EE488 Machine Learning Basics and Practice

Mini-Assignment 2 보고서

20180155 김준범

Part 1. Implement DBSCAN algorithm using Numpy

Step 1: Implement DBSCAN from skeleton code

```
while np.any(self.cluster_labels == 0):
    p_idx = self.pick_arbitrary_point()
    if self.is_core_sample(p_idx):
        self.current_label += 1
        self.visit_all_successive_neighbors(p_idx)
    else:
        self.cluster_labels[p_idx] = -1

return self.cluster_labels
```

위 사진은 DBSCAN skeleton code의 fit_predict 함수에 있는 코드 부분이다. visit하지 않은 점이 없을 때까지 while문을 돌며 pick_arbitrary_point, is_core_sample, visit_all_successive_neighbors 함수를 실행한다. 아래에는 언급한 세 함수의 implementation에 대해 설명할 것이다.

```
def pick_arbitrary_point(self):
    '''
    Pick arbitrary point among points that are not visited so far (for next successive visiting).
    It returns an "index" of point("p_idx"), not a data point itself.
    '''
    # fill in the blank ------------------------------------
    indices = np.where(self.cluster_labels == 0)
    p_idx = np.random.choice(indices[0], 1)[0]
    # ------------------------------------------------------
    assert self.cluster_labels[p_idx] == 0  # sanity check
    return p_idx
```

위 사진은 pick_arbitrary_point 함수의 코드를 캡처한 것이다. pick_arbitrary_point 함수는 아직 방문하지 않은 점들 중 임의의 점의 index를 return해야 한다. 방문하지 않은 점들의 index에 해당하는 cluster_labels의 값이 0임을 활용하여 np.where안에 조건을 위와 같이 작성하면 아직 방문하지 않은 모든 점들의 index를 모은 numpy.ndarray를 return 받을 수 있다. 그 다음, np.random.choice 함수로 임의의 index를 선택해서 return하면 된다.

```
def is_core_sample(self, p_idx):
    '''
    parameter:
    p_idx: index of point

    Check whether the "p_idx" is a core sample or not.
    If it is, return True. Otherwise, return False.
    You can use "get_neighbors" method, which is defined below.
    You can define a core sample that has greater than or equal to
    '''
    # fill in the blank ------------------------------------

    neighbers = self.get_neighbors(p_idx)
    if len(neighbers) >= self.min_samples:
      return True
    else:
      return False
    # ------------------------------------------------------
```

위 사진은 is_core_sample 함수의 코드를 캡처한 것이다. is_core_sample 함수는 argument로 받은 p_idx에 대응되는 점이 core sample인지를 확인하여 true 또는 false를 return 한다. 우선 get_neighbors 함수(구현에 대한 설명은 아래에 있다)는 p_idx의 점과 eps 거리 안에 있는 모든 neighbor 점을 가진 set을 return한다. 따라서 해당 set의 length가 min_samples 보다 크거나 같으면 core sample이 맞으므로 true를 return하고 그렇지 않으면 false를 return한다.

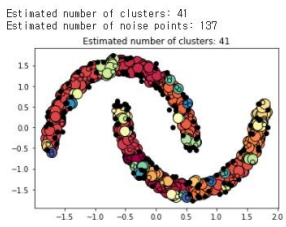
```
def get_neighbors(self, p_idx):
    '''
    parameter:
    p_idx: index of point

    It returns a "set of indices" of neighbors of "p_idx" point.
    l2 norm will be considered for computing distances.
    '''
    # fill in the blank ----------------------------------------
    neighbors = set()
    point = self.X[p_idx]
    for i in range(0,len(X)):
      if np.linalg.norm(self.X[i]-point,2) < self.eps:
        neighbors.add(i)
    return neighbors
    # ----------------------------------------------------------
```

위 사진은 get_neighbors 함수의 코드를 캡처한 것이다. neighbors란 빈 set을 선언하고 p_idx에 대응되는 점을 point라고 한 다음 모든 X의 점에 대한 for문을 돌면서 거리가 eps보다 작은, neighbor에 해당되는 점들의 index를 집합에 추가한다. 반복문이 끝나면 neighbors 집합을 return 한다.

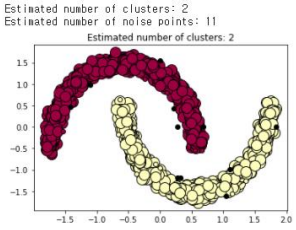
```
def visit_all_successive_neighbors(self, p_idx):
    '''
    parameter:
    p_idx: index of point

    Visit all the neighbors of "p_idx" as well as the neighbors of all tha
    Assign current cluster label everytime you visited. But you don't nee
    It returns nothing but modifies "self.cluster_labels" in-place when la
    '''
    all_neighbors_indices = {p_idx}
    while all_neighbors_indices:

    # fill in the blank ----------------------------------------
      index = all_neighbors_indices.pop()
      if index not in self.labeled_indices:
        self.cluster_labels[index] = self.current_label
        self.labeled_indices.add(index)
        neighbors = self.get_neighbors(index)
        if len(neighbors) >= self.min_samples:
          self.core_sample_indices_.append(index)
          all_neighbors_indices = all_neighbors_indices | neighbors
    # ----------------------------------------------------------
```

위 사진은 visit_all_successive_neighbors 함수의 코드를 캡처한 것이다. all_neighbors_indices는 visit할 점의 index를 모은 set 자료형으로, while문은 이 집합에 요소가 있는 동안 계속 반복되게 된다. 따라서 pop 메소드로 특정한 점을 set 뽑고 label된 index가 아니라면 cluster_labels의 값을 current_label로 바꿔 label을 설정해주고, labeled_indices에 index를 추가해 이제 label이 되었음을 표시해준다. 그 다음 pop한 점이 core sample인 경우는 core_sample_indices_에 넣고 neighbors들을 all_neighbors_indices에 추가하여 core sample의 neighbor들까지 visit할 수 있게 만들어 준다.

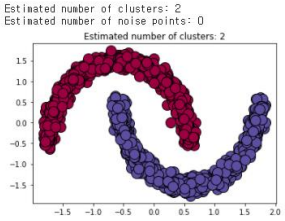
Step 2: Clustering with two different datasets



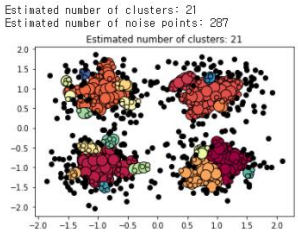
(eps, MinPts) = (0.05, 3)



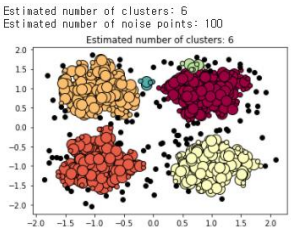
(eps, MinPts) = (0.1, 3)



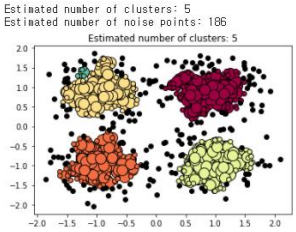
(eps, MinPts) = (0.2, 3)



(eps, MinPts) = (0.10, 5)



(eps, MinPts) = (0.15, 5)



(eps, MinPts) = (0.15, 8)

우선 noisy_moons dataset에서 (eps, MinPts) = (0.05, 3)일 때는 41개의 cluster가 만들어졌다. 이는 eps가 굉장히 작기 때문에 밀도가 조금이라도 낮은 곳은 core sample로 분류가 되지 않아 cluster가 이어지지 않기 때문이다. (eps, MinPts) = (0.1, 3)로 eps를 2배 늘리게 되면 더 많은 core sample이 생기게 되며 이전의 경우보다 cluster가 잘 이어질 수 있어 2개의 cluster가 만들어진다. (eps, MinPts) = (0.2, 3)로 더 eps를 늘리면 이전에 분류가 안되던 noise point 마저도 거리 안에 포함되며 noise point의 개수가 0으로 줄어들게 된다.

Make_blobs 함수로 만든 dataset에서 (eps, MinPts) = (0.1, 5)인 것은 noisy_moons의 첫 번째와 유사하게 eps가 굉장히 작아 21개의 cluster가 만들어진다. 마찬가지로 Eps를 0.15로 늘려주면 cluster의 개수가 6개로 줄어드는 것을 볼 수 있다. 여기서 MinPts를 8로 늘리면 밀도가 확실하게 높은 부분을 제외한 나머지 부분들은 모두 noise point로 분류되며 cluster가 5개로 줄고, noise point의 개수가 증가하게 된다.

Q1. What is the main difference between noise and border point?

Border point는 core point로부터 eps거리안에 있는 점이고 noise point는 core point도 아니고 core point로부터 eps거리 밖에 있는 점이다.

Q2. Let us slightly modify the algorithm (in fit_predict method) as follows: Then, what is the possible range of number of noise samples after the clustering is completed

Noise sample의 가능한 범위는 전체 sample의 수가 N일 때 $[0, \mathrm{N}\text{- core\_sample\_number}]$ 이다. 바뀐 algorithm에서는 임의로 선택한 점이 core sample이 아닌 경우, noise sample로 확정이 난다. 최악의 경우 모든 core sample이 아닌 점을 먼저 visit하게 되면 나중에 boundary point로 분류가 될 수 있던 점들 마저도 noise sample 분류가 된다. 이 경우, core sample을 제외한 모든 점들은 noise sample이 되어 N- core_sample_number 만큼 noise sample이 생기게 된다. 만약 운좋게 모든 sample이 인접하게 배치되어 있고 core sample들을 먼저 visit하게 되면 noise sample이 0개가 될 수 있다.

Part 2. Implement the Batchnormalization Layer in Deep Neural Network

Step2: Run DNN(Deep Neural Network) model without Batchnorm.

Step2-1: Implement DNN model **with six fully-connected layers(i.e. five hidden layers)**

```
    def forward(self, x):

        # fill in the blank ------------------------------------
        x = x.view(-1, 32*32*3)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        x = F.relu(self.fc4(x))
        x = F.relu(self.fc5(x))
        x = self.fc6(x)
        # ------------------------------------------------------

        return x
```

위 사진은 Batch Normalization이 없는 Deep Neural Network model의 forward process이다. Input x는 [256,3,32,32]로 batch size는 256이고, 각 data마다 3,32,32의 shape을 갖는다. 우선 view 메소드를 통해 [3,32,32]을 [3072]로 바꾸어 주고, 각 linear layer과 activation을 통과시켜 준다.

Step2-2: Implement train function for training

```
def train(model, data_loader, criterion, optimizer, n_epoch):
    model.train()
    for epoch in range(n_epoch):
        running_loss = 0
        for i, (images, labels) in enumerate(data_loader):
            images, labels = images.cuda(), labels.cuda()
            # fill in the blank ------------------------------------
            optimizer.zero_grad() # clear out the gradients to avoid accumulating with the gradients computed in previous step.
            outputs = model(images) # 1) Proceed forward pass
            loss = criterion(outputs, labels) # 2) Compute the loss by comparing the predictive outputs and the actual labels
            running_loss += loss
            loss.backward() # 3) Compute a gradient of a loss function by backpropagation
            optimizer.step()
            # ------------------------------------------------------

        print('Epoch {}, loss = {:.3f}'.format(epoch + 1, running_loss/len(data_loader)))
```

위 사진은 train 함수의 코드 부분을 캡처한 것이다. zero_grad 함수로 gradient를 초기화하고 forward process로 output을 계산한 뒤 loss를 계산한다. 이때 계산한 loss는 running_loss에 더하여 for문이 끝난 data_loader의 length 나누어 출력한다. backward 메소드를 통해 gradient를 계산하고 optimizer.step으로 weight들을 update한다.

Step2-3: Implement eval function for evaluation

```
def eval(model, data_loader):
    model.eval()
    total = 0
    correct = 0
    with torch.no_grad():
        for images, labels in data_loader:
            images, labels = images.cuda(), labels.cuda()
            # fill in the blank ------------------------------------
            outputs = model(images) # Obtain the outputs from model
            _, predicts = torch.max(outputs.data , 1) # Make a prediction
            total += labels.size(0)
            correct += (predicts==labels).sum().item() # Compute the number of correct predictions
            # ------------------------------------------------------
        accuracy = 100 * correct / total

    print('Test Accuracy: {}%'.format(accuracy))
```

위 사진은 eval 함수의 코드 부분을 캡처한 것이다. model에 input을 argument로 넣어 forward process를 진행하고 output에서 가장 큰 값으로 prediction을 한다. 이 predict 결과와 label을 비교하여 일치하는 것만큼 correct에 더 하고 전체 size로 나누어 accuracy를 계산한다.

Step2-4: Train the defined DNN model using train function.

아래는 Batch Normalization이 없는 Deep Neural Network model로 train을 하는 과정에서 매 epoch마다 출력된 결과이다.

```
Epoch 1, loss = 1.097
Epoch 2, loss = 0.910
Epoch 3, loss = 0.863
Epoch 4, loss = 0.823
Epoch 5, loss = 0.797
Epoch 6, loss = 0.775
Epoch 7, loss = 0.759
Epoch 8, loss = 0.744
Epoch 9, loss = 0.720
Epoch 10, loss = 0.719
Epoch 11, loss = 0.715
Epoch 12, loss = 0.700
Epoch 13, loss = 0.695
Epoch 14, loss = 0.685
Epoch 15, loss = 0.681
Epoch 16, loss = 0.661
Epoch 17, loss = 0.672
Epoch 18, loss = 0.657
Epoch 19, loss = 0.640
Epoch 20, loss = 0.629
```

Step2-5: Check the result

아래는 Batch Normalization이 없는 Deep Neural Network model을 train 한 뒤, test data로 eval 한 결과로 accuracy는 68.05%가 나왔다. 이 값을 Batch Normalization을 추가한 DNN model과 step4에서 비교할 것이다.

```
Test Accuracy: 68.05%
```

Step 3: Implement MyBacthNorm1d() class inheriting nn.Module

```
class MyBatchNorm1d(nn.Module):
    def __init__(self, num_features):
        super(MyBatchNorm1d, self).__init__()
        self.gamma = nn.Parameter(torch.ones(num_features)) # register the tensor as a parameter in this
        self.beta = nn.Parameter(torch.zeros(num_features))

        # fill in the blank ----------------------------------------
        self.t = 1
        self.mean = nn.Parameter(torch.zeros(num_features))
        self.var = nn.Parameter(torch.zeros(num_features))
        # ----------------------------------------------------------

    def forward(self, input):

        # fill in the blank ----------------------------------------
        if self.training:              # set to be True automatically when 'model.train()' is called
          if self.t == 1:
            self.mean = nn.Parameter(torch.mean(input, axis = 0))
            self.var = nn.Parameter(torch.var(input, axis = 0))
          else:
            self.mean = nn.Parameter(0.1 * torch.mean(input,axis = 0) + 0.9 * self.mean)
            self.var = nn.Parameter(0.1 * torch.var(input,axis = 0) + 0.9 * self.var)
          self.t = self.t + 1
          std = torch.sqrt(torch.var(input,axis = 0))
          x_norm = (input-torch.mean(input,axis = 0))/std
          output = self.gamma * x_norm + self.beta
        else:                          # if the 'model.eval()' was called
          x_norm = (input-self.mean)/torch.sqrt(self.var)
          output = self.gamma * x_norm + self.beta
        # ----------------------------------------------------------
        return output
```

위 사진은 Batch Normalization이 있는 Deep Neural Network model의 코드 부분이다. Model의 생성자에서 gamma, beta, t, mean, var이 선언되어 있다. t는 training 과정에서 평균과 분산의 moving average를 계산할 때 가장 처음인지 아닌지를 구별하기 위한 변수이다. Training에서 t==1(처음)인 경우는 mean과 var에 방금 계산한 평균과 분산이 저장되고, t!=1(처음이 아닌) 경우는 0.1*(현재 계산한 평균 또는 분산) + 0.9*(평균 또는 분산의 moving average)의 값을 mean과 var에 저장한다. Forward 과정에서 training을 하는 경우는 방금 설명한 mean, var의 moving average 계산을 하고 현재 계산한 평균과 표준 편차를 통해 input data의 normalization을 진행한다. Inference를 하는 경우는 training 과정에서 계산했던 평균과 분산의 moving average 값으로 input data의 normalization을 진행한다. Normalization을 진행한 다음은, gamma 값에 곱하고 beta를 더 해주어 output으로 return하는 동일한 과정을 거친다.

Step4: Implement DNN with MyBatchNorm1d()

Step4-1: Implement DNN_BatchNorm

```
class DNN_BatchNorm(nn.Module):

    def __init__(self):

        super(DNN_BatchNorm, self).__init__()
        # torch.nn.Linear(size of each input sample, size of each output sample)
        self.fc1 = nn.Linear(3072, 1800) # Input size should be 3072 (32*32*3) a
        self.norm1 = MyBatchNorm1d(num_features=1800)
        self.fc2 = nn.Linear(1800, 1200) # Input size should be the size of firs
        self.norm2 = MyBatchNorm1d(num_features=1200)
        self.fc3 = nn.Linear(1200, 1200)
        self.norm3 = MyBatchNorm1d(num_features=1200)
        self.fc4 = nn.Linear(1200, 640)
        self.norm4 = MyBatchNorm1d(num_features=640)
        self.fc5 = nn.Linear(640, 320)
        self.norm5 = MyBatchNorm1d(num_features=320)
        self.fc6 = nn.Linear(320, 4) # Input size should be the size of the prev

    def forward(self, x):

        # fill in the blank ------------------------------------
        x = x.view(-1, 32*32*3)
        x = F.relu(self.norm1(self.fc1(x)))
        x = F.relu(self.norm2(self.fc2(x)))
        x = F.relu(self.norm3(self.fc3(x)))
        x = F.relu(self.norm4(self.fc4(x)))
        x = F.relu(self.norm5(self.fc5(x)))
        x = self.fc6(x)

        # ------------------------------------------------------

        return x
```

위 사진은 Batch Normalization이 있는 Deep Neural Network model의 forward process이다. Step 2-1과 달리 norm1 ~ norm5로 batch normalization을 진행할 layer가 추가되었다. Input x는 여전히 [256,3,32,32]로 batch size는 256이고, 각 data마다 3,32,32의 shape을 갖는다. 따라서 view 메소드를 통해 [3,32,32]을 [3072]로 바꾸어 주고, 각 linear layer을 통과한 뒤, activation 전에 batch normalization을 진행한다.

Step4-2: Train the defined DNN_BatchNorm model using train function.

아래는 Batch Normalization이 있는 Deep Neural Network model로 train을 하는 과정에서 매 epoch마다 출력된 결과이다.

```
Epoch 1, loss = 1.010
Epoch 2, loss = 0.884
Epoch 3, loss = 0.837
Epoch 4, loss = 0.808
Epoch 5, loss = 0.787
Epoch 6, loss = 0.770
Epoch 7, loss = 0.751
Epoch 8, loss = 0.736
Epoch 9, loss = 0.715
Epoch 10, loss = 0.714
Epoch 11, loss = 0.703
Epoch 12, loss = 0.699
Epoch 13, loss = 0.685
Epoch 14, loss = 0.681
Epoch 15, loss = 0.668
Epoch 16, loss = 0.660
Epoch 17, loss = 0.664
Epoch 18, loss = 0.646
Epoch 19, loss = 0.638
Epoch 20, loss = 0.624
```

Step4-3: Check the result

아래는 Batch Normalization이 있는 Deep Neural Network model을 train한 뒤, test data로 eval한 결과이다. accuracy는 74.375%가 나왔다. Step2-5에서 batch normalization이 없을 때 test accuracy가 68.05%가 나왔고 이 값과 비교해보면 accuracy가 오른 것을 확인할 수 있다.

```
Test Accuracy: 74.375%
```