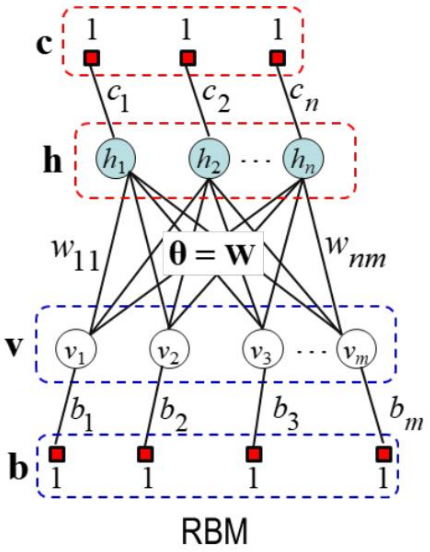
PATTERN RECOGNITION – Homework 4

Solution

Implement Restricted Boltzmann Machine (RBM) using numpy:



The RBM implemented in this assignment will contain:

+ input layers: 28 x 28 = 784 neurons (based on the size of training image from MNIST dataset)

+ hidden layers: configurable n neurons

+ weight W of size 784 x n

+ bias b for visible layer: 784 size

+ bias c for hidden layer: n size

An RBM class is defined in the following codes:

Firstly, we initialize the weights of the model in the __init__(), the v_size is the number of neurons in input layer (784) and h_size if the number of neurons in hidden layer (default is 100) and can be configured in the the code. W, b, c are weights and biases we mentioned above.

```
class RBM:
  def __init__(self, v_size=784, h_size=100):
    self.v_size = v_size
    self.h_size = h_size
    self.W = np.random.uniform(-1, 1, (v_size, h_size)) # v_size x h_size
    self.b = np.zeros(v_size)
    self.c = np.zeros(h_size)
```

We use sigmoid as the activation function:

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

The function sample_h() is to:

a. Compute $Q\left(h_j^{(t)} = 1|\mathbf{v}^{(t)}\right)$ - it can be computed as

$$Q\left(h_j^{(t)} = 1\middle|\mathbf{v}^{(t)}\right) = \sigma\left(\sum_{i=1}^{m} w_{ij} v_i + b_j\right),$$

+ "ph" as the probability of h given v

+ "h" as hidden status sampled from Bernoulli distribution

b. Sample $h_j^{(t)}$ from $Q\left(h_j^{(t)} = 1|\mathbf{v}^{(t)}\right)$.

The sampling of hidden states from Bernoulli distribution can be given as:

$$h_j^{(t)} = \begin{cases} 1, & \text{if } Q\left(h_j^{(t)} = 1\middle|\mathbf{v}^{(t)}\right) > rand(0,1) \\ 0, & o.w. \end{cases}$$

```
def sample_h(self, v): # compute probability and sample h given v
  ph = self.sigmoid(v @ self.W + self.c) # batch x h_size --> P(h|v)
  rand = np.random.uniform(0, 1, size=ph.shape)
  h = 1 * (ph > rand)
  return ph, h
```

The function sample_v() is used in the negative phase to:

a. Compute $P\left(\hat{v}_j^{(t+1)} = 1|\mathbf{h}^{(t)}\right)$ - it can be computed as

$$Q\left(\hat{v}_i^{(t+1)} = 1\middle|\mathbf{h}^{(t)}\right) = \sigma\left(\sum_{j=1}^{n} w_{ij} h_j^{(t)} + c_j\right),$$

+ "pv" as the probability of v given h

+ "v" as visible states sampled from Bernoulli distribution

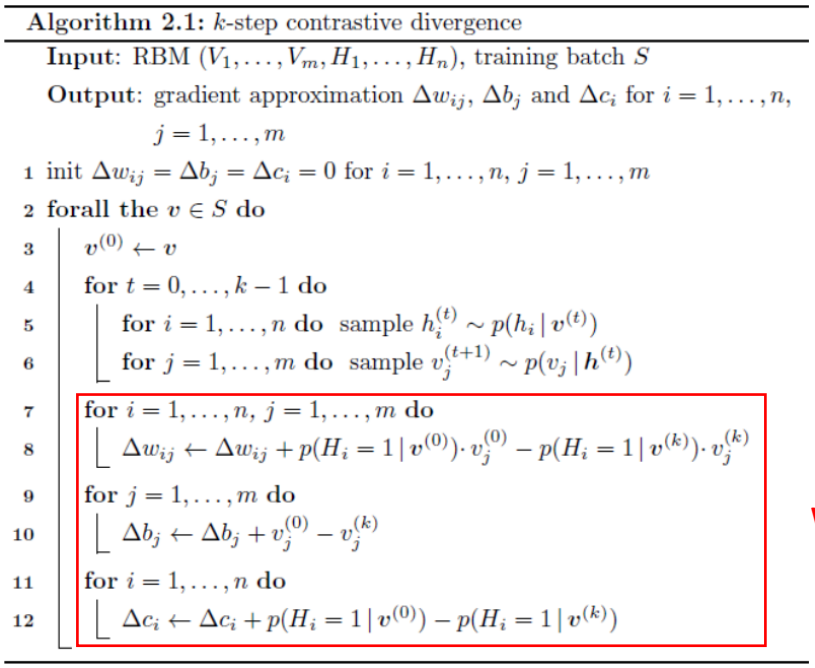
b. Sample $\hat{v}_j^{(t+1)}$ from $P\left(\hat{v}_j^{(t+1)} = 1|\mathbf{h}^{(t)}\right)$.

The sampling of visible states from Bernoulli distribution can be given as:

$$\hat{v}_i^{(t+1)} = \begin{cases} 1, & \text{if } Q\left(\hat{v}_i^{(t+1)} = 1\middle|\mathbf{h}^{(t)}\right) > rand(0,1) \\ 0, & o.w. \end{cases}$$

```
def sample_v(self, h): # compute probability and sample v given h
  pv = self.sigmoid(h @ self.W.T + self.b) # batch x v_size --> P(v|h)
  rand = np.random.uniform(0, 1, size=pv.shape)
  v = 1 * (pv > rand)
  return pv, v
```

The training function follows the k-step Contrastive divergence algorithm, which performing multiple steps of Gibbs sampling for the negative phase.



Algorithm 2.1: k-step contrastive divergence

Input: RBM $(V_1, \ldots, V_m, H_1, \ldots, H_n)$, training batch S

Output: gradient approximation Δw_{ij}, Δb_j and Δc_i for $i = 1, \ldots, n$, $j = 1, \ldots, m$

init $\Delta w_{ij} = \Delta b_j = \Delta c_i = 0$ for $i = 1, \ldots, n$, $j = 1, \ldots, m$

forall the $v \in S$ **do**

  $v^{(0)} \leftarrow v$

  for $t = 0, \ldots, k-1$ **do**

    for $i = 1, \ldots, n$ **do** sample $h_i^{(t)} \sim p(h_i \mid v^{(t)})$

    for $j = 1, \ldots, m$ **do** sample $v_j^{(t+1)} \sim p(v_j \mid h^{(t)})$

  for $i = 1, \ldots, n$, $j = 1, \ldots, m$ **do**

    $\Delta w_{ij} \leftarrow \Delta w_{ij} + p(H_i = 1 \mid v^{(0)}) \cdot v_j^{(0)} - p(H_i = 1 \mid v^{(k)}) \cdot v_j^{(k)}$

  for $j = 1, \ldots, m$ **do**

    $\Delta b_j \leftarrow \Delta b_j + v_j^{(0)} - v_j^{(k)}$

  for $i = 1, \ldots, n$ **do**

    $\Delta c_i \leftarrow \Delta c_i + p(H_i = 1 \mid v^{(0)}) - p(H_i = 1 \mid v^{(k)})$

Weight update

In practice, since setting k = 1 has yielded good results, k = 1 is used in this assignment. The default learning rate ("lr") is 0.05.

```
def train_one_epoch(self, dataloader, lr=0.05, max_epoch=1, k=1):
  for epoch in range(max_epoch):
    for batch_idx, (input, target) in enumerate(dataloader["train"]):
      batch_size = input.shape[0]
      input, target = input.view(batch_size, -1).numpy(), target.view(batch_size, -1).numpy() # tensor to numpy

      v_0 = input
      # Positive phase:
      ph_0, h_0 = self.sample_h(v_0)

      h_k = h_0
      # Negative phase:
      for i in range(k):
        pv_k, v_k = self.sample_v(h_k) # use h^(t) to generate negative data
        ph_k, h_k = self.sample_h(v_k) # use v^(t+1) to generate negative data

      # Update weight:
      self.W += lr * (v_0.T @ ph_0 - v_k.T @ ph_k) / batch_size
      self.b += lr * np.sum(v_0 - v_k, axis=0) / batch_size
      self.c += lr * np.sum(ph_0 - ph_k, axis=0) / batch_size
```

Given the RBM defined above, now we can solve our classification:

Classification

Some necessary libraries:

```
import numpy as np
```

```
import random
import torch
from torch.utils.data import DataLoader, Dataset
from torchvision.datasets import MNIST
import torchvision.transforms as transforms

# set random seed for reproducibility
torch.manual_seed(0)
random.seed(0)
np.random.seed(0)
```

Use Dataloader from pytorch to download the MNIST dataset and split it into training, validation, test set with the size of 50000, 10000, 10000 samples respectively:

```
mnist_train = MNIST("./", train=True, transform=transforms.ToTensor(),
target_transform=None, download=True)
mnist_test = MNIST("./", train=False, transform=transforms.ToTensor(),
target_transform=None, download=True)
mnist_train, mnist_valid = torch.utils.data.random_split(mnist_train,
[50000, 10000])
dataloader = {}
dataloader["train"] = DataLoader(mnist_train, batch_size=32, shuffle=True)
dataloader["valid"] = DataLoader(mnist_valid, batch_size=32, shuffle=True)
dataloader["test"] = DataLoader(mnist_test, batch_size=32, shuffle=False)
```

For our classification problem, a simple classifier is defined as follow:

```
import torch.nn as nn
import torch.optim as optim
from torch.utils.tensorboard import SummaryWriter
%load_ext tensorboard
writer = SummaryWriter() # tensorboard writer

class Classifier(nn.Module):
  def __init__(self, input_size=100, output_size=10):
    super().__init__()
    self.linear = nn.Linear(input_size, output_size)
    self.softmax = nn.Softmax(dim=1)

  def forward(self, x):
    return self.softmax(self.linear(x))
```

The input is the representation generated from a partially trained RBM. The input size should be the same as the number of neurons in hidden layers of RBM model we defined before and the output size is the number of classes in MNIST from 0 → 9, which is 10 by default.

Now, given the RBM and Classifier classes we defined, we can now start training procedure. Firstly, we create rbm_model and cls_model. We use Cross Entropy as the loss function and SGD as the optimizer. The learning rate for the classifier is 0.05. The number of epochs is 30.

```
hidden_layer_neurons = 100
rbm_model = RBM(v_size=784, h_size=hidden_layer_neurons)
cls_model = Classifier(input_size=hidden_layer_neurons)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(cls_model.parameters(), lr=0.05)

max_rbm_epoch = 30
max_cls_epoch = 1
```

The training procedure with explanation in the code:

```
for rbm_epoch in range(max_rbm_epoch):
  # Train RBM model first
  rbm_model.train_one_epoch(dataloader)

  # Given the partially trained RBM model (freeze RBM's weights)
  # Train the classifier using validation data
  for cls_epoch in range(max_cls_epoch):
    for batch_index, (input, target) in enumerate(dataloader["valid"]):
      batch_size = input.shape[0]
      _, hidden_output = rbm_model.sample_h(input.view(batch_size, -1).numpy()) # generate hidden represetation
      hidden_output = torch.from_numpy(hidden_output.astype(np.float32))
      pred = cls_model(hidden_output) # prediction output
      loss = criterion(pred, target) # training loss
      # Backward and optimization
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
```

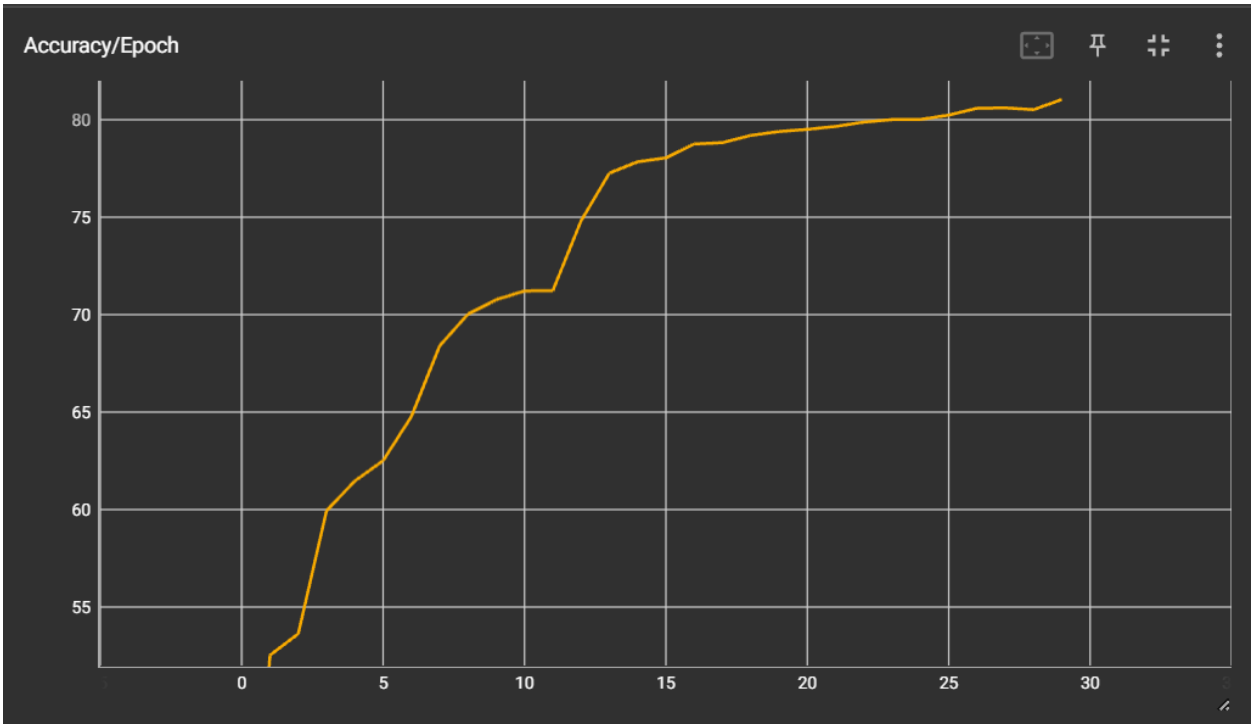
```
  # Evaluate on test data
  cls_model.eval()
  with torch.no_grad():
    count = 0
    correct = 0
    total_loss = 0.0
    for image, target in dataloader["test"]:
      _, hidden_output = rbm_model.sample_h(image.view(image.shape[0], -1).numpy()) # generate hidden representation
      hidden_output = torch.from_numpy(hidden_output.astype(np.float32))
      test_pred = cls_model(hidden_output) # prediction output
      test_loss = criterion(test_pred, target) # loss on test data
      count += image.shape[0]
      correct += test_pred.max(-1).indices.eq(target).sum().item() # number of correct predictions
      total_loss += loss.item()
    accuracy = 100 * correct / count
    avg_loss = total_loss / len(dataloader["test"])
    print(f'Epoch [{rbm_epoch + 1}/{max_rbm_epoch}], Loss: {avg_loss:.4f}, Accuracy: {accuracy:.2f}%')
    writer.add_scalar('Loss/Epoch', avg_loss, rbm_epoch) # log
    writer.add_scalar('Accuracy/Epoch', accuracy, rbm_epoch) # log

writer.flush()
writer.close()
```

Output:

```
Epoch [1/30], Loss: 2.2051, Accuracy: 33.10%
Epoch [2/30], Loss: 2.0777, Accuracy: 52.59%
Epoch [3/30], Loss: 2.1008, Accuracy: 53.69%
Epoch [4/30], Loss: 1.8826, Accuracy: 59.99%
Epoch [5/30], Loss: 1.9952, Accuracy: 61.51%
Epoch [6/30], Loss: 1.9352, Accuracy: 62.55%
Epoch [7/30], Loss: 1.8497, Accuracy: 64.81%
Epoch [8/30], Loss: 1.7659, Accuracy: 68.43%
Epoch [9/30], Loss: 1.9075, Accuracy: 70.05%
Epoch [10/30], Loss: 1.8916, Accuracy: 70.79%
Epoch [11/30], Loss: 1.7339, Accuracy: 71.24%
Epoch [12/30], Loss: 1.9148, Accuracy: 71.25%
Epoch [13/30], Loss: 1.7868, Accuracy: 74.83%
Epoch [14/30], Loss: 1.7671, Accuracy: 77.27%
Epoch [15/30], Loss: 1.6856, Accuracy: 77.85%
Epoch [16/30], Loss: 1.8328, Accuracy: 78.05%
Epoch [17/30], Loss: 1.7106, Accuracy: 78.76%
Epoch [18/30], Loss: 1.7385, Accuracy: 78.83%
Epoch [19/30], Loss: 1.7294, Accuracy: 79.20%
Epoch [20/30], Loss: 1.8120, Accuracy: 79.40%
Epoch [21/30], Loss: 1.7668, Accuracy: 79.51%
Epoch [22/30], Loss: 1.7833, Accuracy: 79.66%
Epoch [23/30], Loss: 1.6356, Accuracy: 79.88%
Epoch [24/30], Loss: 1.8148, Accuracy: 80.01%
Epoch [25/30], Loss: 1.7058, Accuracy: 80.02%
Epoch [26/30], Loss: 1.8033, Accuracy: 80.24%
Epoch [27/30], Loss: 1.7126, Accuracy: 80.59%
Epoch [28/30], Loss: 1.8605, Accuracy: 80.61%
Epoch [29/30], Loss: 1.8056, Accuracy: 80.52%
Epoch [30/30], Loss: 1.5952, Accuracy: 81.04%
```

Visualization from tensorboard:

```
%tensorboard --logdir /content/runs
```



We can see that when training the RBM for more epochs, the accuracy curve on test data keeps increasing from 33% at epoch #1 to 81% at epoch #30 and the loss also decreases from 2.2 at first to 1.5 at epoch #30. This means that the RBM model we trained have learned the distribution of the training data well, which can help to generate useful representation for a very simple classifer.