Coursework 4: Convolutional Neural Network

CNN Architectures with PyTorch

Course: CS3317: Artificial Intelligence
University: Shanghai Jiao Tong University

Overview

You will implement **CNN architectures** using PyTorch for FashionMNIST classification. Unlike CW2 and CW3 where you implemented everything from scratch, here you use PyTorch's `nn.Module` and **autograd** — you only need to define the architecture; PyTorch handles gradient computation and backpropagation automatically.

- **Dataset:** FashionMNIST (same dataset, but now loaded as $1 \times 28 \times 28$ images via PyTorch DataLoader).
- **Framework:** PyTorch.
- **Model:** Convolutional neural network with conv layers, pooling, and fully connected layers.

Connection to CW2 & CW3

CW	What you did	What PyTorch does for you
CW2	Manual gradient: `dW = X.T @ grad`	`loss.backward()`
CW3	Manual backprop: chain of `layer.backward()`	`loss.backward()`
CW4	Define architecture only	PyTorch autograd handles everything

Key Insight

`loss.backward()` in PyTorch automatically does what you built by hand in CW3 — propagating gradients through all layers via the chain rule. In CW4, you focus on **architecture design** rather than gradient mechanics.

Learning Objectives

By the end of this coursework you should be able to:

1. Design CNN architectures using PyTorch's `nn.Module`.
2. Understand why convolutional layers are effective for image data (spatial structure, parameter sharing, translation invariance).
3. Explore the effect of hyperparameters: optimizer, learning rate, architecture, data augmentation.
4. Compare CNN performance with CW2 (logistic regression) and CW3 (MLP), and explain the progression.

Setup

CW4 uses PyTorch's built-in FashionMNIST downloader (no need to run `setup_data.py` again, though it won't hurt). Ensure PyTorch is installed:

```
cd code
pip install -r requirements.txt
cd cw4_cnn
```

What You Need to Implement

File	Function / Method	Description
`model.py`	`SimpleCNN.__init__()`	Define conv/pool/fc layers
`model.py`	`SimpleCNN.forward(x)`	Forward pass through the network
`model.py`	`DeepCNN.__init__()`	Advanced architecture (bonus)
`model.py`	`DeepCNN.forward(x)`	Forward pass (bonus)

That's it! You only need to implement `model.py`. Everything else is provided: data loading (`dataset.py`), training loop (`trainer.py`), configuration (`config.py`), and entry point (`run.py`).

Tasks

Task 1 — Implement SimpleCNN

Design and implement a CNN in `model.py` that classifies $1 \times 28 \times 28$ grayscale images into 10 classes. Your network should use:

- **Convolutional layers** (`nn.Conv2d`) to extract spatial features.
- **Pooling layers** (`nn.MaxPool2d`) to downsample feature maps.
- **Activation functions** (`nn.ReLU`) for non-linearity.
- **Fully connected layers** (`nn.Linear`) for classification.

Requirements:

- Input shape: `(batch_size, 1, 28, 28)`.
- Output shape: `(batch_size, 10)` — raw logits (do **not** apply softmax; it is included in `nn.CrossEntropyLoss`).
- The model should achieve reasonable test accuracy on FashionMNIST.

Design Tips

- Use `nn.Sequential` to group layers into blocks for cleaner code.
- Track tensor shapes at each layer to ensure dimensions match (especially at the flatten/FC boundary).
- A typical pattern: `Conv` → `ReLU` → `Pool` → `Conv` → `ReLU` → `Pool` → `Flatten` → `FC` → `FC`.
- Increasing the number of filters as spatial dimensions shrink is a common strategy.

Task 2 — Hyperparameter Exploration

Explore at least **two** of the following and report the results:

- **Optimizer:** Adam vs. SGD (with momentum).
- **Learning rate:** Try several values.
- **Architecture variations:** Different numbers of filters, FC layer sizes, number of conv blocks.
- **Batch size.**

Task 3 — Data Augmentation

Train your SimpleCNN with and without data augmentation and compare:

```
# Without augmentation (default)
python run.py

# With augmentation
python run.py --use_augmentation
```

The provided augmentation pipeline (in `dataset.py`) applies random horizontal flips and random crops. Report the effect on test accuracy and the train-validation gap.

Task 4 — DeepCNN (Bonus)

Design a more advanced CNN architecture targeting higher accuracy. Consider using:

- **Batch normalization** (`nn.BatchNorm2d`) for training stability.
- **Dropout** (`nn.Dropout`) for regularization.
- **Residual / skip connections** for deeper networks.
- **Global average pooling** (`nn.AdaptiveAvgPool2d`) to reduce parameters.

Task 5 — Cross-Coursework Comparison

Compare the performance of all three approaches across CW2–CW4:

- Logistic Regression (CW2)
- MLP (CW3)
- CNN (CW4)

Discuss: Why does CNN outperform MLP? What role do convolutional layers play in capturing spatial patterns? How does parameter sharing in CNNs compare to fully connected layers?

How to Run

```
# Quick mode (for debugging --- fewer epochs)
python run.py --quick

# Train SimpleCNN (default)
python run.py

# Train with data augmentation
```

```
python run.py --use_augmentation

# Train DeepCNN (bonus)
python run.py --model_type deep_cnn --num_epochs 20

# Different optimizer and learning rate
python run.py --optimizer_type sgd --learning_rate 0.01
```

How to Verify Your Implementation

Shape and Gradient Checks

```
# From the code/ directory
python -m tests.test_cw4
```

This test verifies:

- **Output shape:** Input `(B, 1, 28, 28)` produces output `(B, 10)`.
- **Gradient flow:** All parameters receive non-zero gradients after `loss.backward()`.
- **Variable batch sizes:** The model works with different batch sizes (1 and 16).
- **Parameter count:** Reports the total number of trainable parameters.

Training Convergence

After running `python run.py`, your SimpleCNN should achieve meaningful improvement over CW3's MLP, demonstrating the advantage of convolutional feature extraction.

Outputs

After training, the following files are generated in the `outputs/` directory:

- `training_summary.png` — Training loss and validation accuracy curves.
- `confusion_matrix.png` — Confusion matrix on the test set.
- `sample_predictions.png` — Sample images with predicted and true labels.
- `results.json` — Numerical results.
- `best_model.pth` — Best model checkpoint (by validation accuracy).

Deliverables

Submit a single **zip file** with the following structure:

```
cw4_cnn/                    # The entire code directory
  model.py
  run.py
  config.py
  dataset.py
  trainer.py
  outputs/                  # Your training outputs
  ...
report.pdf                  # Your report
```

Report (use the template in `report_template/`, see example report in `report_examples/`):

- CNN architecture description and design choices.
- Hyperparameter exploration (≥ 2 hyperparameters) with tables/plots.
- Data augmentation comparison.
- Cross-coursework comparison: CW2 vs. CW3 vs. CW4 with discussion of why CNN excels at image tasks.

Grading Rubric (100 points)

Component	**Points**
SimpleCNN architecture (valid conv/pool/fc layers, correct shapes)	20
Forward pass (correct sequential computation)	10
Training convergence (SimpleCNN achieves reasonable accuracy)	15
DeepCNN / advanced architecture (BatchNorm, Dropout, etc.)	15
Data augmentation (implements and reports effect)	5
Hyperparameter exploration report (optimizer, LR, architecture)	15
Cross-coursework comparison (CW2 vs. CW3 vs. CW4)	10
Code quality (proper `nn.Module`, device handling)	10
Total	**100**
Bonus (capped at 100 total): Learning rate scheduler	+3
Bonus (capped at 100 total): Model ensembling	+4
Bonus (capped at 100 total): Filter visualization	+3

Note: The final grade including bonus will not exceed 100 points.

Project Structure

```
cw4_cnn/
  run.py              # Entry point (provided)
  config.py           # Hyperparameter defaults (provided)
  trainer.py          # Training loop (provided)
  dataset.py          # PyTorch DataLoader + augmentation (provided)
  model.py            # (*) SimpleCNN, DeepCNN

common/               # Shared utilities (provided, same as CW2/CW3)
tests/
  test_cw4.py         # Shape and gradient flow checks

(*) = file you need to implement
```

Useful PyTorch Layers

Layer	Usage
`nn.Conv2d(in_ch, out_ch, kernel_size, padding)`	2D convolution
`nn.MaxPool2d(kernel_size)`	Max pooling (downsampling)
`nn.ReLU()`	ReLU activation
`nn.Linear(in_features, out_features)`	Fully connected layer
`nn.Flatten()`	Flatten spatial dims to 1D
`nn.Sequential(...)`	Group layers into a block
`nn.BatchNorm2d(num_features)`	Batch normalization (bonus)
`nn.Dropout(p)`	Dropout regularization (bonus)
`nn.AdaptiveAvgPool2d(output_size)`	Global average pooling (bonus)

Estimated Running Times (CPU)

Task	Quick Mode	Full Run
SimpleCNN (15 epochs)	∼3 min	∼15 min
DeepCNN (20 epochs)	∼5 min	∼25 min

Academic Integrity

- You may discuss high-level ideas with other students.
- You must write your own code.
- Do not copy implementations from other students or online sources.