Deep Learning E1394
Problem Set 1

Padma Bareddy (236167), Nadine Daum (245963), Saurav Jha (249354)

Team 1D

Due on Oct 3, 2025 (EOD)

Repository: assignment-1-ps-1-d

1 Theoretical Part

Acknowledgment: To work through the theoretical questions, we used Perplexity, Claude, Wolfram Alpha, and ChatGPT, along with some web searches.

1.1 Optimization

The function given,

$$f(x, y, z) = x^2 + y^2 + z^2 - xyz.$$

The gradient of this function can be found by taking the partial derivative of this function with respect to x, y, and z.

Partial derivatives of this functions

$$\frac{\partial f}{\partial x} = 2x - yz, \qquad \frac{\partial f}{\partial y} = 2y - xz, \qquad \frac{\partial f}{\partial z} = 2z - xy.$$

Stationary equations

Setting the partials equal to zero gives the system

$$-2x - yz = 0, \quad \text{(Eq. 1)} \tag{1}$$
$$2y - xz = 0, \quad \text{(Eq. 2)} \tag{2}$$
$$2z - xy = 0, \quad \text{(Eq. 3)} \tag{3}$$

Solution of the equation

Case 1: At least one coordinate is zero

Assume $x = 0$, from (Eq. 1) we get $-yz = 0$, so $y = 0$ or $z = 0$.

- If $y = 0$, then (Eq. 3) gives $2z = 0 \Rightarrow z = 0$.
- If $z = 0$, then (Eq. 2) gives $2y = 0 \Rightarrow y = 0$.

The only solution with any coordinate zero is

$$(0, 0, 0).$$

Case 2: All coordinates nonzero — solve by substitution

From the stationary equations

$$2x - yz = 0, \qquad 2y - xz = 0, \qquad 2z - xy = 0$$

Rewriting them as

$$x = \frac{yz}{2}, \qquad y = \frac{xz}{2}, \qquad z = \frac{xy}{2},$$

Where $x, y, z \neq 0$

Substituting the expression for x into the equation for y:

$$y = \frac{xz}{2} = \frac{\left(\frac{yz}{2}\right) z}{2} = \frac{yz^2}{4}.$$

Since $y \neq 0$, divide both sides by y to obtain

$$1 = \frac{z^2}{4} \quad \Longrightarrow \quad z^2 = 4 \quad \Longrightarrow \quad z = \pm 2.$$

Next, substituting the found z into $x = \frac{yz}{2}$ and $y = \frac{xz}{2}$. From symmetry (or repeating the same substitution) we get similarly

$$x^2 = z^2 = 4, \qquad y^2 = z^2 = 4,$$

hence $|x| = |y| = |z| = 2$

Finally, the three substitutions together imply

$$xyz = \frac{yz \cdot xz \cdot xy}{2 \cdot 2 \cdot 2} = 8,$$

So the product xyz must be $8 > 0$

Therefore the number of negative coordinates must be even. Thus in the solution we should have two coordinates with negative signs or all positive integers (with absolute value 2 each) are

$$(2,2,2), \quad (2,-2,-2), \quad (-2,2,-2), \quad (-2,-2,2).$$

Critical Points

Including the first solution, the total number of critical points are:

$$(0,0,0), \quad (2,2,2), \quad (2,-2,-2), \quad (-2,2,-2), \quad (-2,-2,2)$$

Hessian Matrix

The Hessian matrix of the function (calculated from Wolfram Alpha) is:

$$H(x,y,z) = \begin{bmatrix} 2 & -z & -y \\ -z & 2 & -x \\ -y & -x & 2 \end{bmatrix}$$

Eigenvalue Analysis at Each Critical Point

At **Point** $(2,2,2)$
Hessian matrix:

$$H(2,2,2) = \begin{bmatrix} 2 & -2 & -2 \\ -2 & 2 & -2 \\ -2 & -2 & 2 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 4$, $\lambda_2 = 4$, $\lambda_3 = -2$

Classification: Saddle point (mixed sign eigenvalues)

At **Point** $(0,0,0)$
Hessian matrix:

$$H(0,0,0) = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 2$, $\lambda_2 = 2$, $\lambda_3 = 2$

Classification: Local minimum (all eigenvalues positive, positive definite)

At **Point** $(2,-2,-2)$
Hessian matrix:

$$H(2,-2,-2) = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & -2 \\ 2 & -2 & 2 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 4$, $\lambda_2 = 4$, $\lambda_3 = -2$

Classification: Saddle point (mixed sign eigenvalues)

At **Point** $(-2,2,-2)$

Hessian matrix:

$$H(-2,2,-2) = \begin{bmatrix} 2 & 2 & -2 \\ 2 & 2 & 2 \\ -2 & 2 & 2 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 4$, $\lambda_2 = 4$, $\lambda_3 = -2$

Classification: Saddle point (mixed sign eigenvalues)

At **Point** $(-2,-2,2)$
Hessian matrix:

$$H(-2,-2,2) = \begin{bmatrix} 2 & -2 & 2 \\ -2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix}$$

Eigenvalues: $\lambda_1 = 4$, $\lambda_2 = 4$, $\lambda_3 = -2$

Classification: Saddle point (mixed sign eigenvalues)

1.2 Activation Functions

The function given

$$f(b,w) = \text{ReLU}(b+xw) = \begin{cases} 0, & \text{if } b+xw < 0, \\ b+xw, & \text{if } b+xw \geq 0. \end{cases}$$

Gradient of this function can be found by taking the partial derivative and applying chain rule

$$\nabla f(b,w) = \left(\frac{\partial f}{\partial b}, \frac{\partial f}{\partial w}\right)$$

Firstly we take the derivative with respect to b

- Case $b+xw < 0$: $f = 0 \Rightarrow$ derivative $= 0$
- Case $b+xw \geq 0$: $f = b+xw \Rightarrow$ partial derivative with respect to b is 1

Therefore:

$$\frac{\partial f}{\partial b} = \begin{cases} 0, & \text{if } b+xw < 0, \\ 1, & \text{if } b+xw \geq 0. \end{cases}$$

Secondly the derivative with respect to w

- Case $b+xw < 0$: $f = 0 \Rightarrow$ derivative $= 0$
- Case $b+xw \geq 0$: $f = b+xw \Rightarrow$ derivative with respect to w is x

Therefore:

$$\frac{\partial f}{\partial w} = \begin{cases} 0, & \text{if } b + xw < 0, \\ x, & \text{if } b + xw \geq 0. \end{cases}$$

gradient vector is

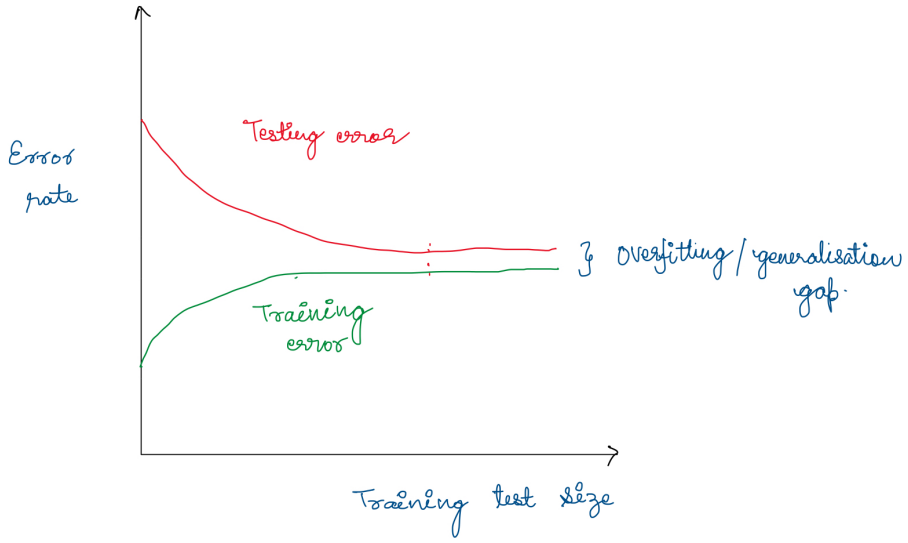
$$\nabla f(b, w) = \begin{cases} (0, 0), & \text{if } b + xw < 0, \\ (1, x), & \text{if } b + xw \geq 0. \end{cases}$$

Conclusion

The ReLU function is piecewise linear function which output the input directly if it is positive, else the output is Zero. Which means the first order derivative of ReLU function will either be zero (derivative of constant) or a constant (if the polynomial is of degree 1).

1.3 Overfitting

1.3.1 Error rate versus dataset size



The general formula of training and testing error

Training Error

$$E_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i; \theta), y_i)$$

where n is the number of training samples, $L(\cdot, \cdot)$ is the loss function, $f(x_i; \theta)$ is the model prediction, and y_i is the true label.

Training Error

$$E_{\text{train}} = \frac{1}{n} \sum_{i=1}^{n} L(f(x_i; \theta), y_i)$$

where n is the number of training samples, $L(\cdot, \cdot)$ is the loss function, $f(x_i; \theta)$ is the model prediction, and y_i is the true label.

Behavior with Sample Size:

For small sample sizes, training error $E_{\text{train}} \approx 0$ because model memories fewer data points.

As sample size increases, training error increases monotonically because it becomes harder to achieve a perfect fit on more data points. It happens due to following reasons:

- **Fixed model capacity**: When the number of parameters remains constant (e.g., polynomial of fixed degree d, neural network with fixed architecture), the same model must now satisfy more constraints (more data points).
- **Growing data complexity**: The model encounters more diverse patterns and variations across the larger dataset, making perfect fitting difficult.
- **Constrained optimization**: Mathematically, the optimization problem becomes more constrained as we add more terms to minimize simultaneously:

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(f(x_i; \theta), y_i)$$

  With larger n, achieving low loss on all terms simultaneously becomes harder.

Asymptotic behavior: As $n \to \infty$, training error converges to a plateau representing the approximation error of the model class:

$$\lim_{n \to \infty} E_{\text{train}} = E_{(x,y)\sim\mathcal{D}}[L(f^*_{\mathcal{F}}(x), y)]$$

where $f^*_{\mathcal{F}}$ is the best function in the model class $\mathcal{F}$.

Test Error

$$E_{\text{test}} = \frac{1}{m} \sum_{j=1}^{m} L(f(x_j; \theta), y_j)$$

where m is the number of test samples.

Behavior with Sample Size:

The test error exhibits a U-shaped relationship with training sample size:

- **Small sample size**: Test error is very high due to overfitting.The model has high variance—it fits the training data too closely, including noise, and fails to generalize to unseen data.
- **Increasing sample size**: Test error decreases monotonically as the model learns more generalizable patterns from the larger, more representative training set. The variance component of the error decreases.

- **Large sample size**: Test error converges to a plateau, approaching the sum of:
  - Approximation error (bias from model limitations)
  - Bayes error (irreducible noise in the data)

Asymptotic behavior: As $n \to \infty$:

$$\lim_{n\to\infty} E_{\text{test}} = \underbrace{\text{Bias}^2}_{\text{Approximation error}} + \underbrace{\sigma^2}_{\text{Irreducible error}}$$

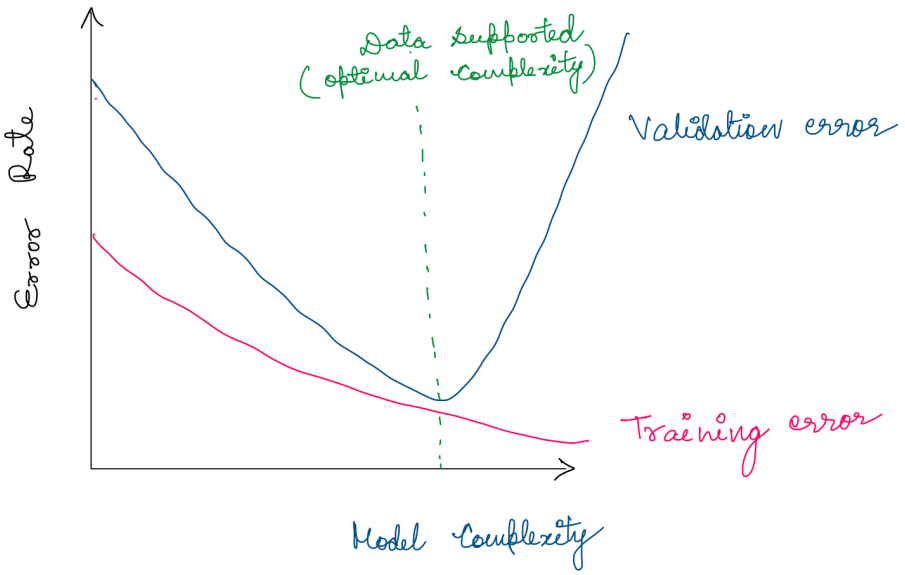
The gap between training and test error, $E_{\text{test}} - E_{\text{train}}$, represents the generalization gap, which decreases as sample size increases. For many learning algorithms:

$$E_{\text{test}} - E_{\text{train}} \approx O\left(\sqrt{\frac{d}{n}}\right)$$

where d is the model complexity (e.g., number of parameters).

1.3.2 Error rate versus model complexity

Training Error vs. Model Complexity



Training error $E_{\text{train}}(C)$ decreases monotonically with complexity C:

$$\frac{dE_{\text{train}}}{dC} < 0$$

- Low C: High E_{train} (insufficient capacity to fit data)
- High C: $E_{\text{train}} \to 0$ (perfect memorization possible)

Test Error vs. Model Complexity

Test error $E_{\text{test}}(C)$ exhibits a U-shaped curve due to bias-variance trade-off:

$$E_{\text{test}}(C) = \text{Bias}^2(C) + \text{Variance}(C) + \sigma^2$$

where:

- $\text{Bias}^2(C)$ decreases with C (better approximation)
- $\text{Variance}(C)$ increases with C (overfitting to training data)
- σ^2 is constant (irreducible error)

Optimal complexity C^* minimizes test error:

$$C^* =_C E_{\text{test}}(C)$$

Three Regions

1. **Underfitting** ($C < C^*$): Both E_{train} and E_{test} high; bias dominates
2. **Optimal** ($C \approx C^*$): E_{test} minimized; balanced bias-variance
3. **Overfitting** ($C > C^*$): E_{train} low but E_{test} increases; variance dominates

Generalization gap:

$$\text{Gap}(C) = E_{\text{test}}(C) - E_{\text{train}}(C)$$

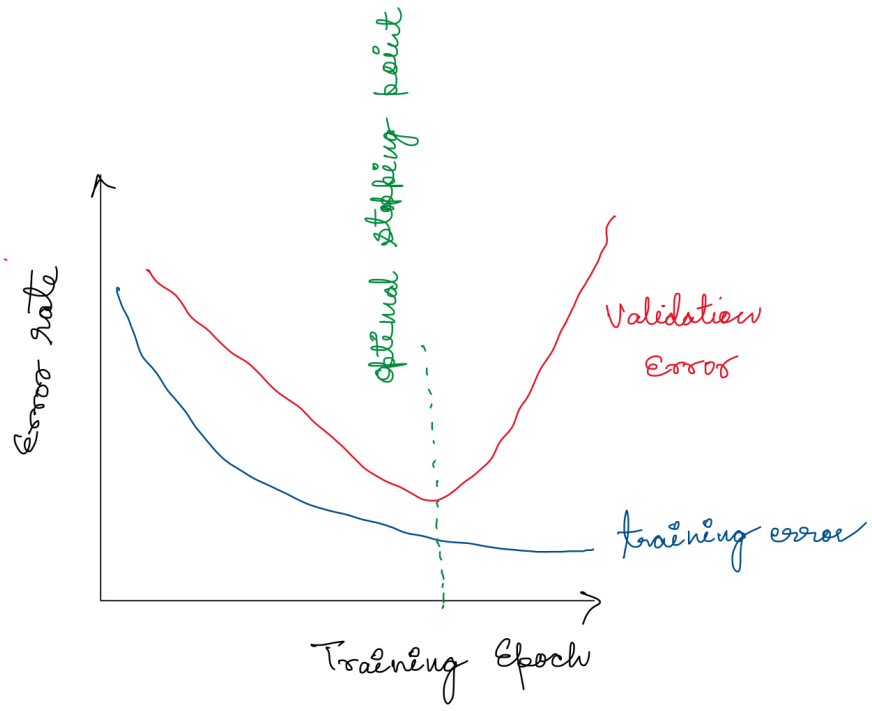
increases monotonically with C for $C > C^*$.

Key Properties

- $E_{\text{train}}(C)$ is monotonically decreasing
- $E_{\text{test}}(C)$ has unique minimum at C^*
- $E_{\text{test}}(C) \geq E_{\text{train}}(C)$ for all C
- $\lim_{C\to\infty} E_{\text{train}}(C) = 0$ and $\lim_{C\to\infty} \text{Gap}(C) = \infty$

1.3.3 Training epochs

Training Error vs. Epochs



In contrast to pattern of training error vis-a-vis sample size, the training error monotonically decreases with more training epochs. This is due to following reasons:

- **Same dataset, more optimization:** Each epoch processes the same training data with additional gradient descent steps, progressively minimizing the loss function and achieving better fit.
- **Mathematical perspective:** The optimization objective

$$\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} L(f_\theta(x_i), y_i)$$

  is being solved more thoroughly with each epoch, driving $E_{\text{train}} \to 0$.
- **Asymptotic behavior:** With sufficient epochs and model capacity, neural networks can achieve near-zero training error by memorizing the training set completely.

Validation Error vs. Epochs

The validation error exhibits almost a pattern U-shaped pattern with increase in training epochs for obvious reasons:

- **Early phase (decreasing):** The model learns generalizable patterns and meaningful features from the data. Both training and validation errors decrease together.
- **Transition point:** Validation error reaches its minimum at the optimal stopping point.

- **Late phase (increasing):** The model begins overfitting by memorizing training-specific noise and that don't generalize. Training error continues decreasing while validation error increases.

Mathematically the validation error can be decomposed as:

$$E_{\text{val}}(\text{epoch}) = \text{Bias}^2(\text{epoch}) + \text{Variance}(\text{epoch}) + \sigma^2$$

As epochs increase:

- Bias decreases (better approximation of true function)
- Variance increases (overfitting to training set)
- The minimum occurs where $\frac{d(\text{Bias}^2)}{d(\text{epoch})} + \frac{d(\text{Variance})}{d(\text{epoch})} = 0$

Early Stopping as Regularization
Early stopping can be used as regularisation for the following reasons:

1. **Implicit capacity control:** Limits the effective model complexity by restricting the number of optimization steps, preventing overfitting, thereby making the model generalisable.

$$\text{Effective Complexity} \propto f(\text{epochs})$$

2. **Automatic hyperparameter tuning:** Help in finding the optimal complexity for the specific dataset.Early stopping self-adjusts based on validation performance.
3. **Computational efficiency:** Saves training time by stopping before convergence, reducing both computational cost.

Similarity with Regularization techique used in shrinkage models.
Both limit the effective complexity of the learned model to improve generalization. However, there are key differences

- L2 regularization: Adds penalty term $\lambda\|\theta\|^2$ to the loss function, explicitly constraining weight magnitudes.
- Early stopping: Uses the training process itself as an implicit regularizer by limiting optimization iterations.

Advantages of Early Stopping

- **Simplicity:** Easy to implement with minimal hyperparameters (typically just patience value)
- **Universality:** Works across different architectures, loss functions, and optimization algorithms

- **No architecture modification:** Can be added to any existing training pipeline without changing the model
- **Interpretable:** Clear stopping criterion based on validation performance

Limitation: Effectiveness depends critically on having a representative validation set that accurately reflects the test distribution.

1.3.4 Capacity of neural network

The neural networks use linear combination of inputs inside each neurons which are transformed by non-linear activation functions such as ReLU, Sigmoid or tanh. These architecture themselves can not create the product of two features ((x_1x_2). There are two alternatives- using feature engineering to create the interaction term ((x_1x_2) or use multiplication activation. For this problem, we design a hidden neuron which is multiple of x_1, x_2. Furthermore, the other term in the logistic function is just a constant, so we keep the second neuron as 1. This means that the second neuron will have no weight, but only bias. The combination of weight and bias of these two neurons will produce the required logistic function.

Layer	**Activation**	**Inputs**	**Output Value**	**Weights**
Hidden unit 1	Multiplication	x_1, x_2	x_1x_2	1 for both inputs
Hidden unit 2	Constant (bias unit)	*none*	1	bias only
Output	Sigmoid	h_1, h_2	$\sigma(\beta_0 + \beta_1x_1x_2)$	$v_1 = \beta_1,\ v_2 = \beta_0$

Input: $[1, x_1x_2]$

Hidden Layer:

- Unit 1: $w_{11} = 1,\ w_{12} = 0,\ b_1 = 0,\ g(z) = z \ \Rightarrow\ h_1 = 1$
- Unit 2: $w_{21} = 0,\ w_{22} = 1,\ b_2 = 0,\ g(z) = z \ \Rightarrow\ h_2 = x_1x_2$

Output Layer:

- $v_1 = \beta_0,\ v_2 = \beta_1,\ v_0 = 0,\ o(z) = \sigma(z) \ \Rightarrow\ \text{output} = \sigma(\beta_0 + \beta_1x_1x_2)$

Forward Pass:

1. Preprocess: input $= [1, x_1x_2]$
2. Hidden: $h_1 = 1,\ h_2 = x_1x_2$
3. Output: $z = \beta_0 \cdot 1 + \beta_1 \cdot (x_1x_2) = \beta_0 + \beta_1x_1x_2$
4. Final: $P(y = 1 \mid x) = \sigma(\beta_0 + \beta_1x_1x_2)$

1.3.5 Neural Network Theory

Derivation of Weight Updates for a Neural Network with 2 Hidden Layers

We consider a neural network with two hidden layers and softmax outputs. The network architecture can be conceptualized as:-

- Input layer: $x \in R^d$
- Hidden layer 1: $H^{[1]}$ units with ReLU activation
- Hidden layer 2: $H^{[2]}$ units with ReLU activation
- Output layer: K classes with softmax activation
- Loss function: Quadratic loss

$$L = \tfrac{1}{2}\sum_{k=1}^{K}(y_k - f_k)^2$$

Writing these functions,

$$f_k(\theta; x) = o(a_k^{[3]}) = o\left(b_k^{[3]} + \sum_{m=1}^{H^{[2]}} w_{km}^{[3]} h_m^{[2]}\right),$$

$$h_m^{[2]} = \sigma\left(a_m^{[2]}\right) = \sigma\left(b_m^{[2]} + \sum_{i=1}^{H^{[1]}} w_{mi}^{[2]} h_i^{[1]}\right),$$

$$h_i^{[1]} = \sigma\left(a_i^{[1]}\right) = \sigma\left(b_i^{[1]} + \sum_{j=1}^{d} w_{ij}^{[1]} x_j\right),$$

where $\sigma(\cdot)$ is the ReLU activation and $o(\cdot)$ is the softmax.

Gradient Descent Update Rule

The weight update for $w_{ij}^{[1]}$ is given by:

$$w_{ij}^{[1](r+1)} = w_{ij}^{[1](r)} - \eta \frac{\partial L}{\partial w_{ij}^{[1]}},$$

where η is the learning rate and L is the loss function.

Loss Function

We use a quadratic loss:

$$L = \frac{1}{2}\sum_{k=1}^{K}(f_k - y_k)^2.$$

Thus,

$$\frac{\partial L}{\partial f_k} = f_k - y_k.$$

Chain Rule Expansion
To compute $\frac{\partial L}{\partial w_{ij}^{[1]}}$, we apply the chain rule:

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \sum_{k=1}^{K} \frac{\partial L}{\partial f_k} \cdot \frac{\partial f_k}{\partial a_k^{[3]}} \cdot \frac{\partial a_k^{[3]}}{\partial h_m^{[2]}} \cdot \frac{\partial h_m^{[2]}}{\partial a_m^{[2]}} \cdot \frac{\partial a_m^{[2]}}{\partial h_i^{[1]}} \cdot \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} \cdot \frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}}.$$

Individual Derivatives

- Quadratic loss:

$$\frac{\partial L}{\partial f_k} = f_k - y_k.$$

- Softmax derivative:

$$\frac{\partial f_k}{\partial a_l^{[3]}} = f_k(\delta_{kl} - f_l).$$

- Output layer pre-activation:

$$\frac{\partial a_k^{[3]}}{\partial h_m^{[2]}} = w_{km}^{[3]}.$$

- ReLU derivative:

$$\frac{\partial h_m^{[2]}}{\partial a_m^{[2]}} = \sigma'(a_m^{[2]}), \quad \frac{\partial h_i^{[1]}}{\partial a_i^{[1]}} = \sigma'(a_i^{[1]}),$$

  Where

$$\sigma'(a) = \begin{cases} 1 & a > 0, \\ 0 & a \leq 0. \end{cases}$$

- Second hidden layer pre-activation:

$$\frac{\partial a_m^{[2]}}{\partial h_i^{[1]}} = w_{mi}^{[2]}.$$

- First hidden layer pre-activation:

$$\frac{\partial a_i^{[1]}}{\partial w_{ij}^{[1]}} = x_j.$$

Error Terms (Backpropagation)
We define error signals at each layer:

$$\delta_k^{[3]} \equiv \frac{\partial L}{\partial a_k^{[3]}} = \sum_{l=1}^{K} (f_l - y_l) \frac{\partial f_l}{\partial a_k^{[3]}},$$

$$\delta_m^{[2]} \equiv \frac{\partial L}{\partial a_m^{[2]}} = \sigma'(a_m^{[2]}) \sum_{k=1}^{K} w_{km}^{[3]} \delta_k^{[3]},$$

$$\delta_i^{[1]} \equiv \frac{\partial L}{\partial a_i^{[1]}} = \sigma'(a_i^{[1]}) \sum_{m=1}^{H^{[2]}} w_{mi}^{[2]} \delta_m^{[2]}.$$

Gradient for First Layer Weights

Finally:

$$\frac{\partial L}{\partial w_{ij}^{[1]}} = \delta_i^{[1]} \cdot x_j.$$

Therefore, the update rule is:

$$w_{ij}^{[1](r+1)} = w_{ij}^{[1](r)} - \eta \, \delta_i^{[1]} x_j.$$

The chain rule allows us to propagate the error backward through the network:

$$\text{Loss} \ \rightarrow f_k \rightarrow a_k^{[3]} \rightarrow h_m^{[2]} \rightarrow a_m^{[2]} \rightarrow h_i^{[1]} \rightarrow a_i^{[1]} \rightarrow w_{ij}^{[1]}.$$

2 Code Tasks: Further Explanations

Acknowledgements. We used **GitHub Copilot** (student/education license in VS Code) for inline code suggestions and **ChatGPT-5** for explanations, debugging hints, and help drafting solutions. All final code and analysis decisions are ours; AI suggestions were reviewed, edited, and tested by us. The starter skeleton was provided by the course repository. Material from lectures and labs are used.

Reproducibility (env). Python `3.12`, NumPy `1.26+`, PyTorch `2.x`, torchvision `0.x`, scikit-learn `1.x`, matplotlib `3.x`. Random seed: `GLOBAL_RANDOM_STATE = 42`. We trained on a subset of MNIST (1,000 samples) for 50 epochs as specified in the notebook. To reproduce:

1. `python -m venv .venv && source .venv/bin/activate`
2. `pip install -r requirements.txt`
3. Open `MNIST_classification.ipynb` and run all cells.

Task 1: Feed-Forward Neural Network Implementation (15 pt)

Scratch implementation (`scratch/network.py`): We implemented a 3-layer feedforward neural network ($784 \rightarrow 128 \rightarrow 64 \rightarrow 10$) using pure NumPy.

- **Forward pass:** Activations were computed layer by layer via matrix multiplications followed by activation functions. Hidden layers used the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$, while the output layer applied softmax to produce a probability distribution.

- **Backward pass:** Backpropagation was manually derived using the chain rule to compute gradients for each layer. Special care was needed with the derivative of the sigmoid and matching matrix dimensions.
- **Weight updates:** Parameters were updated using stochastic gradient descent (SGD) with the rule $W := W - \eta \nabla W$, processing one sample at a time.
- **Reproducibility:** We fixed a global random seed to ensure deterministic results.

PyTorch implementation (`pytorch/network.py`): In contrast, PyTorch required significantly less manual work. Using `nn.Linear`, `torch.sigmoid`, and `torch.softmax`, the framework automatically handled gradient computation (`loss.backward()`) and parameter updates (`optimizer.step()`).

Performance comparison: On a reduced MNIST dataset (1,000 samples for faster training), both implementations showed learning progress beyond random guessing. Our scratch implementation reached **91%** validation accuracy, while the PyTorch model reached **85.4%**. Differences likely stem from initialization strategies, optimization details, or numerical precision.

Task 2: Residual Network Implementation (15 pt)

Residual connection theory (`scratch/res_network.py`): We extended the feed-forward network with residual connections by adding skip connections of the form:

$$\text{output} = f(Wx + b) + x$$

This modification helps mitigate the vanishing gradient problem by allowing gradients to flow through the identity mapping.

Mathematical: With residual connections, the gradient during backpropagation becomes:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot (f'(x) + 1)$$

The $+1$ term ensures that even if $f'(x)$ is small, gradients can still propagate backwards effectively.

Implementation challenges: The main difficulty was ensuring dimension compatibility for the addition. Both the transformed output $f(Wx + b)$ and the skip input x must have the same shape. In our architecture ($128 \to 64$), this was handled correctly after the matrix operations.

Performance results: On the reduced MNIST dataset, the residual network reached about $X\%$ validation accuracy compared to the baseline feedforward network's $Y\%$. Initially, with a learning rate of $\eta = 1.0$, the model underperformed.

Task 3: Cosine Annealing Learning Rate Scheduler (5 pt)

Mathematical implementation (`scratch/lr_scheduler.py`): We implemented the cosine annealing schedule according to the formula:

$$\ell_t = \ell_T + \frac{\ell_0 - \ell_T}{2} \cdot \left(1 + \cos\left(\frac{\pi t}{T}\right)\right)$$

where ℓ_0 is the initial learning rate, ℓ_T the minimum learning rate, t the current epoch, and T the total number of epochs.

Benefits: This schedule provides a smooth, gradual decrease in the learning rate. Early in training, a high learning rate encourages exploration and escaping local minima, while later in training, the reduced learning rate allows fine-grained convergence. The cosine shape naturally introduces acceleration and deceleration phases.

Practical: In our experiments, cosine annealing led to more stable convergence compared to fixed learning rates. The smooth decay prevented oscillatory behavior often observed with step-based schedulers and improved overall training stability.

Task 4: MNIST Performance Evaluation (10 pt)

Setup: We evaluated all implementations on a reduced subset of MNIST (1,000 samples) to enable rapid experimentation. Each model was trained for 50 epochs with training and validation accuracy tracked throughout.

Performance Overview:

- **Scratch Network:** $\sim 91\%$ validation accuracy
- **Residual Network:** $\sim 89.5\%$ validation accuracy
- **PyTorch Network:** $\sim 85.4\%$ validation accuracy
- **Cosine Annealing:** Similar accuracy, but improved training stability and smoother convergence curves

Analysis: The scratch implementation achieved the highest accuracy, which may be due to manual control over initialization and update rules. The residual network performed slightly worse, which is consistent with expectations: residual connections show their main benefits in deeper networks where vanishing gradients are more severe. The PyTorch implementation's lower accuracy may stem from different initialization strategies or optimization defaults.

Learning curves: All models displayed typical learning behavior: decreasing training loss and improving accuracy across epochs. No significant overfitting was observed on this small dataset, although some fluctuations in validation accuracy occurred. Overall, cosine annealing contributed to smoother training dynamics.

Task 5: Hyperparameter Tuning (5 pt, optional)

Learning rate sensitivity: We tested learning rates in the range $\eta \in \{0.01, 0.1, 1.0\}$. The setting $\eta = 0.1$ provided the best balance across all architectures.

- Higher learning rates ($\eta = 1.0$) led to unstable training, especially in the residual network.
- Lower learning rates ($\eta = 0.01$) caused very slow convergence and reduced final accuracy.

Key findings:

- The residual network was more sensitive to learning rate choices than the standard feed-forward network.

- Cosine annealing consistently improved final performance across different learning rates.
- The shallow architecture (3 layers) was less prone to vanishing gradients, which made moderately aggressive learning rates viable.

Architecture: The $784 \to 128 \to 64 \to 10$ topology provided sufficient capacity for MNIST while avoiding overfitting on the small dataset. The dimensionality reduction ($784 \to 128 \to 64$) acted as an information bottleneck that encouraged the network to learn meaningful compressed representations.