Introduction to Machine Learning Project Phase 2 Report

Ali Sadeghian[1], Amirreza Tanevardi[2]

[1] Sharif University of Technology, Electrical Engineering, 400101464, ali.sadeghian@ee.sharif.edu
[2] Sharif University of Technology, Electrical Engineering, 400100898, amirrezatanevardi@gmail.com

Keywords:

Restricted Boltzmann Machines (RBMs) are undirected generative models that use a layer of hidden variables to model a distribution over visible variables. Typically, they are employed to model input distributions in classification tasks, but they can also capture the joint distribution of inputs and target classes. This characteristic makes them particularly useful in deep learning architectures, such as the last layer of a Deep Belief Network.

An RBM with n hidden units is a parametric model describing the joint distribution between hidden variables $h = (h_1, \ldots, h_n)$ and observed variables (x, y), where $x = (x_1, \ldots, x_d)$ represents input features and y denotes the target class. The joint probability distribution takes the form:

$$p(y, x, h) \propto e^{-E(y,x,h)}$$

where the energy function is given by:

$$E(y, x, h) = -h^T W x - b^T x - c^T h - d^T \tilde{y} - h^T U \tilde{y}$$

with parameters $\Theta = (W, b, c, d, U)$ and one-hot encoded class representation $\tilde{y} = (1_{y=i})_{i=1}^{C}$. The model is applicable to binary, integer-valued, and continuous-valued inputs, making it flexible across various applications.

Theoretical Question 1

Explain the ElGamal encryption algorithm completely and write its mathematical relationships.

Theoretical Question 1 answer

The **ElGamal encryption algorithm** is a public-key cryptosystem based on the *Discrete Logarithm Problem (DLP)*. It consists of three main steps: **key generation, encryption, and decryption**.

Key Generation

1. Choose a large prime number p.
2. Select a primitive root g of p.
3. Choose a private key x such that $1 \leq x \leq p - 2$.

4. Compute the public key y as:
$$y = g^x \mod p$$
5. The **public key** is (p, g, y), and the **private key** is x.

Encryption

To encrypt a message M:

1. Represent M as an integer such that $0 \leq M < p$.
2. Choose a random integer k such that $1 \leq k \leq p - 2$.
3. Compute the first part of the ciphertext:
$$C_1 = g^k \mod p$$
4. Compute the second part of the ciphertext:
$$C_2 = M \cdot y^k \mod p$$
5. The ciphertext is (C_1, C_2).

Theoretical Question 2

Explain partial decryption in detail.

Theoretical Question 2 answer

Partial decryption in ElGamal encryption refers to computing an intermediate decryption step without fully recovering the plaintext message. This step involves calculating the shared secret S.

Computing the Shared Secret

Given the ciphertext (C_1, C_2), the receiver computes:
$$S = C_1^x \mod p$$
Since $C_1 = g^k$, we get:
$$S = (g^k)^x \mod p = g^{kx} \mod p$$

Importance of Partial Decryption

- The shared secret S is a crucial value used in full decryption.
- Without knowing x, an attacker cannot compute S.
- Partial decryption stops at this step, leaving the final message recovery incomplete.

Theoretical Question 3

Provide an algorithm similar to the method seen above for computing the multiplication of two numbers. Assume that Party A knows the number M, and Party B knows the number N, and they want to compute MN.

Theoretical Question 3 answer

The following steps outline a secure multiplication protocol using ElGamal encryption:

Step 1: Initialization

- Party A generates a random number R.
- Party A computes:
$$m_i = M \cdot i - R$$
where i represents the possible input values of Party B.
- Each m_i is treated as plaintext and encrypted using the ElGamal encryption scheme:
$$E(m_i, r_i)$$
where r_i is a new random number for each encryption.
- Party A sends $E(m_i, r_i)$ to Party B in increasing order of i.

Step 2: Party B Picks and Rerandomizes

- Party B selects the ciphertext corresponding to its value N, i.e., $E(m_N, r_N)$.
- Party B rerandomizes the encryption by adding a new random value s:
$$E(m_N, r') = E(m_N, r_N + s)$$
- Party B sends $E(m_N, r')$ back to Party A.

Step 3: Partial Decryption by Party A

- Party A partially decrypts $E(m_N, r')$.
- The partially decrypted message is then sent to Party B.

Step 4: Final Decryption by Party B

- Party B fully decrypts the received message.
- The decrypted message is:
$$m_N = M \cdot N - R$$

- Since R is only known to Party A, and m_N is only known to Party B, the final result is computed as:

$$MN = m_N + R$$

Simulation Question 1

Using the MNIST dataset and the secure RBM as a feature extractor, perform the classification task and compare its performance against the regular RBM.

Simulation Question 1 answer

The code is presented part by part and the functionality of each section is explained. First we inspect the Regular RBM. here is the models code:

```python
class RBM:
    def __init__(self, input=None, n_visible=784, n_hidden=128, W=None, hbias=None, vbias=None, numpy_rng=None
        self.n_visible = n_visible
        self.n_hidden = n_hidden

        if numpy_rng is None:
            numpy_rng = np.random.RandomState(1234)

        if W is None:
            a = 1. / n_visible
            W = numpy_rng.uniform(-a, a, size=(n_visible, n_hidden))

        if hbias is None:
            hbias = np.zeros(n_hidden)

        if vbias is None:
            vbias = np.zeros(n_visible)

        self.W = W
        self.hbias = hbias
        self.vbias = vbias
        self.numpy_rng = numpy_rng

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sample_h_given_v(self, v):
        h_mean = self.sigmoid(np.dot(v, self.W) + self.hbias)
        h_sample = self.numpy_rng.binomial(size=h_mean.shape, n=1, p=h_mean)
        return h_mean, h_sample

    def sample_v_given_h(self, h):
        v_mean = self.sigmoid(np.dot(h, self.W.T) + self.vbias)
        v_sample = self.numpy_rng.binomial(size=v_mean.shape, n=1, p=v_mean)
        return v_mean, v_sample

    def gibbs_hvh(self, h0_sample):
        v_mean, v_sample = self.sample_v_given_h(h0_sample)
        h_mean, h_sample = self.sample_h_given_v(v_sample)
        return v_mean, v_sample, h_mean, h_sample

    def contrastive_divergence(self, data, k=1, lr=0.1):
        ph_mean, ph_sample = self.sample_h_given_v(data)

        chain_start = ph_sample
        for step in range(k):
            if step == 0:
                nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh(chain_start)
            else:
```

```
                nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh(nh_sample)

        self.W += lr * (np.dot(data.T, ph_sample) - np.dot(nv_sample.T, nh_mean)) / len(data)
        self.vbias += lr * np.mean(data - nv_sample, axis=0)
        self.hbias += lr * np.mean(ph_sample - nh_mean, axis=0)

    def reconstruct(self, v):
        h = self.sigmoid(np.dot(v, self.W) + self.hbias)
        reconstructed_v = self.sigmoid(np.dot(h, self.W.T) + self.vbias)
        return reconstructed_v

    def save_model(self, file_path):
        with open(file_path, 'wb') as f:
            pickle.dump(self, f)

    @staticmethod
    def load_model(file_path):
        with open(file_path, 'rb') as f:
            return pickle.load(f)
```

This is the exact same model we developed in the first phase of the project. The training function is as follows:

```

def train_rbm(train_loader, test_loader, n_visible, n_hidden, learning_rate, epochs, k):
      rbm = RBM(n_visible=n_visible, n_hidden=n_hidden)

      for epoch in range(epochs):
            for batch, _ in train_loader:
                batch = batch.view(-1, n_visible).numpy()
                rbm.contrastive_divergence(batch, k=k, lr=learning_rate)

            test_data = next(iter(test_loader))[0].view(-1, n_visible).numpy()
            reconstruction = rbm.reconstruct(test_data)
            loss = np.mean((test_data - reconstruction) ** 2)
            print(f"Epoch {epoch + 1}, k={k}: Reconstruction Loss = {loss:.4f}")

      return rbm
```

The explanation for these pieces of code was given before. The piece of code we wrote was this function that was used to extract features from the MNIST dataset using the RBM model. The features are considered to be the mean of the hidden layer after the first step of Gibbs sampling process.

```

        from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# Extract features using RBM
def extract_features_rbm(rbm, loader):
    features = []
    labels = []

    with torch.no_grad():
        for batch, lbls in loader:
            batch = batch.view(-1, rbm.n_visible)
            prob_h , _ = rbm.sample_h_given_v(batch)
            features.append(prob_h)
            labels.append(lbls)

    return features, labels

transform = transforms.Compose([transforms.ToTensor(), transforms.Lambda(lambda x: x.view(-1))])
train_dataset = datasets.MNIST(root="./data", train=True, transform=transform, download=True)
test_dataset = datasets.MNIST(root="./data", train=False, transform=transform, download=True)

train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True)
```

```
test_loader = DataLoader(test_dataset, batch_size=1, shuffle=False)
x_train_rbm, y_train_rbm = extract_features_rbm(regular_rbm, train_loader)
x_test_rbm, y_test_rbm = extract_features_rbm(regular_rbm, test_loader)

# Train a logistic regression model on RBM features
clf_rbm = LogisticRegression(max_iter=1000)
clf_rbm.fit(np.array(x_train_rbm).squeeze(1), np.array(y_train_rbm).squeeze(1))

# Evaluate the model
y_pred_rbm = clf_rbm.predict(np.array(x_test_rbm).squeeze(1))
accuracy_rbm = accuracy_score(np.array(y_test_rbm).squeeze(1), y_pred_rbm)
print(f"RBM Feature Classification Accuracy: {accuracy_rbm:.4f}")

```

As you can see, we used a simple Logistic Regression model as our classifier. Here is the result for this classification task:

Model	Precision	Recall	F1-score	accuracy
Regular RBM (CD=1)	0.93	0.93	0.93	0.94

Table 1: Performance of regular RBM

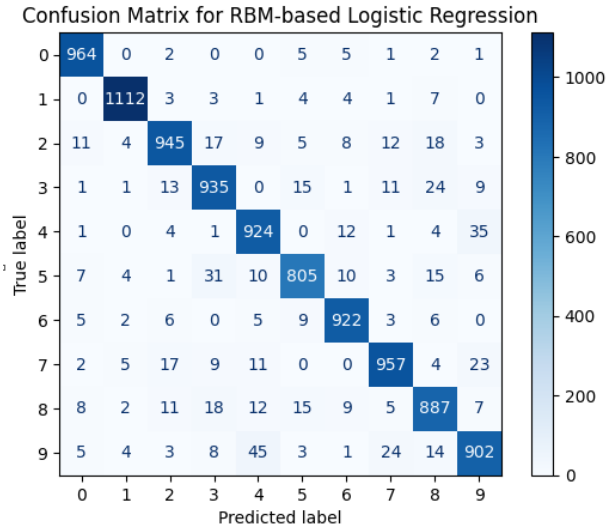

Figure 1: Confusion Matrix for Regular RBM

Now we examine the secure RBM. The model stayes the same for the most part. But the data gets split vertically before passing to the sampling process. The sampling is done via the secure method described in the project sheet.

```
class SecureRBM:
    def __init__(self, input=None, n_visible=784, n_hidden=128, W1=None, W2=None, hbias=None, vbias1=None, vbi
        self.n_visible = n_visible
        self.n_hidden = n_hidden

        if numpy_rng is None:
            numpy_rng = np.random.RandomState(1234)

        if W1 is None:
            a = 1. / n_visible
            W1 = numpy_rng.uniform(-a, a, size=(n_visible//2, n_hidden))

        if W2 is None:
```

```
            a = 1. / n_visible
            W2 = numpy_rng.uniform(-a, a, size=(n_visible//2, n_hidden))

        if hbias is None:
            hbias = np.zeros(n_hidden)

        if vbias1 is None:
            vbias = np.zeros(n_visible)


        self.W1 = W1
        self.W2 = W2
        self.hbias = hbias
        self.vbias = vbias
        self.numpy_rng = numpy_rng

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sample_h_given_vA(self, v):
        h_mean = np.dot(v, self.W1) + self.hbias
        #h_sample = self.numpy_rng.binomial(size=h_mean.shape, n=1, p=h_mean)
        return h_mean

    def sample_h_given_vB(self, v):
        h_mean = np.dot(v, self.W2) + self.hbias
        #h_sample = self.numpy_rng.binomial(size=h_mean.shape, n=1, p=h_mean)
        return h_mean

    def sample_v_given_hA(self, h):
        W = np.concatenate((self.W1 , self.W2))
        v_mean = np.dot(h, W.T) + self.vbias
        #v_sample = self.numpy_rng.binomial(size=v_mean.shape, n=1, p=v_mean)
        return v_mean

    def sample_v_given_hB(self, h):
        W = np.concatenate((self.W1 , self.W2))
        v_mean = np.dot(h, W.T) + self.vbias
        #v_sample = self.numpy_rng.binomial(size=v_mean.shape, n=1, p=v_mean)
        return v_mean


    def gibbs_hvh_secure(self, h0_1 , h0_2):
        v1_mean1 = self.sample_v_given_hA(h0_1)
        v1_mean2 = self.sample_v_given_hB(h0_1)

        v_1 = self.secure_sig(v1_mean1 , v1_mean2)

        h1_mean1= self.sample_h_given_vA(v_1[: , :self.n_visible//2] )
        h1_mean2= self.sample_h_given_vB(v_1[: , self.n_visible//2:] )

        h_1 = secure_sig(h1_mean1 , h1_mean2)

        return v_1, v_1, h_1, h_1


    def secure_sig(self, h1 ,h2):
        return self.sigmoid(h1+h2)

    def contrastive_divergence(self, data, k=1, lr=0.1):
        data_A = data[: , :self.n_visible//2]
        data_B = data[: , self.n_visible//2:]
        ph_mean1 = self.sample_h_given_vA(data_A)
        ph_mean2 = self.sample_h_given_vB(data_B)
        h0 = secure_sigmoid(ph_mean1, ph_mean2)

        nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh_secure(h0/2 , h0/2)



        chain_start = h0
        for step in range(k):
            if step == 0:
```

```
                nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh_secure(chain_start/2 , chain_start/2)
            else:
                nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh_secure(nh_sample/2 , nh_sample/2)


        self.W1 += lr * (np.dot(data_A.T, h0) - np.dot(nv_sample[: , :self.n_visible//2].T, nh_mean)) / len(da
        self.W2 += lr * (np.dot(data_B.T, h0) - np.dot(nv_sample[: , self.n_visible//2:].T, nh_mean)) / len(da


        self.vbias += lr * np.mean(data - nv_sample, axis=0)
        self.hbias += lr * np.mean(h0 - nh_mean, axis=0)

    def reconstruct(self, v):
        h = self.sigmoid(np.dot(v, self.W) + self.hbias)
        reconstructed_v = self.sigmoid(np.dot(h, self.W.T) + self.vbias)
        return reconstructed_v

    def save_model(self, file_path):
        with open(file_path, 'wb') as f:
            pickle.dump(self, f)

    @staticmethod
    def load_model(file_path):
        with open(file_path, 'rb') as f:
            return pickle.load(f)
```

The security operations are performed by these functions:

```
import torch
import random
from sympy import mod_inverse

# ElGamal key generation for tensors
def generate_keys(p, shape):
    """Generate ElGamal public and private keys for tensors."""
    g = torch.randint(2, p - 2, shape)  # Generator tensor
    x = torch.randint(1, p - 2, shape)  # Private key tensor
    y = torch.pow(g, x) % p  # Public key tensor
    return (p, g, y), x  # Public key: (p, g, y), Private key: x

# ElGamal encryption for tensors
def encrypt(public_key, message):
    p, g, y = public_key
    k = torch.randint(1, p - 2, message.shape)  # Random integer tensor k
    c1 = torch.pow(g, k) % p
    c2 = (message * torch.pow(y, k)) % p
    return c1, c2

# ElGamal decryption (partial and final) for tensors
def decrypt_partial(private_key, public_key, ciphertext):
    p, g, y = public_key
    c1, c2 = ciphertext
    s = torch.pow(c1, private_key) % p  # Shared secret tensor
    return c2, s  # Partially decrypted message with the shared secret

def decrypt_final(partially_decrypted, shared_secret, p):
    c2, s = partially_decrypted

    # Handle the case where s = 0 (modular inverse of 0 does not exist)
    s = torch.where(s == 0, torch.tensor(1, dtype=s.dtype), s)

    # Compute the modular inverse of s
    s_inv = torch.tensor([mod_inverse(int(val), p) for val in s.flatten()]).reshape(s.shape)

    # Decrypt the message
    message = (c2 * s_inv) % p
    return message

# Sigmoid function for tensors
def sigmoid(z):
    return 1 / (1 + torch.exp(-z))  # Sigmoid function for tensors

```

```python
# Function to generate possible_input as a grid of tensors
def generate_possible_input(y):
    """Generate the nearest tensor to y, where each element is rounded to the nearest 0.1."""
    # Round each element of y to the nearest 0.1
    possible_input = torch.round(y * 10) / 10
    return [possible_input]

# Secure Sigmoid Computation for tensors
def secure_sigmoid(x, y):
    """Securely compute sigmoid(x + y) using ElGamal encryption for tensors."""
    p = 467  # A prime number
    shape = x.shape  # Shape of the input tensors

    possible_input = generate_possible_input(y)

    # Step 1: Party A generates a random number R and computes mi = sigmoid(x + i) - R for i in possible_input
    #print(torch.max(y))
    R = torch.randint(1, int(torch.max(y)) + 2, shape)  # Random tensor R

    # Generate keys
    public_key, private_key = generate_keys(p, shape)

    encrypted_messages = []

    # Iterate over each tensor i in the set possible_input
    for i in possible_input:
        m_i = (sigmoid(x + i) * p - R).int()  # Scale sigmoid to match finite field
        encrypted_messages.append(encrypt(public_key, m_i))

    # Step 2: Party B selects the encrypted message corresponding to its input y
    selected_ciphertexts = encrypted_messages[0]  # Reshape to match tensor shape

    # Party B re-randomizes the selected ciphertexts
    random_s = torch.randint(1, p - 1, shape)
    randomized_c1 = (selected_ciphertexts[0] * torch.pow(public_key[1], random_s)) % p
    randomized_c2 = (selected_ciphertexts[1] * torch.pow(public_key[2], random_s)) % p
    randomized_encrypted = (randomized_c1, randomized_c2)

    # Step 3: Party A partially decrypts and sends it back
    partially_decrypted_c2, shared_secret_a = decrypt_partial(private_key, public_key, randomized_encrypted)

    # Step 4: Party B completes decryption
    final_message_b = decrypt_final((partially_decrypted_c2, shared_secret_a), random_s, p)

    # Party B computes final sigmoid value
    result = (final_message_b + R) / p  # Reverse scaling

    return torch.where(result > 1, torch.tensor(0.1), R/p), torch.where(result > 1, result - 1.1, final_messag

# Secure Multiplication Computation for tensors
def secure_multiplication(x, y):
    """Securely compute x * y using ElGamal encryption for tensors."""
    p = 467  # A prime number
    shape = torch.matmul(x,y).shape  # Shape of the input tensors

    possible_input = generate_possible_input(y)

    # Step 1: Party A generates a random number R and computes mi = (x * i) - R for i in possible_input
    R = torch.randint(1, int(torch.max(y)) + 2, shape)  # Random tensor R

    # Generate keys
    public_key, private_key = generate_keys(p, shape)

    encrypted_messages = []

    # Iterate over each tensor i in the set possible_input
    for i in possible_input:
        m_i = (torch.matmul(x,i) * 10 - R).int()
        #print(m_i.shape) # Scale multiplication to match finite field
        encrypted_messages.append(encrypt(public_key, m_i))

    # Step 2: Party B selects the encrypted message corresponding to its input y
    selected_ciphertexts = encrypted_messages[0]
```

```
    # Party B re-randomizes the selected ciphertexts
    random_s = torch.randint(1, p - 1, shape)
    randomized_c1 = (selected_ciphertexts[0] * torch.pow(public_key[1], random_s)) % p
    randomized_c2 = (selected_ciphertexts[1] * torch.pow(public_key[2], random_s)) % p
    randomized_encrypted = (randomized_c1, randomized_c2)

    # Step 3: Party A partially decrypts and sends it back
    partially_decrypted_c2, shared_secret_a = decrypt_partial(private_key, public_key, randomized_encrypted)

    # Step 4: Party B completes decryption
    final_message_b = decrypt_final((partially_decrypted_c2, shared_secret_a), random_s, p)

    # Party B computes final multiplication value
    result = (final_message_b + R) / 10  # Reverse scaling

    return torch.where(result > 1, torch.tensor(0.1), R/10), torch.where(result > 1, result - 1.1, final_messa
```

The model was only trained for 5 epochs due to the slower process of encryption and decryption. The Results are as follows:

Model	Precision	Recall	F1-score	accuracy
Secure RBM (CD=1)	0.85	0.85	0.85	0.85

Table 2: Performance of Secure RBM

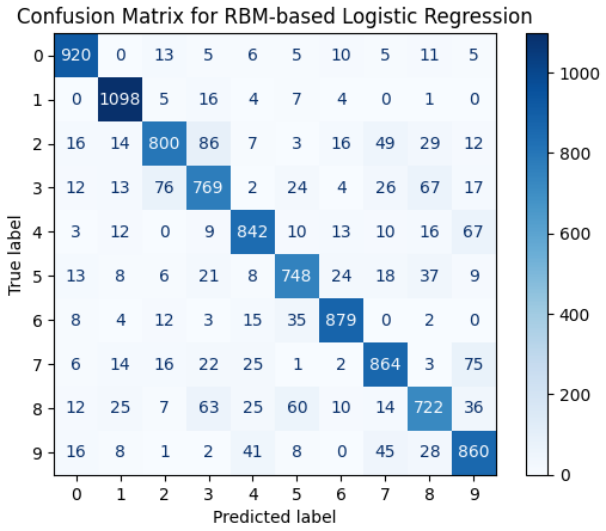

Figure 2: Confusion Matrix for Secure RBM

Theoretical Question 4 answer

1. Joint Distribution $p(y, x)$

The joint distribution $p(y, x, h)$ is defined as:

$$p(y, x, h) = \frac{e^{-E(y,x,h)}}{Z}$$

where Z is the partition function:

$$Z = \sum_{y,x,h} e^{-E(y,x,h)}$$

To obtain $p(y, x)$, we marginalize over h:

$$p(y, x) = \sum_{h} p(y, x, h) = \frac{1}{Z} \sum_{h} e^{-E(y,x,h)}$$

Using the energy function:

$$E(y, x, h) = -h^T W x - b^T x - c^T h - d^T \tilde{y} - h^T U \tilde{y}$$

we get:

$$p(y, x) = \frac{e^{b^T x + d^T \tilde{y}}}{Z} \sum_{h} e^{h^T (Wx + c + U\tilde{y})}$$

Since $h_j \in \{0, 1\}$, the sum simplifies to:

$$\sum_{h} e^{h^T (Wx + c + U\tilde{y})} = \prod_{j=1}^{n} \left(1 + e^{W_j x + c_j + U_j \tilde{y}}\right)$$

Thus, the joint distribution is:

$$p(y, x) = \frac{e^{b^T x + d^T \tilde{y}}}{Z} \prod_{j=1}^{n} \left(1 + e^{W_j x + c_j + U_j \tilde{y}}\right)$$

2. Conditional Distribution $p(y|x)$

The conditional probability follows:

$$p(y|x) = \frac{p(y, x)}{p(x)}$$

where:

$$p(x) = \sum_{y} p(y, x)$$

Substituting $p(y, x)$:

$$p(y|x) = \frac{e^{d^T \tilde{y}} \prod_{j=1}^{n} \left(1 + e^{W_j x + c_j + U_j \tilde{y}}\right)}{\sum_{y^*} e^{d^T \tilde{y}^*} \prod_{j=1}^{n} \left(1 + e^{W_j x + c_j + U_j \tilde{y}^*}\right)} \tag{1}$$

3. Log-likelihood for $p(y, x)$

The log-likelihood function is:

$$L_{\text{gen}}(D_{\text{train}}) = -\sum_{i=1}^{|D_{\text{train}}|} \log p(y_i, x_i)$$

The gradient is:

$$\frac{\partial \log p(y_i, x_i)}{\partial \theta} = -\mathbb{E}_{h|y_i,x_i}\left[\frac{\partial}{\partial \theta} E(y_i, x_i, h)\right] + \mathbb{E}_{y,x,h}\left[\frac{\partial}{\partial \theta} E(y, x, h)\right]$$

The second term is typically approximated using Contrastive Divergence (CD) or Persistent Contrastive Divergence (PCD).

4. Log-likelihood for $p(y|x)$

The conditional log-likelihood:

$$L_{\text{cond}}(D_{\text{train}}) = -\sum_{i=1}^{|D_{\text{train}}|} \log p(y_i|x_i)$$

Its gradient is:

$$\frac{\partial \log p(y_i|x_i)}{\partial \theta} = \frac{\partial}{\partial \theta} \log \left(\frac{e^{d_{y_i}} \prod_{j=1}^{n} \left(1 + e^{W_j x_i + c_j + U_j \tilde{y}_i}\right)}{\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{W_j x_i + c_j + U_j \tilde{y}^*}\right)} \right)$$

5. Free Energy Representation $F(x, y)$

The free energy function is:

$$F(x, y) = -\log \sum_{h} e^{-E(y,x,h)}$$

Using the energy function:

$$F(x, y) = -b^T x - d^T \tilde{y} - \sum_{j=1}^{n} \log \left(1 + e^{W_j x + c_j + U_j \tilde{y}}\right)$$

This representation is crucial for computing probabilities efficiently.

Theoretical Question 5

Prove the following equation:

$$\frac{\partial \log p(y_i|x_i)}{\partial \theta} = \sum_{j} \text{sigm}(o_{yj}(\mathbf{x_i})) \frac{\partial o_{yj}(\mathbf{x_i})}{\partial \theta} - \sum_{j,y^*} \text{sigm}(o_{y^*j}(\mathbf{x_i})) p(y^*|x_i) \frac{\partial o_{y^*j}(\mathbf{x_i})}{\partial \theta} \quad (2)$$

where

$$o_{yj}(x) = c_j + \sum_{k} W_{jk} x_k + U_{jy}$$

As a result, we can compute the gradient exactly.

Theoretical Question 5 answer

To derive the equality (2) we start from the conditional probability $p(y|x)$ and its log-likelihood. Let's break this down step by step.

Write the conditional probability $p(y|x)$

From the previous (1) derivation, the conditional probability $p(y|x)$ is:

$$p(y|x) = \frac{e^{d_y} \prod_{j=1}^{n} \left(1 + e^{o_{yj}(\mathbf{x})}\right)}{\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{o_{y^*j}(\mathbf{x})}\right)},$$

where:

$$o_{yj}(\mathbf{x}) = c_j + \sum_k W_{jk} x_k + U_{jy}$$

Take the logarithm of $p(y|x)$

The log-likelihood of $p(y|x)$ is:

$$\log p(y|x) = \log \left(e^{d_y} \prod_{j=1}^{n} \left(1 + e^{o_{yj}(\mathbf{x})}\right) \right) - \log \left(\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{o_{y^*j}(\mathbf{x})}\right) \right)$$

Simplify the first term:

$$\log \left(e^{d_y} \prod_{j=1}^{n} \left(1 + e^{o_{yj}(\mathbf{x})}\right) \right) = d_y + \sum_{j=1}^{n} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right)$$

Thus:

$$\log p(y|x) = d_y + \sum_{j=1}^{n} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right) - \log \left(\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{o_{y^*j}(\mathbf{x})}\right) \right)$$

Compute the gradient of $\log p(y|x)$ **with respect to** θ

We want to compute:

$$\frac{\partial \log p(y|x)}{\partial \theta}$$

From the expression for $\log p(y|x)$, the gradient is:

$$\frac{\partial \log p(y|x)}{\partial \theta} = \sum_{j=1}^{n} \frac{\partial}{\partial \theta} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right) - \frac{\partial}{\partial \theta} \log \left(\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{o_{y^*j}(\mathbf{x})}\right) \right)$$

Compute the first term

The first term is:

$$\sum_{j=1}^{n} \frac{\partial}{\partial \theta} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right)$$

Using the chain rule:

$$\frac{\partial}{\partial \theta} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right) = \frac{e^{o_{yj}(\mathbf{x})}}{1 + e^{o_{yj}(\mathbf{x})}} \cdot \frac{\partial o_{yj}(\mathbf{x})}{\partial \theta}$$

Since:

$$\text{sigm}(o_{yj}(\mathbf{x})) = \frac{e^{o_{yj}(\mathbf{x})}}{1 + e^{o_{yj}(\mathbf{x})}},$$

we get:

$$\sum_{j=1}^{n} \frac{\partial}{\partial \theta} \log \left(1 + e^{o_{yj}(\mathbf{x})}\right) = \sum_{j=1}^{n} \text{sigm}(o_{yj}(\mathbf{x})) \frac{\partial o_{yj}(\mathbf{x})}{\partial \theta}$$

Compute the second term

The second term is:

$$\frac{\partial}{\partial \theta} \log \left(\sum_{y^*} e^{d_{y^*}} \prod_{j=1}^{n} \left(1 + e^{o_{y^*j}(\mathbf{x})}\right) \right)$$

By applying the chain rule and summing over y^*, we obtain:

$$\sum_{y^*} p(y^*|x) \sum_{j=1}^{n} \text{sigm}(o_{y^*j}(\mathbf{x})) \frac{\partial o_{y^*j}(\mathbf{x})}{\partial \theta}$$

Combine the terms

Thus, we arrive at:

$$\frac{\partial \log p(y_i|x_i)}{\partial \theta} = \sum_{j} \text{sigm}(o_{yj}(\mathbf{x_i})) \frac{\partial o_{yj}(\mathbf{x_i})}{\partial \theta} - \sum_{j,y^*} \text{sigm}(o_{y^*j}(\mathbf{x_i})) p(y^*|x_i) \frac{\partial o_{y^*j}(\mathbf{x_i})}{\partial \theta}$$

Simulation Question 2

Load the **MNIST** dataset and convert it to a binary state (for example, by applying a threshold to the pixel intensity). Then, train a **RBM** to estimate the joint distribution $p(y, x)$ using the algorithm 3. After training the model, perform final classification using $p(y|x)$ and report its accuracy.

Algorithm 1 Training update for RBM over $(y, \mathbf{x})$ using Contrastive Divergence

Input: training pair $(y_i, \mathbf{x}_i)$ and learning rate λ
% Notation: $a \leftarrow b$ means a is set to value b
% $a \sim p$ means a is sampled from p

% Positive phase
$y^0 \leftarrow y_i$, $\mathbf{x}^0 \leftarrow \mathbf{x}_i$, $\widehat{\mathbf{h}}^0 \leftarrow \text{sigm}(\mathbf{c} + W\mathbf{x}^0 + \mathbf{U}\vec{y^0})$

% Negative phase
$\mathbf{h}^0 \sim p(\mathbf{h}|y^0, \mathbf{x}^0)$, $y^1 \sim p(y|\mathbf{h}^0)$, $\mathbf{x}^1 \sim p(\mathbf{x}|\mathbf{h}^0)$
$\widehat{\mathbf{h}}^1 \leftarrow \text{sigm}(\mathbf{c} + W\mathbf{x}^1 + \mathbf{U}\vec{y^1})$

% Update
for $\theta \in \Theta$ **do**
  $\theta \leftarrow \theta - \lambda \left(\frac{\partial}{\partial \theta} E(y^0, \mathbf{x}^0, \widehat{\mathbf{h}}^0) - \frac{\partial}{\partial \theta} E(y^1, \mathbf{x}^1, \widehat{\mathbf{h}}^1) \right)$
end for

Figure 3: Training update for RBM over (y, x) using Contrastive Divergence

Simulation Question 2 answer

Dataset Preprocessing

We use the MNIST dataset and binarize it using a threshold of 0.5 after normalizing the values between 0 and 1. The processed dataset is shown in Figure 4.

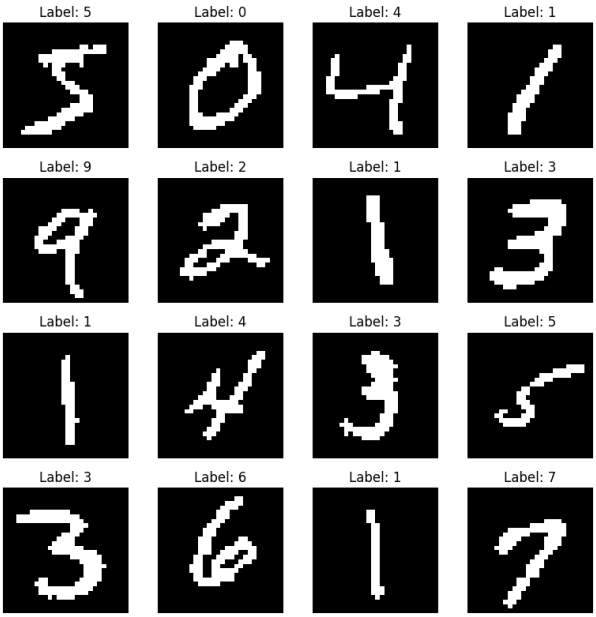

Figure 4: Binarized MNIST dataset.

RBM Model Definition

An RBM consists of visible units v and hidden units h, connected via a weight matrix W. The energy function of the RBM is defined as:

$$E(v, h) = -v^T W h - v^T b - h^T c$$

where b and c are biases for the visible and hidden layers, respectively.

Model Implementation

Below is the PyTorch implementation of our RBM model:

```
class RBM(nn.Module):
  def __init__(self, n_visible=784+10, n_hidden=128, device='cpu'):
      super(RBM, self).__init__()
      self.n_visible = n_visible
      self.n_hidden = n_hidden
      self.device = device
      self.W = nn.Parameter(torch.randn(n_visible, n_hidden) * 0.01)
      self.hbias = nn.Parameter(torch.zeros(n_hidden))
      self.vbias = nn.Parameter(torch.zeros(n_visible))
      self.to(self.device)

  def sample_h_given_v(self, v):
      h_mean = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
      return h_mean, torch.bernoulli(h_mean)

  def sample_v_given_h(self, h):
      v_mean = torch.sigmoid(F.linear(h, self.W, self.vbias))
      return v_mean, torch.bernoulli(v_mean)

  def gibbs_hvh(self, h0_sample):
      v_mean, v_sample = self.sample_v_given_h(h0_sample)
      h_mean, h_sample = self.sample_h_given_v(v_sample)
      return v_mean, v_sample, h_mean, h_sample

```

```
    def contrastive_divergence(self, data, k=1, lr=0.1):
        data = data.to(self.device)
        ph_mean, ph_sample = self.sample_h_given_v(data)
        chain_start = ph_sample
        for _ in range(k):
            nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh(chain_start)
            chain_start = nh_sample
        self.W.data += lr * (torch.mm(data.t(), ph_sample) -
                torch.mm(nv_sample.t(), nh_mean)) / data.size(0)
        self.vbias.data += lr * torch.mean(data - nv_sample, dim=0)
        self.hbias.data += lr * torch.mean(ph_sample - nh_mean, dim=0)

    def energy(self, v, h):
        return -torch.matmul(v, torch.matmul(self.W, h)) -
                torch.matmul(v, self.vbias) - torch.matmul(h, self.hbias)

    def reconstruct(self, v):
        h = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
        return torch.sigmoid(F.linear(h, self.W, self.vbias))

    def save_model(self, file_path):
        with open(file_path, 'wb') as f:
            pickle.dump(self, f)

    @staticmethod
    def load_model(file_path, device=None):
        with open(file_path, 'rb') as f:
            model = pickle.load(f)
        model.device = device if device else torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model.to(model.device)
        return model
```

Training the RBM

We train the RBM using Contrastive Divergence (CD-k), where $k = 1$ is commonly used for efficiency. The training function is implemented as follows:

```
def train_rbm(train_loader, test_loader, n_visible, n_hidden, learning_rate, epochs, k, device):
    rbm = RBM(n_visible, n_hidden, device).to(device)
    optimizer = torch.optim.SGD([rbm.W, rbm.hbias, rbm.vbias], lr=learning_rate)

    for epoch in range(epochs):
        for batch, label in train_loader:
            batch = batch.view(batch.size(0), -1).to(device)
            batch = torch.cat((batch, label.to(device)), dim=1)
            optimizer.zero_grad()
            rbm.contrastive_divergence(batch, k=k)
            optimizer.step()
    return rbm
```

Classification Using RBM Energy

The trained RBM can classify images by computing the energy of all possible labels and selecting the one with the lowest energy:

```
def classify_rbm(model, x, device="cpu"):
    model.to(device)
    x = x.flatten().to(device)
    y_classes = torch.eye(10, device=device)
    x_repeated = x.repeat(10, 1)
    xy_concat = torch.cat((x_repeated, y_classes), dim=1)
    _, h_sampled = model.sample_h_given_v(xy_concat)
    energies = torch.tensor([model.energy(v, h) for v, h in zip(xy_concat, h_sampled)], device=device)
    return torch.argmin(energies).item()
```

All the results have been gathered in theoretical question 6.

Simulation Question 3

Load the MNIST dataset and convert it to a binary format (e.g., by applying a threshold to pixel intensities). Then, train an RBM to estimate the distribution $p(x, y)$ using the algorithm in Figure 2. After training the model, perform classification using $p(y|x)$ and report its accuracy.

Simulation Question 3 answer

The DRBM model is similar to the RBM but differs in how it handles visible units. Instead of treating all visible units the same, it explicitly separates the feature vector x and the label vector y. During Gibbs sampling, x is kept fixed, and only y is sampled. like Algorithm 5 .

Algorithm 2 Training update forDRBM over $(y\ \mathbf{x})$ using Contrastive Divergence

Input: training pair $(y_i, \mathbf{x}_i)$ and learning rate λ
% Notation: $a \leftarrow b$ means a is set to value b
% $a \sim p$ means a is sampled from p

% Positive phase
$y^0 \leftarrow y_i$, $\mathbf{x}^0 \leftarrow \mathbf{x}_i$, $\widehat{\mathbf{h}}^0 \leftarrow \text{sigm}(\mathbf{c} + W\mathbf{x}^0 + \mathbf{U}\vec{y^0})$

% Negative phase
$\mathbf{h}^0 \sim p(\mathbf{h}|y^0, \mathbf{x}^0)$, $y^1 \sim p(y|\mathbf{h}^0)$
$\widehat{\mathbf{h}}^1 \leftarrow \text{sigm}(\mathbf{c} + W\mathbf{x}^0 + \mathbf{U}\vec{y^1})$

% Update
for $\theta \in \Theta$ **do**
$\theta \leftarrow \theta - \lambda \left(\frac{\partial}{\partial \theta} E(y^0, \mathbf{x}^0, \widehat{\mathbf{h}}^0) - \frac{\partial}{\partial \theta} E(y^1, \mathbf{x}^0, \widehat{\mathbf{h}}^1) \right)$
end for

Figure 5: Training update for DRBM over $(y|x)$ using Contrastive Divergence

Model Implementation

```python
class DRBM(nn.Module):
    def __init__(self, n_visible=784, n_hidden=128, label_size=10, device='cpu'):
        super(DRBM, self).__init__()
        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.label_size = label_size
        self.feature_size = n_visible - label_size
        self.device = device

        self.W = nn.Parameter(torch.randn(n_visible, n_hidden) * 0.01)
        self.hbias = nn.Parameter(torch.zeros(n_hidden))
        self.vbias = nn.Parameter(torch.zeros(n_visible))

        self.to(self.device)

    def sample_h_given_v(self, v):
        h_mean = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
        h_sample = torch.bernoulli(h_mean)
        return h_mean, h_sample

    def sample_v_given_h(self, h, x_fixed):
        v_mean = torch.sigmoid(F.linear(h, self.W, self.vbias))
        v_sample = v_mean.clone()
```

```
        v_sample[:, :self.feature_size] = x_fixed
        v_sample[:, self.feature_size:] = torch.bernoulli(v_mean[:, self.feature_size:])
        return v_mean, v_sample

    def gibbs_hvh(self, h0_sample, x_fixed):
        v_mean, v_sample = self.sample_v_given_h(h0_sample, x_fixed)
        h_mean, h_sample = self.sample_h_given_v(v_sample)
        return v_mean, v_sample, h_mean, h_sample

    def contrastive_divergence(self, data, k=1, lr=0.1):
        data = data.to(self.device)
        x_fixed = data[:, :self.feature_size]
        ph_mean, ph_sample = self.sample_h_given_v(data)

        chain_start = ph_sample
        for _ in range(k):
            nv_mean, nv_sample, nh_mean, nh_sample = self.gibbs_hvh(chain_start, x_fixed)
            chain_start = nh_sample

        self.W.data +=
            lr * (torch.mm(data.t(), ph_sample) - torch.mm(nv_sample.t(), nh_mean)) / data.size(0)
        self.vbias.data += lr * torch.mean(data - nv_sample, dim=0)
        self.hbias.data += lr * torch.mean(ph_sample - nh_mean, dim=0)

    def energy(self, v, h):
        term1 = -torch.sum((v @ self.W) * h, dim=1)
        term2 = -torch.matmul(v, self.vbias)
        term3 = -torch.matmul(h, self.hbias)
        return term1 + term2 + term3

    def reconstruct(self, v):
        v = v.to(self.device)
        h = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
        reconstructed_v = torch.sigmoid(F.linear(h, self.W, self.vbias))
        reconstructed_v[:, :self.feature_size] = v[:, :self.feature_size]
        return reconstructed_v , v[:, self.feature_size:]
```

Training and Classification Using RBM Energy

The training and classification process follows the same approach as the standard RBM, with one key difference: in DRBM, Contrastive Divergence (CD) is modified to ensure that x remains fixed while sampling y. This adjustment allows the model to learn the conditional probability $p(y|x)$ directly.

All the results have been gathered in theoretical question 6.

Simulation Question 4

Once again, perform training using the hybrid method with $\alpha \in \{1, 0.1, 0.01\}$, then classify using $p(y|x)$ and report the final accuracy.

Simulation Question 4 answer

The HybridRBM model is a modified Restricted Boltzmann Machine (RBM) that incorporates a unified learning approach by combining two types of loss functions: the standard RBM loss and the Discriminative RBM (DRBM) loss, weighted by a parameter α.

Model Implementation

Below is the implementation of the HybridRBM model:

```
class UnifiedRBM(nn.Module):
    def __init__(self, n_visible=784+10, n_hidden=128, label_size=10, alpha=0.5, device='cpu'):
        super(UnifiedRBM, self).__init__()

        self.n_visible = n_visible
        self.n_hidden = n_hidden
        self.label_size = label_size
        self.feature_size = n_visible - label_size
        self.alpha = alpha
        self.device = device

        self.W = nn.Parameter(torch.randn(self.n_visible, n_hidden) * 0.01)
        self.hbias = nn.Parameter(torch.zeros(n_hidden))
        self.vbias = nn.Parameter(torch.zeros(self.n_visible))

        self.to(self.device)

    def sample_h_given_v(self, v):
        h_mean = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
        h_sample = torch.bernoulli(h_mean)
        return h_mean, h_sample

    def sample_v_given_h(self, h, x_fixed=None):
        v_mean = torch.sigmoid(F.linear(h, self.W, self.vbias))
        v_sample = torch.bernoulli(v_mean)

        if x_fixed is not None:
            v_sample[:, :self.feature_size] = x_fixed
        return v_mean, v_sample

    def gibbs_hvh(self, h0_sample, x_fixed=None):
        v_mean, v_sample = self.sample_v_given_h(h0_sample, x_fixed)
        h_mean, h_sample = self.sample_h_given_v(v_sample)
        return v_mean, v_sample, h_mean, h_sample

    def contrastive_divergence(self, data, k=1, lr=0.1):
        data = data.to(self.device)
        x_fixed = data[:, :self.feature_size]

        ph_mean_dis, ph_sample_dis = self.sample_h_given_v(data)
        chain_start_dis = ph_sample_dis
        for _ in range(k):
            nv_mean_dis, nv_sample_dis, nh_mean_dis, nh_sample_dis =
                    self.gibbs_hvh(chain_start_dis, x_fixed)
            chain_start_dis = nh_sample_dis

        W_update_dis = torch.mm(data.t(), ph_sample_dis) - torch.mm(nv_sample_dis.t(), nh_mean_dis)
        vbias_update_dis = torch.mean(data - nv_sample_dis, dim=0)
        hbias_update_dis = torch.mean(ph_sample_dis - nh_mean_dis, dim=0)

        ph_mean_gen, ph_sample_gen = self.sample_h_given_v(data)
        chain_start_gen = ph_sample_gen
        for _ in range(k):
            nv_mean_gen, nv_sample_gen, nh_mean_gen, nh_sample_gen = self.gibbs_hvh(chain_start_gen)
            chain_start_gen = nh_sample_gen

        W_update_gen = torch.mm(data.t(), ph_sample_gen) - torch.mm(nv_sample_gen.t(), nh_mean_gen)
        vbias_update_gen = torch.mean(data - nv_sample_gen, dim=0)
        hbias_update_gen = torch.mean(ph_sample_gen - nh_mean_gen, dim=0)

        W_update = W_update_dis + (self.alpha) * W_update_gen
        vbias_update = vbias_update_dis + (self.alpha) * vbias_update_gen
        hbias_update = hbias_update_dis + (self.alpha) * hbias_update_gen

        self.W.data += lr * W_update / data.size(0)
        self.vbias.data += lr * vbias_update
        self.hbias.data += lr * hbias_update

```

```
    def energy(self, v, h):
        term1 = -torch.matmul(v, torch.matmul(self.W, h))
        term2 = -torch.matmul(v, self.vbias)
        term3 = -torch.matmul(h, self.hbias)
        return term1 + term2 + term3

    def reconstruct(self, v):
        v = v.to(self.device)
        h = torch.sigmoid(F.linear(v, self.W.t(), self.hbias))
        reconstructed_v = torch.sigmoid(F.linear(h, self.W, self.vbias))
        return reconstructed_v, v[:, self.feature_size:]

    def save_model(self, file_path):
        with open(file_path, 'wb') as f:
            pickle.dump(self, f)

    @staticmethod
    def load_model(file_path, device=None):
        with open(file_path, 'rb') as f:
            model = pickle.load(f)
        model.device = device if device else torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model.to(model.device)
        return model
```

Training and Classification

Training and classification using the HybridRBM model follows the standard RBM process, but the Contrastive Divergence (CD) method has been modified within the model's implementation.

All the results have been gathered in theoretical question 6.

Theoretical Question 6

Report the final learning outcomes and classification accuracy for all previous models and compare them. Are the results as expected?

Theoretical Question 6 answer

Model	Precision	Recall	F1-score
RBM (CD=1)	0.78	0.78	0.78
RBM (CD=5)	**0.79**	0.78	0.78
RBM (CD=10)	0.78	0.78	0.78
DRBM (CD=1)	0.81	0.80	0.80
DRBM (CD=5)	**0.83**	0.82	0.82
DRBM (CD=10)	**0.83**	**0.83**	**0.83**
Mix (α = 0.01 , CD=1)	0.77	0.76	0.76
Mix (α = 0.1, CD=1)	0.80	0.80	0.80
Mix (α = 1, CD=1)	0.82	0.82	0.82
Mix (α = 0.01, CD=5)	0.80	0.79	0.79
Mix (α = 0.1, CD=5)	0.81	0.81	0.81
Mix (α = 1, CD=5)	**0.83**	0.82	0.82
Mix (α = 0.01 , CD=10)	0.81	0.81	0.81
Mix (α = 0.1, CD=10)	0.82	0.82	0.82
Mix (α = 1, CD=10)	**0.83**	**0.83**	**0.83**

Table 3: Performance comparison of different RBM-based models. Bold values indicate the best performance in each metric.

Analysis of Model Performance

Table 3 highlights some key insights into how different models perform:

- **RBM Performance:** The standard RBM with Contrastive Divergence (CD) shows stable performance across different CD values. Increasing CD from 1 to 5 leads to slight improvements, but beyond that (CD=10), the gains are negligible. The best performance is observed at CD=5, with a precision of 0.79.
- **DRBM Performance:** The Discriminative RBM (DRBM) performs significantly better than the standard RBM. This suggests that explicitly separating the feature vector x from the label vector y enhances classification. Performance improves as CD increases, reaching its best at CD=10, where all metrics achieve 0.83. This indicates that deeper sampling benefits the DRBM framework.
- **Mix Model Performance:** The hybrid Mix RBM model, which combines RBM and DRBM losses, is highly sensitive to the tuning parameter α. The results show that:
  - At $\alpha = 0.01$, the model underperforms compared to DRBM, likely because the RBM loss dominates.
  - At $\alpha = 0.1$, performance improves, approaching that of DRBM.
  - At $\alpha = 1$, the model matches DRBM (CD=10), suggesting that at this setting, the DRBM loss is the dominant factor.

  This suggests that adjusting α allows the Mix model to balance between RBM and DRBM characteristics, making it a flexible option for different tasks.

Expected vs. Actual Results

- As anticipated, DRBM outperforms the standard RBM, reinforcing the idea that conditioning the visible layer helps classification.
- The Mix model with $\alpha = 1$ achieves performance on par with DRBM (CD=10), confirming that combining both training approaches is effective.
- Increasing CD generally enhances performance, but improvements taper off after CD=5, particularly for RBM.

Conclusion

- **Best overall model:** The DRBM (CD=10) and Mix model ($\alpha = 1, CD = 10$) both achieve the highest performance, with all metrics reaching 0.83.
- **Hybrid approach is effective:** The Mix model can achieve DRBM-level performance while offering flexibility through α.

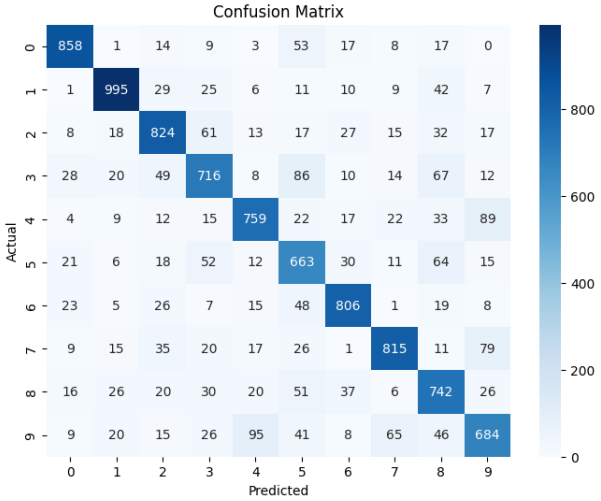

Figure 6: Confusion Matrix for RBM

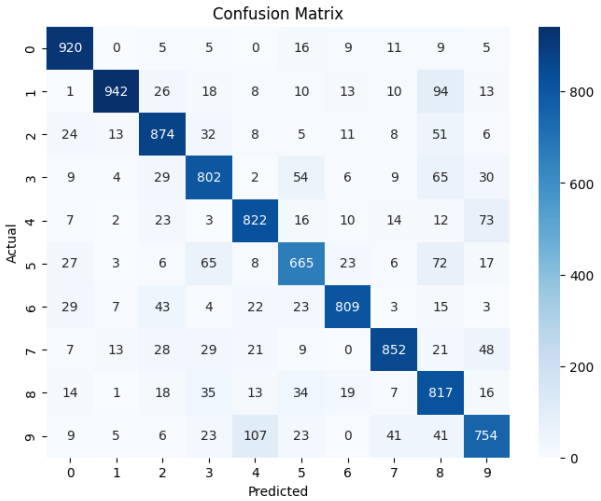

Figure 7: Confusion Matrix for DRBM

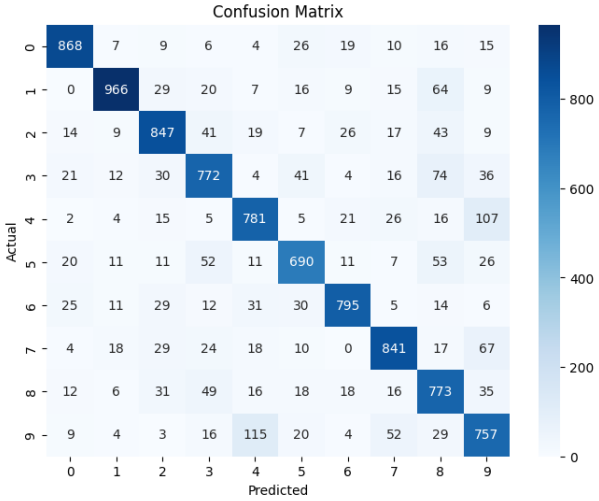

Figure 8: Confusion Matrix for Mix $\alpha = 0.01$

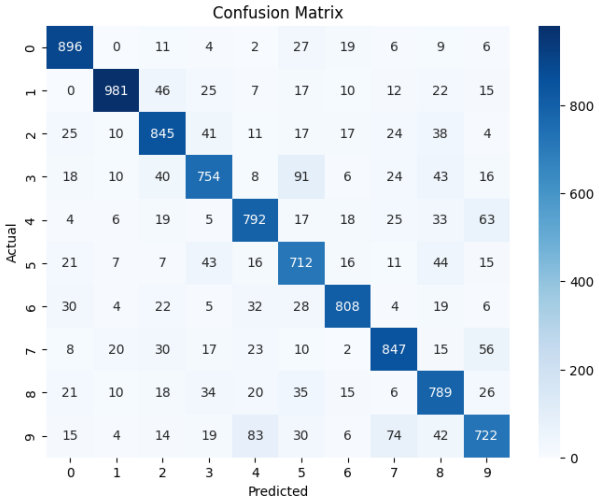

Figure 9: Confusion Matrix for Mix $\alpha = 0.1$

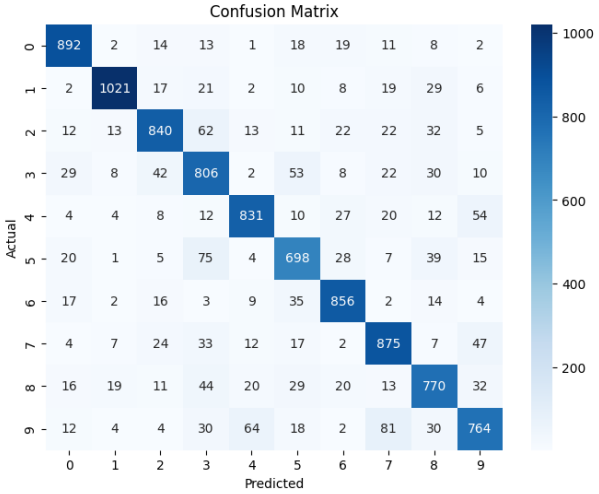

Figure 10: Confusion Matrix for Mix $\alpha = 1$

Theoretical Question 7

Can we use unlabeled data D_{unsup} alongside labeled data D_{sup}? To be precise, we aim for a final objective function as follows:

$$\mathcal{L}_{semi-sup}(D_{sup}, D_{unsup}) = \mathcal{L}_{sup}(D_{sup}) + \beta \mathcal{L}_{unsup}(D_{unsup})$$

Propose a suitable function for $\mathcal{L}_{unsup}$ and explain the details of its optimization.

Theoretical Question 7 answer

Yes, we can use unlabeled data D_{unsup} alongside labeled data D_{sup} in semi-supervised learning. Semi-supervised learning algorithms leverage the unlabeled data to introduce constraints on the trained model, which helps improve generalization.

Defining $\mathcal{L}_{unsup}$

A suitable choice for $\mathcal{L}_{unsup}$ is the negative log-likelihood of the unlabeled data under the model's distribution:

$$\mathcal{L}_{unsup}(D_{unsup}) = -\sum_{i=1}^{|D_{unsup}|} \log p(x_i)$$

where $D_{unsup} = \{x_i\}_{i=1}^{|D_{unsup}|}$ represents the set of unlabeled inputs. This function ensures that the model learns a good generative representation of the unlabeled data, making it useful for semi-supervised learning.

Optimization Strategy

To optimize this loss, we use **contrastive divergence**, which provides an approximation of the log-likelihood gradient. The gradient is computed as:

$$\frac{\partial \log p(x_i)}{\partial \theta} = -\mathbb{E}_{y,h|x_i}\left[\frac{\partial}{\partial \theta} E(y_i, x_i, h)\right] + \mathbb{E}_{y,x,h}\left[\frac{\partial}{\partial \theta} E(y, x, h)\right]$$

where $E(y, x, h)$ is the energy function in the RBM framework. The first term can be efficiently computed using:

$$\mathbb{E}_{y|x_i}\left[\mathbb{E}_{h|y,x_i}\left[\frac{\partial}{\partial \theta} E(y_i, x_i, h)\right]\right]$$

This optimization can be performed using **sampling techniques**, where we either:

1. Compute the expected gradient over all possible class labels y, weighted by $p(y|x_i)$, or
2. Sample a class label $y \sim p(y|x_i)$ and compute the gradient for the selected y.

Final Semi-Supervised Objective

The final objective function for semi-supervised learning is:

$$\mathcal{L}_{semi-sup}(D_{sup}, D_{unsup}) = \mathcal{L}_{TYPE}(D_{sup}) + \beta \mathcal{L}_{unsup}(D_{unsup})$$

where $\mathcal{L}_{TYPE}$ corresponds to a supervised loss, which can be generative (gen), discriminative ($disc$), or a hybrid of both. The parameter β controls the contribution of the unsupervised loss.