Deep Learning — HW1

February 4, 2026

Problem 1: Warmup — Softmax + Matrix Gradients (15 pts)

1. **Temperature-scaled softmax Jacobian (6 pts).**
   Let $z \in \mathbb{R}^n$ and define the temperature softmax

$$y_i = \mathrm{softmax}_\tau(z)_i = \frac{e^{z_i/\tau}}{\sum_{j=1}^{n} e^{z_j/\tau}}, \quad \tau > 0.$$

   Derive the Jacobian $J = \frac{\partial y}{\partial z}$ and show it can be written in matrix form as

$$\frac{\partial y}{\partial z} = \frac{1}{\tau}\left(\mathrm{diag}(y) - yy^\top\right).$$

   (You may use indicator notation $\mathbf{1}[i = j]$ if helpful.)

2. **Gradient w.r.t. A and b in an affine least-squares objective (6 pts).**
   Let $A \in \mathbb{R}^{d\times m}$, $b \in \mathbb{R}^d$, and $x \in \mathbb{R}^m$. Consider

$$f(A, b) = \frac{1}{2}\|Ax + b - t\|_2^2,$$

   where $t \in \mathbb{R}^d$ is a fixed target vector. Write down $\nabla_A f$ and $\nabla_b f$ in clean matrix/vector form.

3. **Softmax vs. sigmoid: when and why? (3 pts).**
   Give *one* real ML task where softmax is the right output activation and sigmoid is not, and *one* task where sigmoid is appropriate and softmax is not. Explain the difference in terms of probabilistic interpretation (mutual exclusivity vs. independent labels) and loss choice.

Problem 2: Linear Regression with Huber Loss (15 pts)

We define residuals $r_i(w) = y_i - w^\top x_i$, where $w \in \mathbb{R}^d$, $x_i \in \mathbb{R}^d$, and $y_i \in \mathbb{R}$. The Huber loss is

$$\ell_\delta(r) = \begin{cases} \frac{1}{2}r^2 & |r| \le \delta, \\ \delta\left(|r| - \frac{1}{2}\delta\right) & |r| > \delta, \end{cases}$$

and the objective is $L_\delta(w) = \sum_{i=1}^{n} \ell_\delta(r_i(w))$.

1. **Gradient derivation (6 pts).**
   Derive $\nabla_w L_\delta(w)$. Your final answer should be a sum over i with a piecewise expression depending on $|r_i(w)| \le \delta$ versus $|r_i(w)| > \delta$.

2. **Optimal w: closed form or not? (4 pts).**
   Does minimizing $L_\delta(w)$ admit a single closed-form solution like ordinary least squares? Explain clearly. If not, briefly explain what property breaks the normal equations and name one standard numerical approach used in practice.

3. **Add L_2 regularization (3 pts).**
   Define the new objective with ridge regularization:
$$\tilde{L}(w) = \sum_{i=1}^{n} \ell_\delta(r_i(w)) + \frac{\lambda}{2}\|w\|_2^2.$$
   Write down $\nabla_w \tilde{L}(w)$ and explain in 2–3 sentences how λ changes the solution behavior, especially under outliers and high-dimensional settings.

4. **ML pipeline reflection (2 pts).**
   Relate this to the 3-step ML recipe (define objective → optimize → evaluate). In 3–4 sentences: how does Huber change (i) optimization behavior and (ii) evaluation robustness compared to squared loss?

Problem 3: Adam vs. SGD with Momentum (15 pts)

Recall Adam (scalar parameter w) at step t:
$$g_t = \nabla \ell_t(w_t), \quad m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$
Bias-correct:
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t},$$
Update:
$$w_{t+1} = w_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}.$$

1. **Compute the first two Adam updates explicitly (6 pts).**
   Assume $m_0 = 0$ and $v_0 = 0$. Given gradients g_1 and g_2, write explicit expressions for:
   - $m_1, v_1, \hat{m}_1, \hat{v}_1, w_2$,
   - $m_2, v_2, \hat{m}_2, \hat{v}_2, w_3$.

   Keep it symbolic (in terms of $g_1, g_2, \alpha, \beta_1, \beta_2, \varepsilon$).

2. **Compare to momentum SGD update form (5 pts).**
   Momentum SGD (scalar) is:
$$u_t = \mu u_{t-1} + g_t, \quad w_{t+1} = w_t - \alpha u_t.$$
   Compare Adam vs. momentum SGD in terms of:
   (a) whether the effective step size depends on gradient magnitude,
   (b) what happens when gradients are consistently scaled by a constant factor c,
   (c) the role of ε (why it exists, qualitatively).

3. **Short explanation: noisy gradients + sparse features (4 pts).**
   In 3–5 sentences, explain how Adam differs from momentum SGD in:
   (a) learning-rate adaptation across parameters, and
   (b) behavior on noisy gradients or rarely-updated (sparse) features (e.g., in NLP with large embeddings).

Problem 4: Blood Cell Classification with MLPs (15 points)

In this problem, you will train a multi-layer perceptron (MLP) to classify microscopy images of blood cells. Medical image classification is a rapidly growing application of deep learning, and this exercise will give you hands-on experience with a real-world healthcare dataset.

Dataset: BloodMNIST

BloodMNIST is part of the MedMNIST collection — a standardized set of medical image datasets designed for benchmarking. It contains **17,092 microscopy images** of individual blood cells, each labeled as one of **8 cell types**:

Class ID	Cell Type
0	Basophil
1	Eosinophil
2	Erythroblast
3	Immature Granulocyte (IG)
4	Lymphocyte
5	Monocyte
6	Neutrophil
7	Platelet

Each image is **28 × 28 pixels with 3 color channels (RGB)**. The dataset is pre-split into training (11,959), validation (1,712), and test (3,421) sets.

Your Tasks

Q4.1: Data Loading and Exploration (2 points)

1. Install and load BloodMNIST using the `medmnist` package
2. Report the number of samples in train/val/test splits
3. Display a grid of **16 random training images** (2 per class) with their class labels
4. Plot the **class distribution** (bar chart) for the training set. Is the dataset balanced?

Q4.2: Build and Train an MLP (5 points)

1. Flatten each $28 \times 28 \times 3$ RGB image into a **2,352-dimensional** input vector
2. Build a 3-layer MLP in PyTorch with the following architecture:
   - Input: 2352 $\rightarrow$ Hidden 1: 256 (ReLU) $\rightarrow$ Hidden 2: 128 (ReLU) $\rightarrow$ Output: 8
3. Use the **Adam optimizer** with learning rate `1e-3` and **CrossEntropyLoss**
4. Train for **30 epochs** with batch size 64
5. Plot **training loss** and **validation loss** on the same figure (x-axis: epoch)
6. Plot **training accuracy** and **validation accuracy** on the same figure

Q4.3: Evaluation and Analysis (5 points)

1. Report the **final test accuracy** (as a percentage)
2. Generate a **confusion matrix** (8×8) for the test set using `sklearn.metrics.confusion_matrix` and visualize it as a heatmap with class labels
3. Identify the **two most confused cell type pairs** (highest off-diagonal values in the confusion matrix)
4. Compute and report **per-class precision and recall** using `sklearn.metrics.classification_report`
5. Which cell type has the **lowest recall**? Examine 5 misclassified examples of this cell type and hypothesize why the model struggles with it.

Q4.4: Prediction Confidence Analysis (3 points)

For this analysis, categorize predictions into four quadrants based on confidence and correctness:

	Correct	Incorrect
High Confidence (max prob > 0.9)	✓ Confident	$\times$ Confident
Low Confidence (max prob < 0.6)	✓ Uncertain	$\times$ Uncertain

1. Find and display **2 examples from each quadrant** (8 images total) with:
   - The image
   - True label
   - Predicted label
   - Prediction confidence (max softmax probability)
2. **Written Analysis (3-4 sentences)**: What visual characteristics distinguish the "Incorrect but Confident" examples? Why might the model be overconfident on these samples?

Problem 5: Weight Initialization and Training Dynamics (15 points)

Proper weight initialization is crucial for training deep neural networks. Poor initialization can cause activations to vanish (all zeros) or explode (very large values), making learning impossible. In this problem, you will empirically investigate how different initialization schemes affect activation statistics and training dynamics.

Your Tasks

Q5.1: Implement Initialization Schemes (3 points)

Implement the following four initialization methods for a weight matrix of shape `(fan_in, fan_out)`:

1. **Zero Initialization**: All weights set to 0
2. **Small Random**: Weights sampled from $\mathcal{N}(0, \sigma^2 = 0.01^2)$
3. **Xavier/Glorot**: Weights sampled from $\mathcal{N}(0, \sigma^2 = \frac{2}{fan\_in + fan\_out})$
4. **He/Kaiming**: Weights sampled from $\mathcal{N}(0, \sigma^2 = \frac{2}{fan\_in})$

Note: The second parameter in the normal distribution notation above represents the ***variance*** *(σ^2), not the standard deviation (σ).*

Write a function `initialize_weights(shape, method)` that returns a PyTorch tensor initialized according to the specified method. Biases should always be initialized to zero.

Q5.2: Activation Statistics Before Training (5 points)

Build a **6-layer MLP** for MNIST classification:

- Architecture: $784 \to 256 \to 256 \to 256 \to 256 \to 256 \to 10$
- Use **Tanh** activations for all hidden layers

For each of the 4 initialization methods:

1. Initialize the network (do NOT train yet)
2. Forward pass a batch of 256 random MNIST images

3. Record the **mean** and **standard deviation** of activations at each of the 5 hidden layers
4. Create a figure with **2 subplots**:
    - Subplot 1: Mean activation vs. layer depth (4 lines, one per init method)
    - Subplot 2: Std of activation vs. layer depth (4 lines, one per init method)

Written Analysis (3-4 sentences): Which initialization methods show vanishing activations (std $\rightarrow 0$)? Which maintain stable activation statistics across layers? Explain why Xavier initialization is designed for Tanh/Sigmoid activations.

Q5.3: Training Dynamics Comparison (4 points)

Now train all 4 networks for **10 epochs** on MNIST:

- Optimizer: SGD with learning rate 0.1
- Batch size: 128
- Loss: CrossEntropyLoss

1. Plot all 4 **training loss curves** on the same figure
2. Report the **test accuracy** after 10 epochs for each initialization

Note: *Zero initialization is* ***expected to fail completely*** *— the network will produce exactly zero gradients due to symmetry (all neurons compute the same values and receive the same updates). This is the "symmetry breaking" problem. If your zero-initialized network shows no learning, your implementation is likely correct.*

Initialization	Test Accuracy (%)
Zero	
Small Random	
Xavier	
He	

Q5.4: ReLU Activation Experiment (3 points)

Repeat Q5.2 and Q5.3 but replace **Tanh** with **ReLU** activations.

1. Create the same activation statistics plots (mean and std vs. layer depth) for all 4 initializations with ReLU
2. Train for 10 epochs and report test accuracies

Written Analysis (4-5 sentences):

- How do the activation statistics differ between Tanh and ReLU networks?
- Which initialization works best for ReLU? Why is He initialization specifically designed for ReLU?
- Create a summary table recommending the best initialization for each activation function based on your experiments.

Problem 6: Building a Neural Network Library from Scratch (25 points)

In this problem, you will implement the core components of a neural network library using only NumPy — no PyTorch autograd allowed. This exercise will deepen your understanding of how frameworks like PyTorch work under the hood.

You will build modular components with a consistent API, then combine them to train on real data. **All gradient computations must be implemented manually.**

Library Architecture

Your library should follow this structure:

```
mytorch/
|-- nn/
|   |-- modules/
|   |   |-- linear.py      # Linear (fully-connected) layer
|   |   |-- activation.py  # ReLU, Sigmoid activations
|   |   `-- loss.py        # SoftmaxCrossEntropy loss
|   |-- sequential.py      # Sequential container
|   `-- optim.py           # SGD optimizer
`-- utils/
    `-- gradient_check.py  # Numerical gradient verification
```

Module API

Every module must implement:

```python
class Module:
    def forward(self, x):
        """Compute output given input x. Cache values needed for backward."""
        raise NotImplementedError

    def backward(self, grad_output):
        """Compute gradients given upstream gradient. Return gradient w.r.t. input."""
        raise NotImplementedError

    def get_parameters(self):
        """Return list of parameter arrays (empty for activations)."""
        return []

    def get_gradients(self):
        """Return list of gradient arrays (same order as parameters)."""
        return []
```

Your Tasks

Q6.1: Implement Linear Layer (5 points)

Implement a fully-connected layer: $Z = XW^T + b$

Where:

- X: input of shape `(batch_size, in_features)`
- W: weights of shape `(out_features, in_features)`
- b: bias of shape `(out_features,)`
- Z: output of shape `(batch_size, out_features)`

Forward pass:

$$Z = XW^T + b$$

Backward pass: Given $\frac{\partial L}{\partial Z}$ (shape: `batch_size` $\times$ `out_features`), compute:

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T X$$

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{batch} \frac{\partial L}{\partial Z_i}$$

$$\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Z} \cdot W$$

Initialize weights using **Xavier/Glorot initialization** (consistent with Problem 5, Q5.1):

$$W \sim \mathcal{N}\left(0, \sigma^2 = \frac{2}{n_{in} + n_{out}}\right)$$

Note: Xavier initialization is theoretically designed for Tanh/Sigmoid activations. While He initialization ($\sigma^2 = \frac{2}{n_{in}}$) is optimal for ReLU networks, Xavier works adequately for shallow networks like the one in Q6.7. You may optionally implement both and compare.

Q6.2: Implement Activations (4 points)

ReLU:

$$\text{forward: } y = \max(0, x)$$

$$\text{backward: } \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \odot \mathbf{1}_{x>0}$$

Sigmoid:

$$\text{forward: } y = \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\text{backward: } \frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \odot \sigma(x)(1 - \sigma(x))$$

Use numerically stable implementations (clip inputs to avoid overflow).

Q6.3: Implement SoftmaxCrossEntropy Loss (5 points)

Implement the combined softmax and cross-entropy loss for numerical stability:

Forward pass:

$$\hat{y}_i = \text{softmax}(z)_i = \frac{e^{z_i - \max(z)}}{\sum_j e^{z_j - \max(z)}}$$

$$L = -\sum_i y_i \log(\hat{y}_i)$$

Where y is the one-hot encoded true label.

Backward pass:

$$\frac{\partial L}{\partial z} = \hat{y} - y$$

This simplified gradient is one of the beautiful results of combining softmax with cross-entropy!

Batch handling: The loss should be **averaged over the batch**:

$$L = -\frac{1}{N} \sum_{n=1}^{N} \sum_i y_i^{(n)} \log(\hat{y}_i^{(n)})$$

Consequently, the backward gradient should also be divided by the batch size N:

$$\frac{\partial L}{\partial z^{(n)}} = \frac{1}{N}(\hat{y}^{(n)} - y^{(n)})$$

Q6.4: Implement Sequential Container (2 points)

Implement a `Sequential` class that:

1. Stores a list of modules
2. `forward(x)`: passes input through all modules in order
3. `backward(grad)`: passes gradient backward through all modules in reverse order
4. `get_parameters()`: returns all parameters from all modules
5. `get_gradients()`: returns all gradients from all modules

Q6.5: Implement SGD Optimizer (2 points)

Implement stochastic gradient descent:

```
class SGD:
    def __init__(self, parameters, lr=0.01):
        self.parameters = parameters  # List of parameter arrays
        self.lr = lr

    def step(self, gradients):
        # Update parameters using gradients.
        for param, grad in zip(self.parameters, gradients):
            param -= self.lr * grad

    def zero_grad(self):
        # Zero out stored gradients (if applicable).
        pass
```

Q6.6: Implement Gradient Checking (3 points)

Implement numerical gradient verification:

```
def gradient_check(model, x, y, epsilon=1e-5):
#    Compare analytical gradients with numerical gradients.
#
#    For each parameter p:
#        numerical_grad[i] = (L(p[i] + eps) - L(p[i] - eps)) / (2 * eps)
#
#    Returns: max relative error across all parameters
```

The relative error should be computed as:

$$\text{relative\_error} = \frac{||\text{analytical} - \text{numerical}||}{||\text{analytical}|| + ||\text{numerical}||}$$

Compute this error ***element-wise*** *for each individual parameter entry (each weight and bias value), then return the* ***maximum*** *relative error across all parameter elements in the network. This approach catches any single incorrect gradient computation.*

A correct implementation should achieve relative error $< 10^{-5}$.

Q6.7: Train and Evaluate (4 points)

Using your library, train a neural network on MNIST:

Architecture: 784 → 128 (ReLU) → 64 (ReLU) → 10 (SoftmaxCE)

Training:

1. First, run gradient check on a small batch (e.g., 5 samples) and verify error $< 10^{-5}$

2. Train for 3 epochs on the full training set
3. Use learning rate 0.1, batch size 64

Deliverables:

1. Screenshot/output of gradient check showing relative error $< 10^{-5}$
2. Plot of training loss vs. iteration (or epoch)
3. Report final **test accuracy** (should achieve $> 90\%$)
4. Display 10 random test images with predicted vs. true labels

Academic Integrity

You may discuss concepts with classmates, but all code and written analysis must be your own. Your experimental results (accuracies, confusion matrices, plots) should reflect your actual training runs — do not fabricate or copy results.