Total time: 90 mins Total marks: 18 Name and Roll Number: _____

Questions

1. (1 point) In principle, gradients can be estimated numerically via finite differences:

$$\frac{\partial J}{\partial \theta_i} \approx \frac{J(\theta_i + \epsilon) - J(\theta_i - \epsilon)}{2\epsilon}.$$

However, modern deep learning frameworks instead use *automatic differentiation (autograd)*.

List and briefly explain **two reasons** why finite differences are not preferred in modern neural networks.

Solution:

- **Computational cost:** Finite differences require at least two forward passes per parameter. For millions of parameters, this becomes prohibitively slow. Autograd computes all gradients in roughly the same time as a few forward passes (via backprop).
- **Numerical instability:** Choice of ϵ affects accuracy — too large ⇒ poor approximation, too small ⇒ floating-point cancellation errors. Autograd yields exact (up to machine precision) derivatives via the chain rule.

2. (3 points) Second-order update in Linear Regression. Consider the mean squared loss for linear regression:

$$J(\theta) = \frac{1}{2N}\|\mathbf{y} - X\theta\|_2^2,$$

where $J(\theta)$ is the objective function, $\nabla J(\theta)$ denotes its **gradient vector**, and $H = \nabla_\theta^2 J(\theta)$ denotes its **Hessian matrix** (matrix of second partial derivatives).

Suppose we perform a single **second-order (Newton)** update:

$$\theta_{\text{new}} = \theta_{\text{old}} - H^{-1}\nabla J(\theta_{\text{old}}).$$

(a) Compute $\nabla J(\theta)$ and H for this loss.

(b) Show what θ_{new} becomes after one Newton step (simplify fully).

(c) Explain why this result is interesting: what happens regardless of the initial θ_{old}? Why doesn't this generalize to deep networks?

Solution:

(a)

$$\nabla J(\theta) = -\frac{1}{N}X^\top(\mathbf{y} - X\theta), \qquad H = \frac{1}{N}X^\top X.$$

(b) Substitute into the Newton update:

$$\theta_{\text{new}} = \theta_{\text{old}} - (X^\top X)^{-1}\left(-X^\top(\mathbf{y} - X\theta_{\text{old}})\right) = (X^\top X)^{-1}X^\top\mathbf{y}.$$

Thus, one Newton step lands exactly at the **normal equation** solution — in a single step.

(c) This happens because the loss is a convex quadratic, and H is constant. In deep networks, H depends on θ and the loss is non-quadratic, so Newton's method no longer reaches the optimum in one step.

3. (4 points) Here is a canonical example of PyTorch training and testing loop

```
import torch
import torch.nn as nn
import torch.optim as optim

class SimpleModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(2, 3)
        self.fc2 = nn.Linear(3, 1)
    def forward(self, x):
        return self.fc2(self.fc1(x))
```

```
model = SimpleModel()
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

# XOR dataset
inputs = torch.tensor([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
targets = torch.tensor([[0.],[1.],[1.],[0.]])

for epoch in range(200):
    outputs = model(inputs)
    loss = criterion(outputs, targets)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {loss.item()}")
```

(a) What does `optimizer.zero_grad()` do, and why is it necessary before each backward pass?

(b) Explain the roles of `loss.backward()` and `optimizer.step()` in this code.

(c) How many trainable parameters does this model have in total?

(d) The dataset is XOR. Do you expect this model to learn it? Give a short mathematical or geometric justification.

Solution:

(a) PyTorch accumulates gradients by default. `optimizer.zero_grad()` clears previous gradients before computing new ones. Without it, gradients would add up across epochs, corrupting updates.

(b) `loss.backward()` computes $\nabla_\theta J(\theta)$ for all parameters using autograd. `optimizer.step()` updates each parameter based on these gradients (e.g. SGD rule).

(c)
$$\text{Layer 1: } 2\times 3+3=9, \quad \text{Layer 2: } 3\times 1+1=4, \quad \text{Total: 13 parameters.}$$

(d) No — the model is still *linear overall*. Composition of two linear maps is linear:
$$f(x)=W_2(W_1x+b_1)+b_2=(W_2W_1)x+(W_2b_1+b_2).$$
Hence, the decision boundary is linear, and XOR is not linearly separable.

(e) Adding a nonlinearity (e.g. ReLU, sigmoid) between layers breaks this linear composition. This enables the model to learn nonlinear decision boundaries and correctly separate XOR.

4. (3 points) Backpropagation through a simple tanh network. We define a scalar computation:
$$z=wx+b, \quad h=\tanh(z), \quad \hat{y}=h, \quad L=\tfrac{1}{2}(y-\hat{y})^2.$$
Assume $x=2,\ y=1,\ w=0.5,\ b=0$.

(a) Prove that
$$\frac{d}{dz}\tanh(z)=1-\tanh^2(z).$$
Hint: Start from $\tanh(z)=\dfrac{e^z-e^{-z}}{e^z+e^{-z}}$ and use quotient rule.

(b) Draw the computational graph.

(c) Using (a) and the *local* $\times$ *upstream* rule, compute $\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$. Show intermediate local gradients and numerical values.

Solution:

(a) $\tanh'(z)=\frac{(e^z+e^{-z})^2-(e^z-e^{-z})^2}{(e^z+e^{-z})^2}=\frac{4}{(e^z+e^{-z})^2}=1-\left(\frac{e^z-e^{-z}}{e^z+e^{-z}}\right)^2=1-\tanh^2(z).$

(b) **Forward:** $z=wx+b=1.0,\ h=\tanh(1)\approx 0.7616,\ L=\frac{1}{2}(1-0.7616)^2\approx 0.0284.$

Locals: $\frac{\partial L}{\partial \hat{y}}=-(y-\hat{y})=-0.2384,\ \frac{\partial \hat{y}}{\partial h}=1,\ \frac{\partial h}{\partial z}=1-\tanh^2(z)=1-0.7616^2=0.4199,$
$\frac{\partial z}{\partial w}=x=2,\ \frac{\partial z}{\partial b}=1.$

Chain rule:

$$\frac{\partial L}{\partial w} = (-0.2384)(1)(0.4199)(2) \approx -0.200, \qquad \frac{\partial L}{\partial b} = (-0.2384)(1)(0.4199)(1) \approx -0.100.$$

Note: Gradients are smaller due to partial saturation of tanh at $z = 1$.

5. (4 points) Softmax temperature and control of diversity in next-token prediction. In next-token prediction, the model produces logits $\mathbf{z} = [z_1, \ldots, z_V]$ over a vocabulary. To convert logits into probabilities, we use the **temperature-scaled softmax**:

$$p_i(T) = \frac{e^{z_i/T}}{\sum_j e^{z_j/T}},$$

where $T > 0$ is called the **temperature**. Many large language models (including OpenAI's) expose this as a user setting `temperature` to influence the "creativity" of generated text.

Consider the vocabulary $\{a, b, c, d, e\}$ with logits:

$$\mathbf{z} = [3,\ 2,\ 1,\ 0,\ -1].$$

(a) Compute the probabilities $p_i(T)$ for $T = 1$. Which token is most likely to be generated?

(b) If we sample 10 tokens independently from this distribution, approximately how many times do we expect each token $\{a, b, c, d, e\}$ to appear?

(c) Compute $p_i(T)$ again for $T = 2$. Compare the results and describe what changed.

(d) Predict qualitatively what happens when $T \to 0$ and when $T \to \infty$. Which regime produces deterministic outputs, and which produces exploratory ones? Explain in terms of how the softmax scales the logits.

Solution:

(a) For $T = 1$:

$$e^{[3,2,1,0,-1]} = [20.09,\ 7.39,\ 2.72,\ 1.00,\ 0.37], \quad Z = 31.57,$$
$$p = [0.637,\ 0.234,\ 0.086,\ 0.032,\ 0.012].$$

Most likely token: **a**.

(b) Expected counts (out of 10):

$$a : 6.4, \quad b : 2.3, \quad c : 0.9, \quad d : 0.3, \quad e : 0.1.$$

(c) For $T = 2$:

$$e^{[1.5,1,0.5,0,-0.5]} = [4.48, 2.72, 1.65, 1.00, 0.61], \quad Z = 10.46,$$
$$p(T = 2) = [0.428,\ 0.260,\ 0.158,\ 0.096,\ 0.058].$$

Distribution is *flatter*: lower top probability, higher tail probabilities.

(d) - As $T \to 0$, logits are divided by a very small number, amplifying their differences; softmax becomes nearly one-hot $\Rightarrow$ **deterministic** outputs. - As $T \to \infty$, all logits shrink toward zero, giving nearly uniform probabilities $\Rightarrow$ **exploratory/random** outputs.

(e) The `temperature` parameter rescales model confidence: low T favors precise, predictable completions (high accuracy), while high T promotes diversity and creativity by sampling from a wider range of tokens. It directly controls the trade-off between **stability and variety**.

6. (1 point) Compare the gradient behaviour of the sigmoid and ReLU activations. What is the **maximum gradient magnitude** achievable for sigmoid, and what is the **typical gradient value** for ReLU in its active region? Explain why sigmoid networks generally learn more slowly.

Solution: Sigmoid's gradient peaks at about 0.25 (when its output is 0.5), but becomes nearly 0 for large positive or negative inputs due to saturation. ReLU's gradient in its active region ($z > 0$) is typically 1, so it propagates gradients more effectively. Sigmoid networks thus suffer from vanishing gradients and slower learning.

7. (1 point) Scikit-learn provides the method `predict_proba()` for most classifiers to output class probabilities. In logistic regression, these probabilities come directly from the sigmoid or softmax model. How are such probabilities estimated in a Decision Tree or Random Forest?

Solution: Decision Trees estimate probabilities empirically — each leaf stores the proportion of training samples of each class that reach it. For a test point, the leaf's class fractions give the probabilities. Random Forests average these probabilities across all trees.

In a decision tree, each leaf node represents a subset of the training data that ended up there after the tree applied all its decision splits. Each leaf contains a distribution of class labels based on the training samples that reached it. `predict_proba()` returns this empirical class distribution, normalized so the probabilities sum to 1.

For example, if a leaf has 8 training samples, with 6 belonging to class 1 and 2 to class 0:

$$P(y = 1 \mid x) = \frac{6}{8} = 0.75, \quad P(y = 0 \mid x) = \frac{2}{8} = 0.25$$

Any new sample that lands in this leaf will get this probability vector:

$$\texttt{predict\_proba}(x) = [0.25, 0.75]$$

`predict_proba()` in a Random Forest

A Random Forest is an ensemble of many decision trees $T_1, T_2, \ldots, T_N$. Each tree predicts its own probability distribution over the classes, just like a single `DecisionTreeClassifier`. The Random Forest then averages these probabilities across all trees.

Formally, if the t^{th} tree predicts:

$$P_k^{(t)}(x) = P(y = k \mid x, T_t)$$

then the overall forest prediction is:

$$P(y = k \mid x) = \frac{1}{N} \sum_{t=1}^{N} P_k^{(t)}(x)$$

Example

Each decision tree in the forest assigns x to a leaf and computes its class probabilities.

The forest averages these:

$$P(y = 0 \mid x) = \frac{0.2 + 0.6 + 0.0}{3} = 0.2667, \quad P(y = 1 \mid x) = \frac{0.8 + 0.4 + 1.0}{3} = 0.7333$$

So the final output is:

$$\texttt{predict\_proba}(x) = [0.2667, 0.7333]$$

and

$$\texttt{predict}(x) = \text{class with higher probability, here class 1.}$$

Hard vs Soft Probabilities in Random Forests

There are two ways to compute probabilities in an ensemble of decision trees:

Hard probabilities (vote-based)

Each tree predicts a single class label (using `predict()`), and the proportion of trees voting for each class is used as the estimated probability:

$$P(y = k \mid x) = \frac{\text{Number of trees predicting class } k}{N}$$

This approach treats each tree's prediction as either 0 or 1 (a hard vote), ignoring the uncertainty within each tree.

Soft probabilities (average-based)

Instead of using class labels, we use each tree's `predict_proba` output and average them:

$$P(y = k \mid x) = \frac{1}{N} \sum_{t=1}^{N} P_k^{(t)}(x)$$

Here, each tree contributes a fractional belief for each class, allowing for more nuanced aggregation.

Why soft probabilities are better

- **Captures uncertainty:** Trees with mixed leaves (e.g., 60% class 1, 40% class 0) contribute proportionally, rather than voting decisively.
- **Smoother and more calibrated:** Averaging probabilistic outputs yields more stable and realistic probability estimates.
- **Hard and soft match only for pure leaves:** If all leaves are pure (each containing samples from only one class), both methods give identical results.

Hence, scikit-learn's `RandomForestClassifier.predict_proba()` uses **soft probabilities** by default, as they provide better-calibrated and more reliable estimates.

8. (1 point) We define a new metric M = Precision × Recall. For the classifier outputs below, decide which threshold $T \in \{0.2, 0.4, 0.6, 0.8\}$ gives the highest M.

S.No	$P(\hat{y} = 1 \mid x_i)$	True label (y_i)
1	0.25	0
2	0.95	1
3	0.10	0
4	0.70	1
5	0.40	0
6	0.55	1
7	0.30	0
8	0.85	1
9	0.60	0
10	0.05	0

Note that we predict positive if $P(\hat{y} = 1|x_i) \geq T$.

Solution: Compute Precision (P), Recall (R), and $M = P \times R$:

T	P	R	$M = P \times R$
0.2	0.50	1.00	0.50
0.4	0.67	1.00	0.67
0.6	0.75	0.75	0.56
0.8	1.00	0.50	0.50

$\boxed{T = 0.4}$ yields the highest $M = 0.67$.

Interpretation: At $T = 0.4$, the balance between precision and recall is optimal under this metric. Extremely low thresholds increase recall but hurt precision; very high thresholds do the opposite.