

Note: This notebook focuses exclusively on the implementation of **Transfer Learning** using ResNet-18. Detailed explanations of fundamental concepts (CNN architecture, data preprocessing, and evaluation metrics) have been omitted as they are thoroughly documented in the previous assignment (**Homework #4**). All markdown commentary in this notebook discusses only the changes and adaptations made to the original workflow to support transfer learning.

```
In [7]: # Import Torch
import torch
from torch import nn

# Import Torchvision
import torchvision
from torchvision import datasets
from torchvision import transforms
from torchvision.transforms import ToTensor
from torch.utils.data import DataLoader
from torchvision.models import resnet18, ResNet18_Weights

# Other
import matplotlib.pyplot as plt
from tqdm.auto import tqdm
from pathlib import Path
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMa
```

```
c:\Data\HIT\deep lerning\DeepLearning-HIT-Course\venv\Lib\site-packages\tqdm\aut
o.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets.
See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

```
In [8]: # HYPERPARAMETERS
EPOCHS = 7
BATCH_SIZE = 32
LEARNING_RATE = 0.0001 # Lower lr for transfer Learning
RANDOM_SEED = 242

CLASS_NAMES = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```

```
In [9]: # Set device agnostic code
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using {device} device")
```

Using cuda device

Data Preparation & Adaptation

To adapt the FashionMNIST dataset for transfer learning with ResNet-18, I made two significant changes to the preprocessing pipeline:

- **Channel Expansion:** Since ResNet models are pre-trained on RGB images and expect 3 input channels, I modified the transform to duplicate the single grayscale

channel three times. This allows the model to process the input natively without modifying its internal architecture.

- **Resizing Strategy:** I adjusted the image resolution to ensure effective feature extraction.
 - *Initial Approach:* I started with the original **28x28** resolution, but the model struggled because the spatial dimensions were pooled down to nothing before reaching the final layers.
 - *Adjustment:* I considered the standard ResNet input of **224x224**, but found it computationally expensive and unnecessary for this specific task.
 - *Final Decision:* I settled on resizing images to **64x64**. This "sweet spot" provides enough spatial resolution for the deep network to retain useful features while keeping training times short and efficient.

```
In [10]: # GET TRAIN DATASET
train_data = datasets.FashionMNIST(
    root="../data",
    train=True,
    download=True,
    transform=ToTensor()
)

# CALCULATE MEAN & STD
imgs = train_data.data.float() / 255.0 # Scale 0-255 integers to 0-1 floats (th
mean = imgs.mean().item()
std = imgs.std().item()

print(f"Calculated Mean: {mean:.4f}")
print(f"Calculated Std: {std:.4f}")

# NORMALIZE TRAIN DATA
normalization = transforms.Compose([
    transforms.Grayscale(num_output_channels=3), # <---- added
    transforms.Resize((64, 64)), # <---- added
    ToTensor(),
    transforms.Normalize((mean,), (std,))
])

train_data.transform = normalization
```

Calculated Mean: 0.2860

Calculated Std: 0.3530

```
In [11]: # GET TEST DATASET
test_data = datasets.FashionMNIST(
    root="../data",
    train=False,
    download=True,
    transform=normalization,
)

# CREATE DATALOADERS
train_dataloader = DataLoader(
    dataset=train_data,
    batch_size=BATCH_SIZE,
    shuffle=True)
```

```
test_dataloader = DataLoader(
    dataset=test_data,
    batch_size=BATCH_SIZE,
    shuffle=False)
```

Model Architecture: Fine-Tuning ResNet-18

I implemented a Fine-Tuning strategy using the **ResNet-18** architecture to adapt the model specifically for FashionMNIST.

- **Pre-trained Backbone:** I initialized the model with ImageNet weights (`ResNet18_Weights.DEFAULT`) and initially froze all parameters to preserve the robust lower-level feature extractors (edges, textures).
- **Fine-Tuning Strategy:** I explicitly **unfroze** `layer4` (the final convolutional block). This allows the model to retrain its highest-level filters to recognize clothing-specific patterns rather than generic ImageNet objects.
- **Classification Head:** I replaced the final fully connected layer with a new `nn.Linear` layer to map the features to the **10 FashionMNIST classes**.

Note: I will discuss the specific reasoning behind unfreezing `Layer4` and its impact on performance in detail in the Report Document.

```
In [12]: # Load Pretrained ResNet50
fashionModel = resnet18(weights=ResNet18_Weights.DEFAULT)

# Freeze all layers
for param in fashionModel.parameters():
    param.requires_grad = False

# Unfreeze Layer 4 (Last Convolutional Block)
for param in fashionModel.layer4.parameters():
    param.requires_grad = True

# Get the number of inputs going into the final layer ( 2048 for ResNet50)
in_features = fashionModel.fc.in_features

# Replace the last FC layer with a new one for FashionMNIST 10 classes
fashionModel.fc = nn.Linear(in_features, 10)

# Move to Device
fashionModel = fashionModel.to(device)
```

Training Configuration

For the training phase, I updated the optimization strategy to better suit the transfer learning process:

- **Optimizer:** I switched to the **Adam** optimizer, which adapts learning rates individually for each parameter. I implemented differential learning rates:
 - **Classifier (fc):** Trained at the base learning rate (`LEARNING_RATE`) to quickly learn the new class associations.

- **Fine-Tuning (layer4):** Trained at a reduced rate ($\text{LEARNING_RATE} / 5$) to gently adjust the pre-trained high-level features without distorting them.
- **Scheduler:** I retained the **StepLR** scheduler to decay the learning rate by a factor of 0.1 every 4 epochs, ensuring the model settles into a precise minimum as training progresses.

```
In [13]: # LOSS FUNCTION
loss_fn = nn.CrossEntropyLoss()

# OPTIMIZER
optimizer = torch.optim.Adam([
    {'params': fashionModel.layer4.parameters(), 'lr': LEARNING_RATE / 5}, # Tra
    {'params': fashionModel.fc.parameters(), 'lr': LEARNING_RATE}          # Tr
], weight_decay=1e-4)

# LEARNING RATE SCHEDULER
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=4, gamma=0.1)

# ACCURACY FUNCTION
def accuracy_fn(y_true, y_pred):
    correct = torch.eq(y_true, y_pred).sum().item()
    acc = correct / len(y_pred) * 100
    return acc
```

```
In [14]: # TRAINING LOOP FUNCTION
def train_step(model: torch.nn.Module,
               data_loader: torch.utils.data.DataLoader,
               loss_fn: torch.nn.Module,
               optimizer: torch.optim.Optimizer,
               accuracy_fn,
               device: torch.device = device):

    # Put model in training mode
    model.train()

    # Initialize train loss and train acc
    train_loss, train_acc = 0, 0

    for batch, (X, y) in enumerate(tqdm(data_loader)):
        # Moving batch images and label to device
        X, y = X.to(device), y.to(device)

        # 1. Forward pass (get logits)
        y_pred = model(X)

        # 2. Calculate loss and accuracy
        loss = loss_fn(y_pred, y)
        train_loss += loss.item()
        train_acc += accuracy_fn(y_true=y, y_pred=y_pred.argmax(dim=1)) # Conve

        # 3. optimizer zero
        optimizer.zero_grad()

        # 4. Loss backward
        loss.backward()

        # 5. Optimizer step
        optimizer.step()
```

```

# Devide train_loass and train_acc in num batch to get average Loss, acc per
train_loss /= len(data_loader)
train_acc /= len(data_loader)
print(f"| Train Loss: {train_loss:.5f} Train accuracy: {train_acc:.4f}% |")

return train_loss, train_acc

# TEST LOOP FUNCTION
def test_step(model: torch.nn.Module,
              data_loader: torch.utils.data.DataLoader,
              loss_fn: torch.nn.Module,
              accuracy_fn,
              device: torch.device = device):

    """Perform Test loop on dataloader"""

    model.eval()
    test_loss, test_acc = 0, 0

    with torch.inference_mode():
        for X, y in tqdm(data_loader):
            # Moving batch images and label to device
            X, y = X.to(device), y.to(device)

            # 1. Forward pass
            y_pred = model(X)

            # 2. Calculate Loss and accuracy
            test_loss += loss_fn(y_pred, y).item()
            test_acc += accuracy_fn(y_true=y, y_pred=y_pred.argmax(dim=1)) # in

        # Get average test Loss and avarge accuracy per batch
        test_loss /= len(data_loader) # Total Loss is not a regular int and thi
        test_acc /= len(data_loader)

    print(f"| Test loss: {test_loss:.5f} Test acc: {test_acc:.4f}% |\n")

    return test_loss, test_acc

```

```

In [15]: results = {
    "train_loss": [],
    "train_acc": [],
    "test_loss": [],
    "test_acc": []
}

for epoch in range(EPOCHS):
    print(f"Epoch: {epoch} -----")

    train_loss, train_acc = train_step(model=fashionModel,
                                       data_loader=train_dataloader,
                                       loss_fn=loss_fn,
                                       optimizer=optimizer,
                                       accuracy_fn=accuracy_fn,
                                       device=device)

    test_loss, test_acc = test_step(model=fashionModel,
                                    data_loader=test_dataloader,
                                    loss_fn=loss_fn,
                                    accuracy_fn=accuracy_fn,

```

```

        device=device)

    scheduler.step()
    print(f"Current Learning Rate: {optimizer.param_groups[0]['lr']}")

    results["train_loss"].append(train_loss)
    results["train_acc"].append(train_acc)
    results["test_loss"].append(test_loss)
    results["test_acc"].append(test_acc)

```

Epoch: 0 -----

100%|██████████| 1875/1875 [00:49<00:00, 38.05it/s]

| Train Loss: 0.48532 Train accuracy: 83.4350% |

100%|██████████| 313/313 [00:07<00:00, 43.91it/s]

| Test loss: 0.31739 Test acc: 88.8578% |

Current Learning Rate: 2e-05

Epoch: 1 -----

100%|██████████| 1875/1875 [00:45<00:00, 40.77it/s]

| Train Loss: 0.27069 Train accuracy: 90.2367% |

100%|██████████| 313/313 [00:07<00:00, 43.02it/s]

| Test loss: 0.28268 Test acc: 89.8263% |

Current Learning Rate: 2e-05

Epoch: 2 -----

100%|██████████| 1875/1875 [00:47<00:00, 39.58it/s]

| Train Loss: 0.20170 Train accuracy: 92.7350% |

100%|██████████| 313/313 [00:07<00:00, 42.44it/s]

| Test loss: 0.27841 Test acc: 90.3055% |

Current Learning Rate: 2e-05

Epoch: 3 -----

100%|██████████| 1875/1875 [00:55<00:00, 33.86it/s]

| Train Loss: 0.15122 Train accuracy: 94.6700% |

100%|██████████| 313/313 [00:16<00:00, 18.67it/s]

| Test loss: 0.28204 Test acc: 90.4852% |

Current Learning Rate: 2.0000000000000003e-06

Epoch: 4 -----

100%|██████████| 1875/1875 [01:19<00:00, 23.47it/s]

| Train Loss: 0.10142 Train accuracy: 96.6633% |

100%|██████████| 313/313 [00:19<00:00, 16.36it/s]

| Test loss: 0.27734 Test acc: 90.8347% |

Current Learning Rate: 2.0000000000000003e-06

Epoch: 5 -----

100%|██████████| 1875/1875 [01:20<00:00, 23.26it/s]

| Train Loss: 0.09147 Train accuracy: 96.8917% |

100%|██████████| 313/313 [00:18<00:00, 17.13it/s]

| Test loss: 0.28338 Test acc: 90.8846% |

Current Learning Rate: 2.0000000000000003e-06

Epoch: 6 -----

100%|██████████| 1875/1875 [01:22<00:00, 22.77it/s]

| Train Loss: 0.08504 Train accuracy: 97.2300% |

100%|██████████| 313/313 [00:18<00:00, 16.79it/s]

| Test loss: 0.28614 Test acc: 90.8446% |

Current Learning Rate: 2.0000000000000003e-06

Evaluation Strategy

The evaluation pipeline remains consistent with the previous assignment to ensure comparable metrics:

- **Performance Tracking:** I continued to monitor **Loss and Accuracy curves** to verify convergence and prevent overfitting.
- **Detailed Metrics:** I utilized the **Confusion Matrix** and **Classification Report** to analyze specific class-level performance.
- **Focused Visualization:** I visualized the learned filters and feature maps specifically for **layer 4**. Since this is the only convolutional block explicitly unfrozen and fine-tuned, analyzing its activations provides insight into how the pre-trained model adapted its high-level feature representations for the FashionMNIST dataset.

```
In [16]: def plot_loss_curves(results):
          """Plots training curves of a results dictionary."""

          # Get the loss values of the results dictionary (training and test)
          loss = results['train_loss']
          test_loss = results['test_loss']

          # Get the accuracy values of the results dictionary (training and test)
          accuracy = results['train_acc']
          test_accuracy = results['test_acc']

          # Setup the figure count
          epochs = range(len(results['train_loss']))

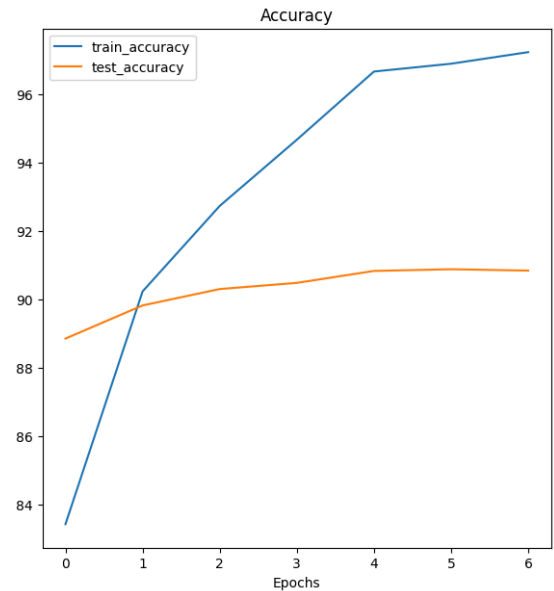
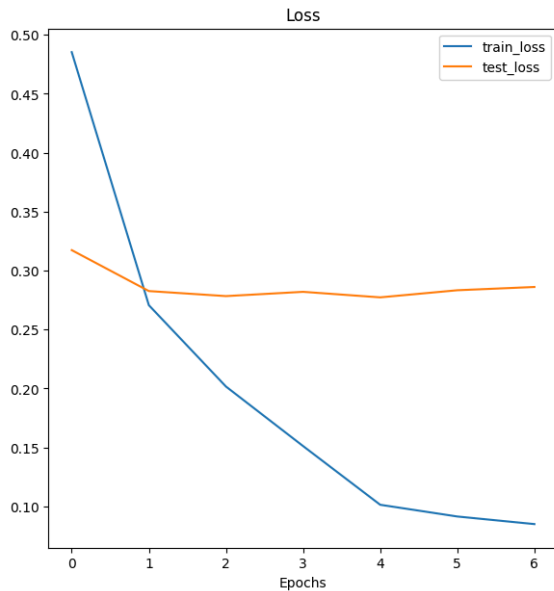
          # Setup a plot
          plt.figure(figsize=(15, 7))

          # Plot the Loss
          plt.subplot(1, 2, 1)
          plt.plot(epochs, loss, label='train_loss')
          plt.plot(epochs, test_loss, label='test_loss')
          plt.title('Loss')
          plt.xlabel('Epochs')
          plt.legend()

          # Plot the accuracy
          plt.subplot(1, 2, 2)
          plt.plot(epochs, accuracy, label='train_accuracy')
          plt.plot(epochs, test_accuracy, label='test_accuracy')
          plt.title('Accuracy')
          plt.xlabel('Epochs')
          plt.legend()

          plt.show()

          # Run the function
          plot_loss_curves(results)
```



```
In [17]: fashionModel.eval()
loss, acc = 0, 0

# Lists to store ALL predictions and ALL true labels
all_preds = []
all_labels = []

with torch.inference_mode():
    for X, y in tqdm(test_dataloader):
        X, y = X.to(device), y.to(device)

        # Forward pass
        y_logits = fashionModel(X)
        y_pred = y_logits.argmax(dim=1) # Convert logits to class labels

        # Calculate Loss & Accuracy (Running average)
        loss += loss_fn(y_logits, y).item()
        acc += accuracy_fn(y_true=y, y_pred=y_pred)

        # 2. Store predictions/labels for later (Move to CPU first!)
        all_preds.append(y_pred.cpu())
        all_labels.append(y.cpu())

# Calculate average Loss/acc
loss /= len(test_dataloader)
acc /= len(test_dataloader)

# Concatenate all batches into single tensors
all_preds_tensor = torch.cat(all_preds)
all_labels_tensor = torch.cat(all_labels)

# Calculate Metrics using Scikit-Learn with
print(f"\nModel Results:\nLoss: {loss:.4f} | Accuracy: {acc:.2f}%")

print("\nClassification Report (Precision, Recall, F1):")
print(classification_report(all_labels_tensor, all_preds_tensor))

# Plot Confusion Matrix
print("Confusion Matrix:")
cm = confusion_matrix(all_labels_tensor, all_preds_tensor)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=CLASS_NAMES)
```

```
# Plotting
fig, ax = plt.subplots(figsize=(10, 10))
disp.plot(cmap='Blues', ax=ax, values_format='d')
plt.title("Confusion Matrix")
plt.show()
```

100%|██████████| 313/313 [00:22<00:00, 14.06it/s]

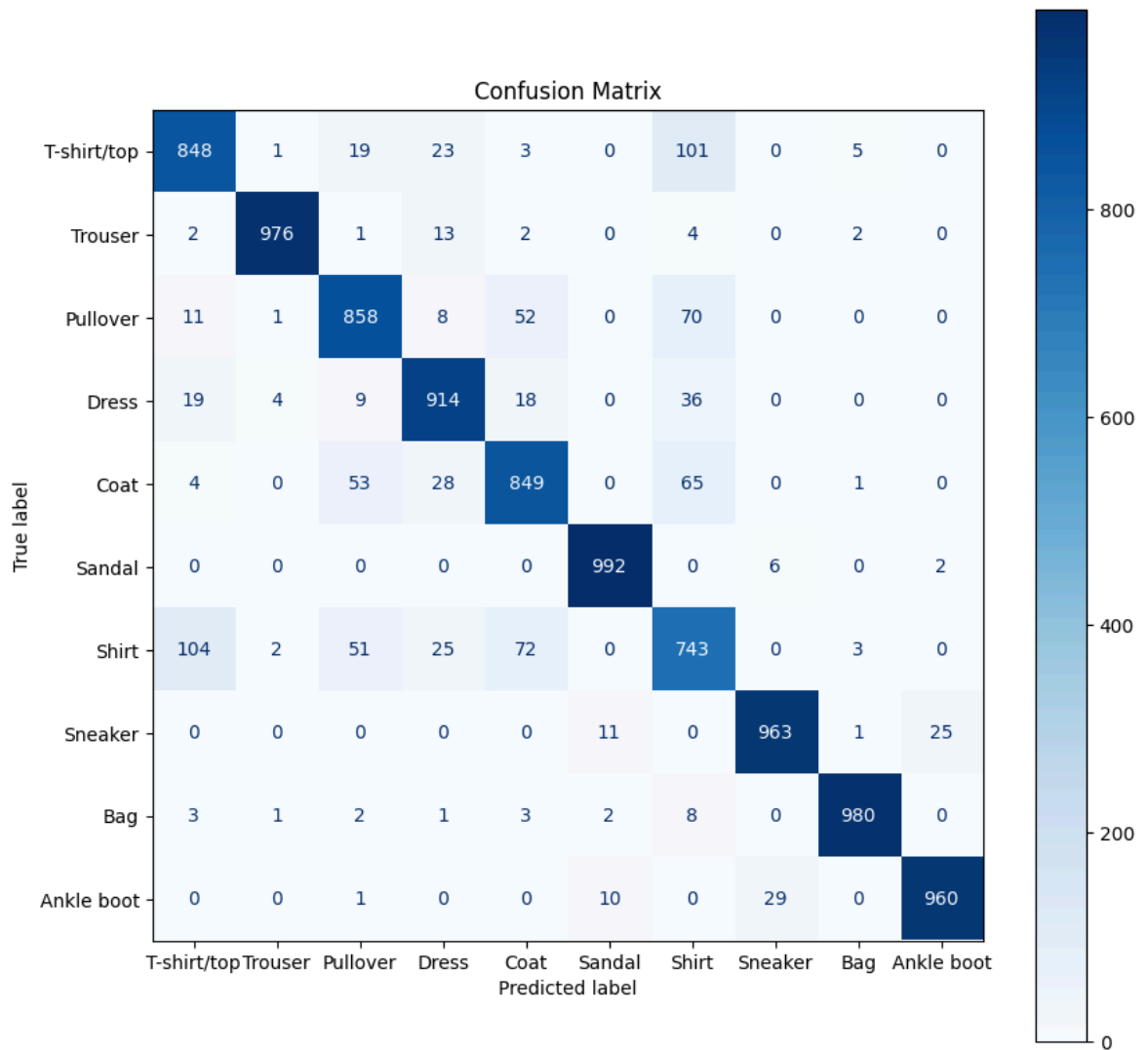
Model Results:

Loss: 0.2861 | Accuracy: 90.84%

Classification Report (Precision, Recall, F1):

	precision	recall	f1-score	support
0	0.86	0.85	0.85	1000
1	0.99	0.98	0.98	1000
2	0.86	0.86	0.86	1000
3	0.90	0.91	0.91	1000
4	0.85	0.85	0.85	1000
5	0.98	0.99	0.98	1000
6	0.72	0.74	0.73	1000
7	0.96	0.96	0.96	1000
8	0.99	0.98	0.98	1000
9	0.97	0.96	0.97	1000
accuracy			0.91	10000
macro avg	0.91	0.91	0.91	10000
weighted avg	0.91	0.91	0.91	10000

Confusion Matrix:



```
In [18]: # SAVE MODEL

MODEL_PATH = Path("./")
MODEL_NAME = "ResNet18_FashionMNIST.pth"
MODEL_SAVE_PATH = MODEL_PATH / MODEL_NAME

print(f"Saving model to: {MODEL_SAVE_PATH}")
torch.save(obj=fashionModel.state_dict(), f=MODEL_SAVE_PATH)
```

Saving model to: ResNet18_FashionMNIST.pth

```
In [19]: def visualize_resnet_filters(model, num_to_show=16):
# Access the specific layer weights
# ResNet structure: model -> Layer4 -> Block 0 -> Conv1
filters = model.layer4[0].conv1.weight.data.cpu().numpy()

# filters shape is [512, 256, 3, 3] (512 filters, depth of 256, 3x3 kernel)
# We slice to show only the first 'num_to_show' filters because showing 512
filters = filters[:num_to_show]

# 2. Setup Plot
n_columns = 8
n_rows = num_to_show // n_columns + (num_to_show % n_columns > 0)

fig, axes = plt.subplots(n_rows, n_columns, figsize=(15, 2 * n_rows))
fig.suptitle('Learned Filters: ResNet Layer 4 (Subset)', fontsize=16)
```

```

# Flatten axes array for easy iteration if multiple rows
axes = axes.flatten() if num_to_show > 1 else [axes]

for i in range(num_to_show):
    # We take the ith filter.
    # We take channel 0 to visualize it as a 2D image.
    f = filters[i, 0, :, :]

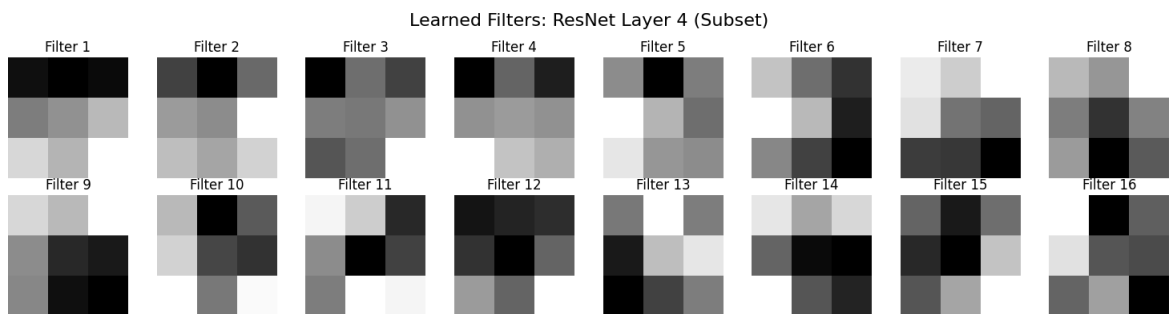
    # Plot
    ax = axes[i]
    ax.imshow(f, cmap='gray') # viridis or gray both work well here
    ax.axis('off')
    ax.set_title(f'Filter {i+1}')

# Turn off any empty subplots
for j in range(i + 1, len(axes)):
    axes[j].axis('off')

plt.tight_layout()
plt.show()

# Run the function
visualize_resnet_filters(fashionModel)

```



```

In [20]: def visualize_feature_maps(model, image, device):
    model.eval()

    # 1. Prepare image
    x = image.unsqueeze(0).to(device)

    # 2. Pass data through the model MANUALLY until we reach layer 4
    # We must run these first so layer 4 gets the correct processed info
    x = model.conv1(x)
    x = model.bn1(x)
    x = model.relu(x)
    x = model.maxpool(x)

    x = model.layer1(x)
    x = model.layer2(x)
    x = model.layer3(x)

    # 3. Pass through Layer 4 and Capture the result
    # This is the layer we modified and want to visualize
    out_layer4 = model.layer4(x)

    # 4. Prepare for plotting
    # Shape is [1, 512, 2, 2] (or [1, 512, 8, 8] if using 224x224 input)
    feature_maps = out_layer4.squeeze(0).detach().cpu()

    # LIMIT to first 16 maps (ResNet has 512, which is too many to plot)

```

```

num_maps_to_show = 16
feature_maps = feature_maps[:num_maps_to_show]

# 5. Plotting Code
n_columns = 8
n_rows = num_maps_to_show // n_columns + (num_maps_to_show % n_columns > 0)

fig, axes = plt.subplots(n_rows, n_columns, figsize=(15, 2 * n_rows))
fig.suptitle(f'Layer 4 Feature Maps (First {num_maps_to_show} of 512)', font

# Flatten axes for easy looping
axes = axes.flatten() if num_maps_to_show > 1 else [axes]

for i in range(num_maps_to_show):
    ax = axes[i]
    ax.imshow(feature_maps[i], cmap='gray')
    ax.axis('off')

plt.tight_layout()
plt.show()

# Run the function
sample_img, label = test_data[0]
visualize_feature_maps(fashionModel, sample_img, device)

```

Layer 4 Feature Maps (First 16 of 512)

