IIT BOMBAY

Indian Institute of Technology Bombay

CS 419 End Semester Examination

Date: November 21, 2024 **Max Marks: 50** **Duration: 180 minutes**

Instructions:

- Answer all questions.
- Show all work clearly, be as clear and precise as possible.
- The exam is open notes.

PART I : Perceptron Convergence Theorem

The **Perceptron Convergence Theorem** states that if the dataset is linearly separable, the Perceptron algorithm will find a separating hyperplane in a finite number of steps. In this question, we will see the proof of this theorem. The problem setup and notations are defined first.

Problem Setup

Assumptions

- The dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ is linearly separable
- $\mathbf{x}_i \in \mathbb{R}^d$ is the feature vector, and $y_i \in \{-1, 1\}$ is the label.
- There exists a weight vector $\mathbf{w}^*$ such that $y_i(\mathbf{w}^* \cdot \mathbf{x}_i) > 0 \quad \forall\, i$

Perceptron Update Rule

- Initialize $\mathbf{w} = \mathbf{0}$.
- For each misclassified point $(\mathbf{x}_i, y_i)$ when $y_i(\mathbf{w}_t \cdot \mathbf{x}_i) < 0$:

$$\mathbf{w_{t+1}} \leftarrow \mathbf{w_t} + y_i \mathbf{x}_i$$

Key Quantities

- Define γ as the **margin** of the dataset. Concretely,γ is given as

$$\gamma = \min_i \frac{y_i(\mathbf{w}^* \cdot \mathbf{x}_i)}{\|\mathbf{w}^*\|}$$

- The norm of feature vectors is bounded by M

$$M = \max_i \|\mathbf{x}_i\|$$

The Cauchy-Schwarz inequality

The Cauchy-Schwarz inequality states that for any vectors $\mathbf{u}$ and $\mathbf{v}$ in $\mathbb{R}^n$:

$$(\mathbf{u} \cdot \mathbf{v})^2 \leq \|\mathbf{u}\|^2 \|\mathbf{v}\|^2,$$

where, $\mathbf{u} \cdot \mathbf{v} = \sum_{i=1}^n u_i v_i$ is the dot product, and $\|\mathbf{u}\|$ is the euclidean norm.

Proof

Bounding the Increase in $\|\mathbf{w}\|$:

(i) When the Perceptron makes an update, we have:

$$\|\mathbf{w}_{t+1}\|^2 = \|\mathbf{w}_t + y_i\mathbf{x}_i\|^2 = \|\underline{\quad (a) \quad}\|^2 + 2y_i(\underline{\quad (b) \quad}) + \|\underline{\quad (c) \quad}\|^2$$

(ii) Since $\|y_i\mathbf{x}_i\|^2 = \|\mathbf{x}_i\|^2 \leq M^2$, we get:

$$\|\mathbf{w}_{t+1}\|^2 \leq \|\mathbf{w}_t\|^2 + \underline{\quad (d) \quad}$$

Increase in Alignment with $\mathbf{w}^*$:

(iii) The dot product with the target vector $\mathbf{w}^*$ increases by at least γ:

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^* = (\mathbf{w}_t + y_i\mathbf{x}_i) \cdot \mathbf{w}^* = \mathbf{w}_t \cdot \mathbf{w}^* + y_i(\mathbf{x}_i \cdot \mathbf{w}^*) \geq \mathbf{w}_t \cdot \mathbf{w}^* + \gamma$$

$$\mathbf{w}_{t+1} \cdot \mathbf{w}^* \geq \mathbf{w}_t \cdot \mathbf{w}^* + \underline{\quad (e) \quad}$$

Bounding the Number of Updates:

(iv) Let N be the number of updates made. Then:

$$\|\mathbf{w}_N\|^2 \leq \underline{\quad (f) \quad}, \qquad \mathbf{w}_N \cdot \mathbf{w}^* \geq \underline{\quad (g) \quad}$$

(v) Prove $N \leq \left(\frac{M\|\mathbf{w}^*\|}{\gamma}\right)^2$ (Apply Cauchy-Schwartz inequality and some more steps)

[**15 Marks:** (i): 1+1+1, (ii): 2, (iii): 3, (iv): 2+2, (v): 3]

PART II : Kernels

(A) Prove that the following are valid kernels by finding a transformation ϕ such that $K(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^T \phi(\mathbf{x}')$.

1. Let $K(\mathbf{x}, \mathbf{x}') = (1 + \mathbf{x}^T \mathbf{x}')^2$. **[2.5 marks]**
2. Let $K(\mathbf{x}, \mathbf{x}') = (\mathbf{x}^T \mathbf{x}')^2 + 3 \cdot (\mathbf{x}^T \mathbf{x}') + 2$. **[2.5 marks]**

PART III : Neural Networks

Implementing boolean functions with neural networks
Consider a binary logic function $f : \{0,1\}^2 \rightarrow \{0,1\}$. A neural network with a single hidden layer of two neurons and ReLU activation is used to approximate this function. Specifically, the function represented by the neural network is $f(x) = w_2(\sigma(w_1 x + b_1)) + b_2$, where $x = [x_1, x_2] \in \mathbb{R}^2$, $w_1 \in \mathbb{R}^{2\times 2}$, $w_2 \in \mathbb{R}^{1\times 2}$. The function $\sigma : \mathbb{R}^2 \rightarrow \mathbb{R}^2$ and is applied pointwise, meaning $\sigma([a, b]) = [\sigma(a), \sigma(b)]$. Please note that $\sigma(.)$ is the ReLU activation itself. Design w_1, w_2 and σ (the activation function) to compute **AND**, **XOR**, **OR** and **XNOR** functions. [**10 marks**]

Refer to the following truth tables for the exact definition of boolean functions: AND, OR, XOR and XNOR.

AND

x_1	x_2	AND(x_1, x_2)
0	0	0
0	1	0
1	0	0
1	1	1

OR

x_1	x_2	OR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x_1	x_2	XOR(x_1, x_2)
0	0	0
0	1	1
1	0	1
1	1	0

XNOR

x_1	x_2	XNOR(x_1, x_2)
0	0	1
0	1	0
1	0	0
1	1	1

PART IV : Logistic Regression

In this problem, we are tasked with performing multi-class classification with three possible classes, denoted as C_1, C_2, and C_3. There is always an option to use the softmax approach but some genius student in CS 419 course came up with the idea of extending the logistic regression to three classes using two sigmoids. Here is how they propose to model probabilities of the first two classes, C_1 and C_2, and derive the probability of the third class, C_3, based on these values.

Modeling Probabilities

The probabilities p_1 and p_2 are modeled using logistic sigmoid functions. Let x be the input feature vector, w_1 and w_2 be the weight vectors for classes C_1 and C_2, and b_1 and b_2 be the bias terms for these classes, respectively. The logistic sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

We can express p_1 and p_2 as follows:

$$p_1 = \sigma(w_1^T x + b_1) = \frac{1}{1 + e^{-(w_1^T x + b_1)}}$$

$$p_2 = \sigma(w_2^T x + b_2) = \frac{1}{1 + e^{-(w_2^T x + b_2)}}$$

The third class probability p_3 is derived as:

$$p_3 = 1 - p_1 - p_2$$

Assume that weight vectors w_1 and w_2 can be any general vectors in the R^d. Thus, the sum of the probabilities is always equal to 1, i.e., $p_1 + p_2 + p_3 = 1$.

1. This, in theory, seems like a great idea but there is an immediate problem. Point out the problem? [**2 mark**]

2. The values for p_1, p_2 and p_3 should satisfy an additional constraint if the above formulation is valid. What is that? [**3 mark**]

3. Given a dataset $\{(x_i, y_i) : i \in \{1, .., 100\}\}$, design a loss function to estimate w_1, w_2, which also encodes the constraint introduced in item 2. [**5 marks**]

PART V : Coding and PyTorch

Read the following part carefully before reading the code.

In the context of autoencoders, the model learns a transformation from the input x to a lower-dimensional representation z, and then reconstructs the input x' from z. The process can be described as:

$$x \xrightarrow{\text{Encoder}} z \xrightarrow{\text{Decoder}} x'$$

Here, $x \in \mathbb{R}^d$ is the original input, $z \in \mathbb{R}^{d'}$ is the compressed representation, where $d' < d$, and x' is the reconstructed output, ideally close to x. The network is trained to minimize the distance between x and x', ensuring that the learned representation captures the most important features of the input data.

In the code below, we aim to train an autoencoder using the distance minimization scheme given above. We provide an autoencoder implementation and the corresponding training and early stopping code. However, there are a number of bugs currently - due to dimension mismatch, error in early stopping and learning rate. Identify and report where the bugs are, and how to fix them. [**10 marks**]

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, TensorDataset, random_split

# Define the Autoencoder class
class Autoencoder(nn.Module):
    def __init__(self, input_dim, hidden_dims, latent_dim):
        super(Autoencoder, self).__init__()

        # Encoder
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dims[0]),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(hidden_dims[0], hidden_dims[1]),
            nn.ReLU(),
            nn.Linear(hidden_dims[1], latent_dim)
        )

        # Decoder
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dims[1]),
            nn.ReLU(),
            nn.Linear(hidden_dims[1], hidden_dims[0]),
            nn.ReLU(),
            nn.Linear(hidden_dims[0], input_dim),
            nn.Sigmoid()
        )

    def forward(self, x):
        # Forward pass
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return encoded, decoded

# Generate some dummy data # NO ERROR IN NEXT 2 LINES
torch.manual_seed(42)
data = get_training_data()  # 1000 samples, 20 features
train_size = int(0.8 * len(data))
es_size = len(data) - train_size
train_data, es_data = random_split(data, [train_size, es_size])
# Create DataLoader
batch_size = 32
train_loader = DataLoader(TensorDataset(train_data), batch_size=batch_size, shuffle=True)
es_loader = DataLoader(TensorDataset(es_data), batch_size=batch_size, shuffle=False)
```

```
# Initialize the model, loss function, and optimizer
input_dim = 32

# Two hidden layers with 128 and 64 units
hidden_dims = [128, 64]

# Latent representation size
latent_dim = 10

model = Autoencoder(input_dim=input_dim,
                    hidden_dims=hidden_dims,
                    latent_dim=latent_dim)

criterion = nn.MSELoss()  # Reconstruction loss
optimizer = torch.optim.Adam(model.parameters(), lr=1e+2)

# Training loop
epochs = 20
patience = 5
best_es_loss = float('inf')
best_model_state = None
no_improve_epochs = 0

for epoch in range(epochs):
    total_loss = 0
    for batch in train_loader:
        inputs = batch[0]   # NO ERROR IN THIS LINE

        # Forward pass
        output1, output2 = model(inputs)
        y_label = inputs   # NO ERROR IN THIS LINE
        y_pred = output2
        loss = criterion(y_label, y_pred)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        total_loss += loss.item()

    print(f"Epoch [{epoch+1}/{epochs}]")
    print(f"Loss: {total_loss/len(train_loader):.4f}")

    model.eval()
    es_loss = 0
    with torch.no_grad():
        for es_batch in train_loader:
            es_inputs = es_batch[0]
            _, es_outputs = model(es_inputs)
```

```
            es_loss += criterion(es_inputs, es_outputs).item()
    es_loss /= len(es_loader)

    print(f"Epoch [{epoch+1}/{epochs}]")
    print(f"Train Loss: {total_loss/len(train_loader):.4f}")
    print(f"Early Stopping Loss: {es_loss:.4f}")

    # Early stopping logic
    if es_loss < best_es_loss:
        best_es_loss = es_loss
        best_model_state = model.state_dict()
        no_improve_epochs = 0

    else:
        no_improve_epochs += 1
        if no_improve_epochs >= patience:
            print("Early stopping triggered.")
            break

# Load the best model state
if best_model_state:
    model.load_state_dict(best_model_state)

# Example: Get latent representations for the data
model.eval()
with torch.no_grad():
    latent, reconstructed = model(data)

print("Latent representation shape:", latent.shape)
```