accuracy:
0.9382, Learning Rate: 0.000076
```

```
Epoch 8/10, Train Loss: 0.0454, Test Loss: 0.1892, Test Accuracy:
0.9361, Learning Rate: 0.000053
Epoch 9/10, Train Loss: 0.0371, Test Loss: 0.1952, Test Accuracy:
0.9378, Learning Rate: 0.000037
Epoch 10/10, Train Loss: 0.0326, Test Loss: 0.1950, Test Accuracy:
0.9374, Learning Rate: 0.000026
Epoch 1/10, Train Loss: 0.3518, Test Loss: 0.2460, Test Accuracy:
0.9113, Learning Rate: 0.000822
Epoch 2/10, Train Loss: 0.2105, Test Loss: 0.2065, Test Accuracy:
0.9256, Learning Rate: 0.000576
Epoch 3/10, Train Loss: 0.1615, Test Loss: 0.2111, Test Accuracy:
0.9216, Learning Rate: 0.000403
Epoch 4/10, Train Loss: 0.1232, Test Loss: 0.1953, Test Accuracy:
0.9293, Learning Rate: 0.000282
Epoch 5/10, Train Loss: 0.0915, Test Loss: 0.1879, Test Accuracy:
0.9348, Learning Rate: 0.000197
Epoch 6/10, Train Loss: 0.0683, Test Loss: 0.1903, Test Accuracy:
0.9370, Learning Rate: 0.000138
Epoch 7/10, Train Loss: 0.0521, Test Loss: 0.1999, Test Accuracy:
0.9353, Learning Rate: 0.000097
Epoch 8/10, Train Loss: 0.0410, Test Loss: 0.1975, Test Accuracy:
0.9379, Learning Rate: 0.000068
Epoch 9/10, Train Loss: 0.0330, Test Loss: 0.2033, Test Accuracy:
0.9384, Learning Rate: 0.000047
Epoch 10/10, Train Loss: 0.0280, Test Loss: 0.2035, Test Accuracy:
0.9378, Learning Rate: 0.000033
```

Step 5: Conclusion

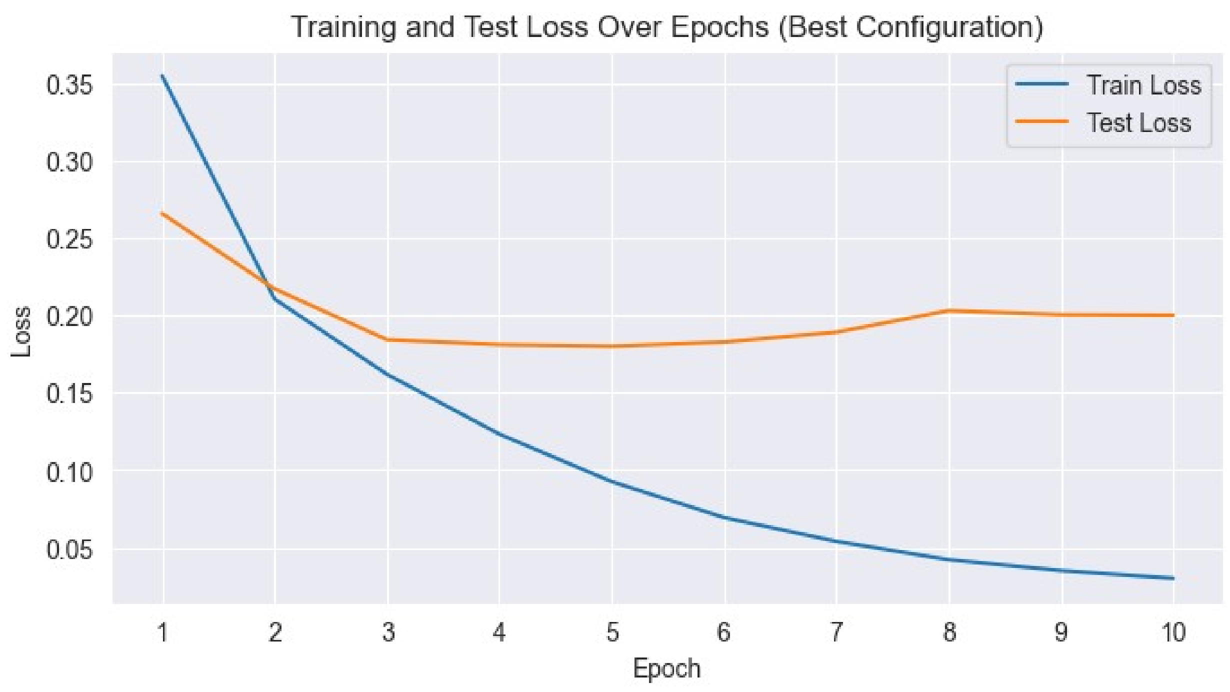
In this notebook I summarized the problem statement (to train a CNN to identify which type of item is in an image from the Fashion MNIST dataset), performed an EDA on the dataset, developed a baseline model, then improved on that model using hyperparameter tuning.

My baseline model achieved an accuracy of 93.27% on the test data. Hyperparameter tuning allowed me to marginally improve this to 93.63%.

For hyperparameter tuning I them employed a plugin called FLAML (Fast Library for Automated Machine Learning & Tuning), which allowed me to set ranges for various parameters as well as a number of configurations to run. I chose to run 20 total samples, which took 30min 53sec to run. The best model achieved an accuracy of 93.63%, which is only marginally better than the accuracy of the baseline model.

```python
# Train with the best configuration
best_config = analysis.best_config
print("Best configuration:", best_config)
best_result = train_fashion_mnist(best_config)
print("Best accuracy:", best_result['accuracy'])
```

```
Best configuration: {'conv1_out': 64, 'conv2_out': 128, 'fc1_out': 64,
'batch_size': 64, 'lr': np.float64(0.0010407752362789212), 'epochs':
10}
Epoch 1/10, Train Loss: 0.3604, Test Loss: 0.2451, Test Accuracy:
0.9097, Learning Rate: 0.000729
Epoch 2/10, Train Loss: 0.2141, Test Loss: 0.2094, Test Accuracy:
0.9255, Learning Rate: 0.000510
Epoch 3/10, Train Loss: 0.1666, Test Loss: 0.1954, Test Accuracy:
0.9274, Learning Rate: 0.000357
Epoch 4/10, Train Loss: 0.1281, Test Loss: 0.1814, Test Accuracy:
0.9335, Learning Rate: 0.000250
Epoch 5/10, Train Loss: 0.0972, Test Loss: 0.1970, Test Accuracy:
0.9296, Learning Rate: 0.000175
Epoch 6/10, Train Loss: 0.0732, Test Loss: 0.1842, Test Accuracy:
0.9362, Learning Rate: 0.000122
Epoch 7/10, Train Loss: 0.0558, Test Loss: 0.1861, Test Accuracy:
0.9355, Learning Rate: 0.000086
Epoch 8/10, Train Loss: 0.0456, Test Loss: 0.1934, Test Accuracy:
0.9347, Learning Rate: 0.000060
Epoch 9/10, Train Loss: 0.0376, Test Loss: 0.1931, Test Accuracy:
0.9373, Learning Rate: 0.000042
Epoch 10/10, Train Loss: 0.0319, Test Loss: 0.1975, Test Accuracy:
0.9363, Learning Rate: 0.000029
Best accuracy: 0.9363

# Plot the loss curves
plt.figure(figsize=(8, 4))
plt.plot(best_result['train_losses'], label='Train Loss')
plt.plot(best_result['test_losses'], label='Test Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training and Test Loss Over Epochs (Best Configuration)')
plt.xticks(range(len(best_result['train_losses'])), range(1,
len(best_result['train_losses']) + 1))  # Adjust x-axis labels
plt.legend()
plt.show()

# Save the trained model
model = best_result['model']
torch.save(model.state_dict(), 'best_model.pth')
print("Model saved to 'best_model.pth'")
```



```
Model saved to 'best_model.pth'
```

The graph above shows the loss of the best performing model for each training epoch. While the loss of the model on the training data continued to improve all the way up to epoch 10, the loss on the test data was optimal around epoch 3 and then stayed relatively steady (even increasing a bit) in later epochs. This suggests that epochs 4-10 may have been increasing the complexity of the model while overfitting the training data and not enhancing the performance on the test data.

Step 5.1: What's Next?

This model may be further improved with the following ideas:

1. Try a different type of model such as Residual Networks (ResNet), Densely Connected Convolutional Networks (DenseNet), or Capsule Networks (CapsNet)
2. Start with a pretrained model such as VGG16, MobileNet, or EfficientNet, which are pretrained on larger datasets and may only require minor tweaks to achieve good performance on this dataset.
3. Use a GAN (Generative Adversarial Network) to generate new images to augment the dataset, which could help regularize the model and improve generalization.
4. Run more than 20 samples. The available configuration options I provided include 243 options + a continuous variable (learning rate), which would yield 2430 options if I replaced the continuous options with 10 discrete options. I only ran 20 samples so I could complete the training in a reasonable timeframe. With infinite time or a higher powered machine, I could run more samples and potentially find a better model.