Coursework 3: Multi-Layer Perceptron

Backpropagation from Scratch

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

Overview

You will implement a **multi-layer perceptron (MLP)** with **backpropagation** from scratch using NumPy. This coursework **builds directly on CW2**: you will reuse and generalize the concepts you implemented there (softmax, cross-entropy, gradient computation) into modular, chainable layers.

- **Dataset:** FashionMNIST (same as CW2).
- **Framework:** NumPy only (no PyTorch for this coursework).
- **Model:** Multi-layer network with configurable depth, width, and activation function.

Connection to CW2

This coursework is **not** a fresh start — it generalizes CW2's concepts into reusable components.

CW2 Concept	CW3 Generalization
`softmax()` in `losses.py`	Reused in `CrossEntropyLoss.forward()`
`cross_entropy_loss()` returning (loss, grad)	Wrapped into `CrossEntropyLoss` class with `forward()` + `backward()`
Optimizer: `dW = X.T @ grad`	Becomes `Linear.backward()`: same math, plus propagation via `grad @ W.T`
`GradientDescent`	Extended to `SGD` with momentum

Key Insight

In CW2, your optimizer computed `dW = X.T @ grad_logits` and `db = grad_logits.sum(axis=0)`. That was secretly the **backward pass of a single Linear layer**! In CW3, you will generalize this by adding `grad_input = grad_output @ W.T` to propagate gradients to the previous layer, enabling multi-layer backpropagation.

Learning Objectives

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

1. Implement forward and backward passes for Linear, ReLU, and Sigmoid layers.
2. Assemble layers into an MLP and perform backpropagation via the chain rule.
3. Implement SGD with momentum.
4. Explore how hyperparameters (depth, width, learning rate, activation) affect performance.
5. Compare MLP with CW2's logistic regression and explain why non-linearity helps.

Setup

Ensure you have already run `setup_data.py` (from CW2). Then:

```
cd code/cw3_mlp
```

What You Need to Implement

File	Function / Method	Description
`layers.py`	`Linear.forward(X)`	$\mathbf{z} = \mathbf{X}\mathbf{W} + \mathbf{b}$, cache $\mathbf{X}$
`layers.py`	`Linear.backward(grad_output)`	Compute grad_W, grad_b, return grad_input
`layers.py`	`ReLU.forward(X)`	$\max(0, \mathbf{X})$, cache input
`layers.py`	`ReLU.backward(grad_output)`	Zero gradient where input ≤ 0
`layers.py`	`Sigmoid.forward(X)`	$\sigma(\mathbf{X}) = 1/(1 + e^{-\mathbf{X}})$
`layers.py`	`Sigmoid.backward(grad_output)`	$\nabla = \text{grad} \cdot \sigma \cdot (1-\sigma)$
`model.py`	`MLP.__init__(...)`	Build list of layers
`model.py`	`MLP.forward(X)`	Sequential forward
`model.py`	`MLP.backward(grad_output)`	Reverse-order backward
`losses.py`	`CrossEntropyLoss.forward(logits, labels)`	Softmax + cross-entropy (reuse CW2)
`losses.py`	`CrossEntropyLoss.backward()`	Return cached gradient
`optimizer.py`	`SGD.__init__(params, lr, momentum)`	Initialize velocities
`optimizer.py`	`SGD.step()`	Update params with momentum
`optimizer.py`	`SGD.zero_grad()`	Reset all gradients to zero

Do not modify the following provided files: `run.py`, `config.py`, `trainer.py`.

Tasks

Task 1 — Implement Layer Forward and Backward Passes

Implement three layer types in `layers.py`. Each layer must implement a `forward(X)` and a `backward(grad_output)` method.

Linear Layer

Forward: Compute $\mathbf{z} = \mathbf{X}\mathbf{W} + \mathbf{b}$ and cache $\mathbf{X}$ for the backward pass.

Backward: Given $\nabla_{\mathbf{z}} L$ (`grad_output`), compute:

$$\nabla_{\mathbf{W}} L = \mathbf{X}_{\text{cache}}^{\top} \cdot \nabla_{\mathbf{z}} L \quad \text{(weight gradient — same as CW2 optimizer!)} \tag{1}$$

$$\nabla_{\mathbf{b}} L = \textstyle\sum_i \nabla_{\mathbf{z}_i} L \quad \text{(bias gradient — same as CW2 optimizer!)} \tag{2}$$

$$\nabla_{\mathbf{X}} L = \nabla_{\mathbf{z}} L \cdot \mathbf{W}^{\top} \quad (\textbf{new}\text{: propagate gradient to previous layer}) \tag{3}$$

Store `grad_W` and `grad_b` on the layer. Return $\nabla_{\mathbf{X}} L$.

ReLU Activation

Forward: $\text{ReLU}(x) = \max(0, x)$. Cache the input.

Backward: The gradient passes through where the input was positive, and is zeroed where it was non-positive:

$$\nabla_{\mathbf{X}} L = \nabla_{\mathbf{z}} L \odot \mathbf{1}[\mathbf{X} > 0] \tag{4}$$

Sigmoid Activation

Forward: $\sigma(x) = \frac{1}{1+e^{-x}}$. Cache the output.
Backward: $\nabla_{\mathbf{X}} L = \nabla_{\mathbf{z}} L \odot \sigma \odot (1 - \sigma)$.

Task 2 — Implement the MLP

In `model.py`, build the MLP by assembling layers:

- `__init__`: Construct a list of layers alternating `Linear` and activation layers. The final `Linear` layer should have **no activation** (softmax is applied in the loss function).
- `forward(X)`: Pass the input through each layer sequentially.
- `backward(grad_output)`: Propagate the gradient through layers in **reverse order**.

Task 3 — Implement Cross-Entropy Loss

In `losses.py`, implement the `CrossEntropyLoss` class:

- `forward(logits, labels)`: Compute softmax probabilities, then the cross-entropy loss. Cache what you need for backward.
- `backward()`: Return the gradient of the loss w.r.t. logits.

You may copy or import your softmax and cross-entropy implementations from CW2.

Task 4 — Implement SGD with Momentum

In `optimizer.py`, implement SGD:

$$\mathbf{v} \leftarrow \mu \cdot \mathbf{v} + \nabla L \tag{5}$$

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \cdot \mathbf{v} \tag{6}$$

where μ is the momentum coefficient. When $\mu = 0$, this reduces to your CW2 gradient descent.

Task 5 — Hyperparameter Exploration

Explore at least **three** of the following hyperparameters and report the results with plots:

- Hidden layer dimension (width)
- Number of hidden layers (depth)
- Learning rate
- Activation function (ReLU vs. Sigmoid)
- Momentum
- Batch size

Task 6 — Compare with CW2

Compare your best MLP configuration with your CW2 logistic regression results. Discuss why the MLP outperforms logistic regression and the role of non-linear activation functions.

How to Run

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

# Default configuration
python run.py

# Custom architecture
python run.py --hidden_dims 256 128 --activation relu --learning_rate 0.01

# Hyperparameter sweeps (generates comparison plots)
python run.py --sweep hidden_dim
python run.py --sweep num_layers
python run.py --sweep learning_rate
```

How to Verify Your Implementation

Gradient Checking

```
# From the code/ directory
python -m tests.test_cw3
```

This tests gradient correctness for:

- **Linear layer**: forward output shape, backward gradients (grad_input, grad_W, grad_b).
- **ReLU layer**: correct zero-threshold behavior.
- **Sigmoid layer**: correct derivative computation.
- **CrossEntropyLoss**: forward/backward correctness.
- **Full MLP**: forward shape and backward gradient flow through all layers.

All gradient checks should pass with relative error $< 10^{-5}$.

Training Convergence

After running `python run.py` with default settings, you should observe meaningful improvement over CW2's logistic regression.

How Backpropagation Works in This Code

The training loop (provided in `trainer.py`) performs five steps each iteration:

1. **Forward pass:** `logits = model.forward(X_batch)`
2. **Loss forward:** `loss = loss_fn.forward(logits, y_batch)`
3. **Loss backward:** `grad = loss_fn.backward()` — gradient w.r.t. logits
4. **Model backward:** `model.backward(grad)` — propagates through all layers in reverse
5. **Optimizer step:** `optimizer.step()` — updates all parameters using stored gradients

This is the same chain rule that PyTorch's `loss.backward()` performs automatically. In CW4, you will see this happen with a single line of code.

Deliverables

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

```
cw3_mlp/                    # The entire code directory
  layers.py
  model.py
  losses.py
  optimizer.py
  run.py
  config.py
  trainer.py
  outputs/                  # Your training outputs
  ...
report.pdf                  # Your report
```

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

- Implementation overview, including connection to CW2.
- Hyperparameter exploration ($\geq$ 3 hyperparameters) with tables and plots.
- Comparison with CW2 logistic regression.
- Analysis of why MLP outperforms logistic regression.

Grading Rubric (100 points)

Component	**Points**
Linear layer forward + backward (passes gradient check)	15
ReLU forward + backward (correct zero-threshold gradient)	10
Sigmoid forward + backward (correct computation)	5
CrossEntropyLoss forward + backward (reuses CW2 logic)	10
MLP assembly: forward + backward (correct layer chaining)	15
SGD optimizer with momentum	10
Training convergence (reasonable test accuracy)	10
Hyperparameter exploration report ($\geq$ 3 hyperparams with plots)	15
Comparison with CW2	5
Code quality (clean, commented)	5
Total	**100**
Bonus (capped at 100 total): Adam optimizer	+5
Bonus (capped at 100 total): Dropout layer	+5

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

Project Structure

```
cw3_mlp/
  run.py              # Entry point (provided)
  config.py           # Hyperparameter defaults (provided)
  trainer.py          # Training loop (provided)
  layers.py           # (*) Linear, ReLU, Sigmoid layers
  model.py            # (*) MLP class
  losses.py           # (*) CrossEntropyLoss (reuse CW2)
  optimizer.py        # (*) SGD with momentum

common/               # Shared utilities (provided, same as CW2)
tests/
  test_cw3.py         # Gradient checking tests

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

Estimated Running Times (CPU)

Task	Quick Mode	Full Run
Single configuration	∼2 min	∼20 min
Hyperparameter sweep (5 values)	∼10 min	∼1.5 hr

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.