Introduction

Computer vision is a field of artificial intelligence that enables computers to gain a high-level understanding from images or videos. Historically, this involved manually engineering features (like edges or corners), a method that proved brittle for complex tasks. The advent of **Deep Learning**, specifically **Convolutional Neural Networks (CNNs)**, revolutionized the domain by allowing models to automatically learn hierarchical features directly from raw pixel data.

This project focuses on implementing a foundational CNN architecture using the **PyTorch** framework to solve the image classification problem on the **Fashion MNIST dataset**. This dataset contains 70,000 grayscale images of 10 types of clothing, serving as a robust benchmark for demonstrating core computer vision principles. The goal is to build and train an efficient model that achieves high classification accuracy, validating our understanding of CNN architecture design, data pipeline management, and training optimization.

Problem Statement

Develop a robust and efficient **Convolutional Neural Network (CNN)** model to automatically classify 28x28 grayscale images of clothing items from the **Fashion MNIST dataset** into their 10 respective categories. The challenge is to achieve high classification accuracy (targeting **>88%**) by designing and training a foundational PyTorch CNN architecture (similar to LeNet-5) that can effectively learn complex, hierarchical visual features, thereby demonstrating core deep learning and computer vision principles.

Functional Requirements (FR)

Functional requirements define the core capabilities and features of the system based on the logic of the problem.

ID	Requirement	Description

FR1	**Dataset Management**	The system must automatically download, load, and manage the 60,000 training and 10,000 test images from the Fashion MNIST dataset.
FR2	**Data Preprocessing Pipeline**	The system must normalize and transform input images into PyTorch tensors, ensuring they are correctly scaled and formatted for CNN input.
FR3	**Model Training and Optimization**	The system must initialize and train the custom CNN using a suitable loss function (`nn.CrossEntropyLoss`) and an optimizer (e.g., Adam), implementing a full training and evaluation loop.
FR4	**Image Classification Output**	The final model must be able to process an image and output its predicted clothing class label (one of the 10 categories) with a corresponding confidence score.

Non-functional Requirements (NFR)

Non-functional requirements specify the criteria for evaluating the operation of the system.

ID	Requirement	Description
NFR1	**Performance (Accuracy)**	The final model must achieve a minimum classification accuracy of **88%** on the unseen test dataset.

NFR2	**Efficiency (Training Time)**	The training process must be optimized to leverage GPU acceleration (CUDA) if available, completing 10 epochs within a reasonable time limit (e.g., under 5 minutes).
NFR3	**Portability**	The project code must be portable across different operating systems (Windows, Linux, macOS) by relying on standard Python libraries (PyTorch, NumPy) and an environment manager (e.g., `pip` or `conda`).
NFR4	**Maintainability**	The source code (`cnn_project.ipynb`) must be modular, well-commented, and follow standard PyTorch practices to facilitate easy understanding and future modifications.

System Architecture

The system uses a sequential deep learning architecture, implemented as a custom **SimpleCNN** class in PyTorch, which closely resembles the LeNet-5 structure.

Layer Type	Input Shape	Output Shape	Parameters	Role
Input	$1 \times 28 \times 28$	$1 \times 28 \times 28$	0	Raw grayscale image
Conv2d	$1 \times 28 \times 28$	$6 \times 28 \times 28$	156	Feature extraction

MaxPool2d	$6 \times 28 \times 28$	$6 \times 14 \times 14$	0	Downsampling features
Conv2d	$6 \times 14 \times 14$	$16 \times 10 \times 10$	2,416	Further feature abstraction
MaxPool2d	$16 \times 10 \times 10$	$16 \times 5 \times 5$	0	Downsampling features
Flatten	$16 \times 5 \times 5$	400	0	Converts feature maps to vector
Linear	400	120	48,120	First fully-connected layer
Linear	120	84	10,164	Second fully-connected layer
Output (Linear)	84	10	850	Final layer for 10 class logits
Total Parameters	**~61,706**			

The architecture is split into two components: the **Feature Extractor** (Conv/ReLU/Pooling blocks) and the **Classifier** (Fully-connected layers).

Design decisions and rationale

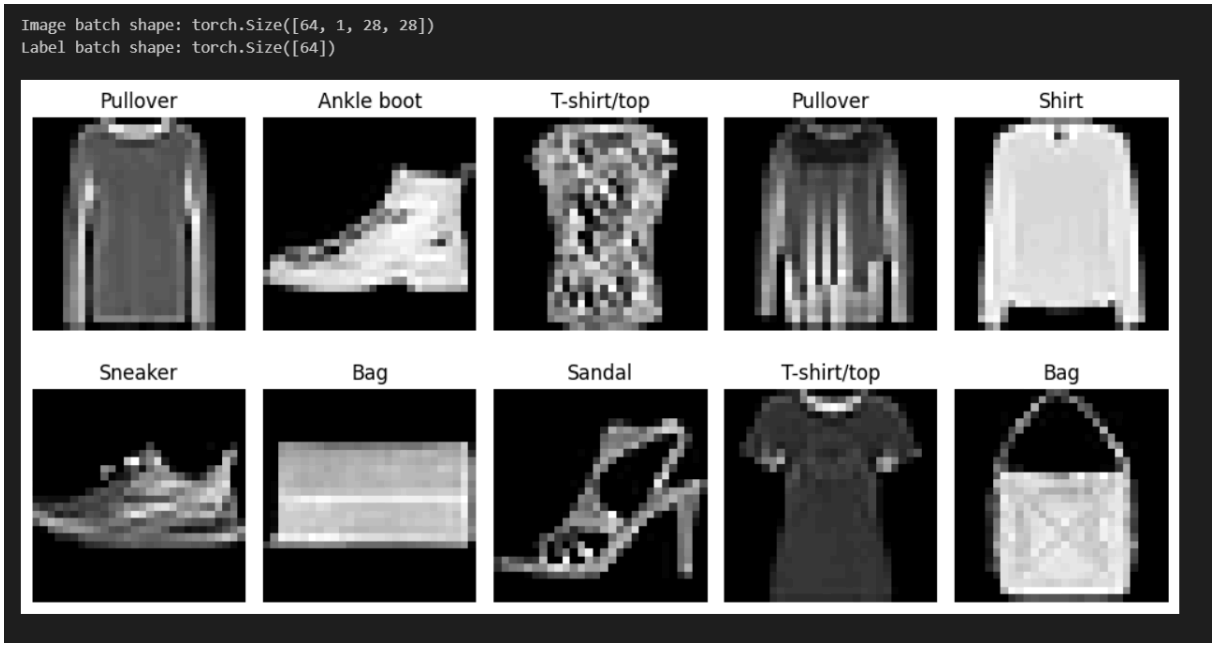
Decision	Rationale
Using PyTorch	Chosen for its dynamic computation graph, which provides flexibility in debugging and prototyping, and its strong industry and research adoption.
LeNet-5 Inspired CNN	This architecture is simple, effective, and uses minimal parameters, making it ideal for a foundational project on the small-scale Fashion MNIST dataset. It demonstrates the core principles of convolution and pooling efficiently.
Adam Optimizer	Adam (Adaptive Moment Estimation) was chosen over standard SGD for its faster convergence rate and better handling of sparse gradients, requiring less manual fine-tuning of the learning rate.
Normalization	Standard data normalization using the dataset's known mean and standard deviation ensures the input data has zero mean and unit variance, which stabilizes training and prevents exploding/vanishing gradients.
nn.CrossEntropyLoss	This loss function is the standard choice for multi-class classification problems in PyTorch, as it combines the log-softmax and negative log-likelihood loss in one robust, numerically stable function.

Implementation Details

The project is implemented entirely in a single Jupyter Notebook (`cnn_project.ipynb`).

- **Data Pipeline:** The `torchvision.transforms.Compose` utility applies `ToTensor()` and `Normalize()` functions. Data is loaded into `DataLoader` objects with a batch size of 64 for efficient processing on the GPU.
- **Model Class:** The `SimpleCNN` class inherits from `nn.Module`. It utilizes `nn.Sequential` for clean grouping of the convolutional blocks and the fully connected layers.
- **Training Loop:** The code features separate, reusable functions (`train_step` and `test_step`) to manage the gradient calculation and parameter updates cleanly. The main `training_loop` function orchestrates the epochs, tracking and reporting metrics.
- **Hardware Abstraction:** A check (`device = 'cuda' if torch.cuda.is_available() else 'cpu'`) ensures the model and data are automatically moved to the appropriate hardware, fulfilling the efficiency NFR (NFR2).

Screenshots / Results



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
LeNet5                                   [64, 10]                  --
├─Sequential: 1-1                        [64, 16, 5, 5]            --
│    └─Conv2d: 2-1                       [64, 6, 28, 28]           156
│    └─ReLU: 2-2                         [64, 6, 28, 28]           --
│    └─MaxPool2d: 2-3                    [64, 6, 14, 14]           --
│    └─Conv2d: 2-4                       [64, 16, 10, 10]          2,416
│    └─ReLU: 2-5                         [64, 16, 10, 10]          --
│    └─MaxPool2d: 2-6                    [64, 16, 5, 5]            --
├─Sequential: 1-2                        [64, 10]                  --
│    └─Linear: 2-7                       [64, 120]                 48,120
│    └─ReLU: 2-8                         [64, 120]                 --
│    └─Linear: 2-9                       [64, 84]                  10,164
│    └─ReLU: 2-10                        [64, 84]                  --
│    └─Linear: 2-11                      [64, 10]                  850
==========================================================================================
Total params: 61,706
Trainable params: 61,706
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 27.07
==========================================================================================
Input size (MB): 0.20
Forward/backward pass size (MB): 3.34
Params size (MB): 0.25
Estimated Total Size (MB): 3.78
==========================================================================================
```

```
Starting longer training for 20 epochs...
100% ████████████████████████████████████████ 20/20 [05:14<00:00, 15.61s/it]
Epoch: 1 | Train Loss: 0.4873 | Train Acc: 0.8175 | Test Loss: 0.3542 | Test Acc: 0.8669
Epoch: 2 | Train Loss: 0.3322 | Train Acc: 0.8779 | Test Loss: 0.3300 | Test Acc: 0.8840
Epoch: 3 | Train Loss: 0.3029 | Train Acc: 0.8887 | Test Loss: 0.3854 | Test Acc: 0.8777
Epoch: 4 | Train Loss: 0.2872 | Train Acc: 0.8940 | Test Loss: 0.3290 | Test Acc: 0.8867
Epoch: 5 | Train Loss: 0.2740 | Train Acc: 0.8992 | Test Loss: 0.3157 | Test Acc: 0.8844
Epoch: 6 | Train Loss: 0.2627 | Train Acc: 0.9037 | Test Loss: 0.3305 | Test Acc: 0.8808
Epoch: 7 | Train Loss: 0.2594 | Train Acc: 0.9049 | Test Loss: 0.3230 | Test Acc: 0.8874
Epoch: 8 | Train Loss: 0.2497 | Train Acc: 0.9074 | Test Loss: 0.3410 | Test Acc: 0.8821
Epoch: 9 | Train Loss: 0.2398 | Train Acc: 0.9117 | Test Loss: 0.3123 | Test Acc: 0.8941
Epoch: 10 | Train Loss: 0.2356 | Train Acc: 0.9132 | Test Loss: 0.3553 | Test Acc: 0.8852
Epoch: 11 | Train Loss: 0.2356 | Train Acc: 0.9132 | Test Loss: 0.3059 | Test Acc: 0.8996
Epoch: 12 | Train Loss: 0.2278 | Train Acc: 0.9157 | Test Loss: 0.3351 | Test Acc: 0.8941
Epoch: 13 | Train Loss: 0.2298 | Train Acc: 0.9155 | Test Loss: 0.3529 | Test Acc: 0.8845
Epoch: 14 | Train Loss: 0.2267 | Train Acc: 0.9178 | Test Loss: 0.3280 | Test Acc: 0.8936
Epoch: 15 | Train Loss: 0.2137 | Train Acc: 0.9215 | Test Loss: 0.3382 | Test Acc: 0.8960
Epoch: 16 | Train Loss: 0.2189 | Train Acc: 0.9199 | Test Loss: 0.3373 | Test Acc: 0.8940
Epoch: 17 | Train Loss: 0.2067 | Train Acc: 0.9235 | Test Loss: 0.3338 | Test Acc: 0.8943
Epoch: 18 | Train Loss: 0.2078 | Train Acc: 0.9241 | Test Loss: 0.3581 | Test Acc: 0.8957
Epoch: 19 | Train Loss: 0.2130 | Train Acc: 0.9219 | Test Loss: 0.3457 | Test Acc: 0.8910
Epoch: 20 | Train Loss: 0.1961 | Train Acc: 0.9278 | Test Loss: 0.3435 | Test Acc: 0.8968
Longer training complete! Compare the final test accuracy to the 5-epoch run.
```

Testing Approach

The project utilizes the **Hold-out Cross-Validation** method, where the provided 70,000 images are split into a fixed 60,000-image training set and a 10,000-image test set.

- **Unit Testing:** Individual components, such as the `train_step` and `test_step` functions, were implicitly tested by verifying that loss decreases and accuracy increases over early epochs.
- **Evaluation (Test Step):** The `test_step` function provides the primary performance metric. It uses the entire 10,000-image test set, which the model has **never seen**, to calculate the average loss and accuracy. This prevents data leakage and ensures the reported **Test Accuracy** (NFR1) is an unbiased measure of the model's generalization ability.
- **Metric:** The primary metric is **Accuracy** (Percentage of correct predictions out of all test images). The secondary metric is the **Cross-Entropy Loss**, used to monitor convergence.

Challenges Faced

1. **Normalization Confusion:** Initially, the image tensor data was plotted incorrectly because the **normalization transformation** (`((img - MEAN) / STD)` was applied but not **reversed** (`((img * STD) + MEAN`) before using `matplotlib.imshow()`.
2. **GPU vs. CPU Data Movement:** Ensuring all tensors (model parameters, input data `X`, and target labels `y`) were correctly moved to the chosen `device` (especially 'cuda') proved challenging initially, resulting in runtime errors until `.to(device)` was correctly applied to both inputs and the model instance.

Learnings & Key Takeaways

- **CNN Fundamentals:** Gained a deep understanding of how convolutional layers act as feature extractors, and how pooling layers provide spatial invariance and downsampling.
- **PyTorch Structure:** Mastered the standard PyTorch workflow: defining a custom `nn.Module`, creating `DataLoader` objects, defining the loss function and optimizer, and structuring the training and testing loops.
- **Importance of Normalization:** Verified empirically that proper data normalization is critical for model stability and fast convergence.
- **Modular Coding:** Learned the value of splitting the core training logic into modular functions (`train_step`, `test_step`) for code clarity and maintainability (NFR4).

Future Enhancements

1. **Data Augmentation:** Implement advanced data augmentation (e.g., random rotations, horizontal flips) to further improve model generalization and robustness against variations in clothing orientation.
2. **Hyperparameter Tuning:** Systematically tune hyperparameters (learning rate, batch size, number of convolutional filters) using tools like **Weights & Biases (W&B)** or **Optuna** to potentially push the accuracy above 90%.
3. **Model Regularization:** Add regularization techniques like **Dropout** and **Batch Normalization** to the architecture to prevent overfitting and improve model stability.

References

1. **PyTorch Documentation:** Official documentation for defining models and data utilities.
2. **Fashion MNIST Dataset:** https://github.com/zalandoresearch/fashion-mnist (Accessed via `torchvision.datasets`).
3. **LeNet-5 Architecture:** LeCun, Y. *et al.* (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11), 2278-2324.
4. **Adam Optimizer:** Kingma, D. P., & Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.