CS 559 Neural Networks – Homework 6

Zohair Hashmi | UIN: 668913771 | zhashm4@uic.edu

Part (a)

Denoising Autoencoder: The MNIST data is first augmented with arbitrary noise before training the model. The autoencoder, as seen in the resulting plots after every epoch, tend to create an output which is less noisy. The autoencoder addresses this by having an encoder- decoder structure as follows:

Encoder: The encoder comprises of three sections, namely Convolutional, Flatten & Linear Section.

- Firstly, the noisy MNIST data (size 28 x 28) goes through three convolutional layers with ReLU activation functions. The first layer takes an input with 1 channel (grayscale image) and produces 8 channels (size: 14 x 14). The second layer takes 8 channels and produces 16 channels (size: 7 x 7). The third layer takes 16 channels and produces 32 channels (size 3 x 3). Each convolutional layer is followed by a ReLU activation function.
- The output from the convolutional layers (size: 3 x 3 x 32) is flattened to a 1D tensor (size 128).
- The flattened output then goes through 2 linear layers. The first layer reduced the dimensionality to 128 with a ReLU activation function and then the second layer further reduces the dimensionality to the encoded_space_dim (which is the dimension of the encoded space set to 4 in our case).

Decoder: The decoder gets a 1D tensor (the encoded output) of size 4 from the Encoder and passes it through a Linear, Reshaping & Convolutional Section to retrieve the original denoised image.

- Firstly, the encoded output serves as an input to a linear layer which increases the dimensionality of the data to 128, followed by another linear layer which increases the dimensionality to match the size of the flattened tensor.
- The output from the Linear Layers is reshaped to the original image shape of the tensor, as it was after the third convolutional layer of the encoder (3 x 3 x 32 in the given code).
- The reshaped output then goes through three transposed Convolutional Layers to upsample the data. Each transposed convolutional layer is followed by batch normalization and ReLU activation. The final output is the reconstructed image which is denoised version of the input.

Therefore, this architecture learns to encode the input image into a lower-dimensional space and then decode it back to reconstruct the input. Reducing the noisy image to a lower-dimensional space helps in removing additional unnecessary information (such as the noise), serving as a denoising autoencoder.

Part (b)

Batch Normalization is a technique used to improve the training of deep neural networks by normalizing the input of each layer across a mini-batch. It was introduced to address the challenges associated with training deep networks, such as vanishing/exploding gradients.

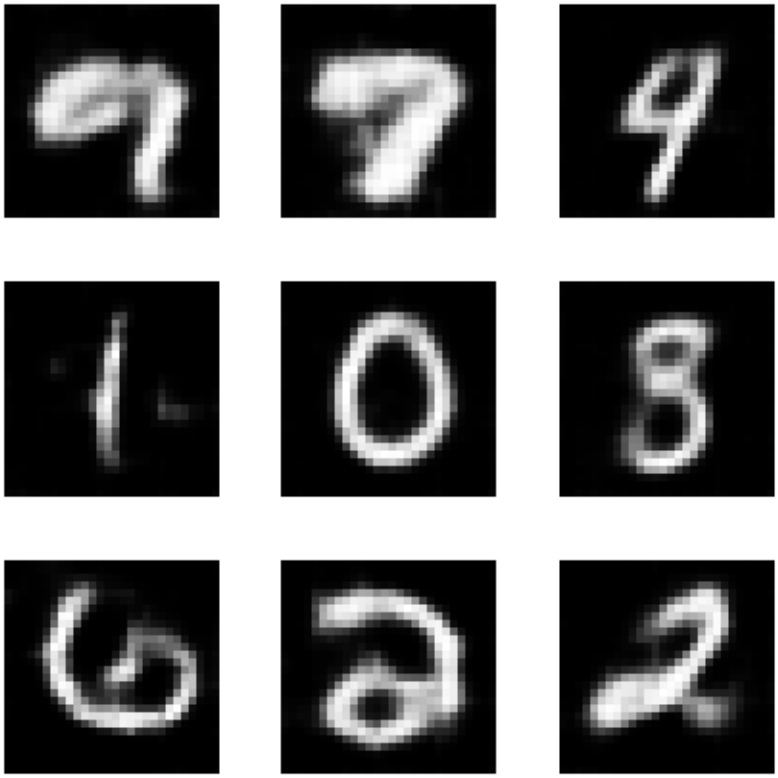
Batch Normalization normalizes the inputs to a layer by adjusting and scaling them to have zero mean and unit variance across the mini-batch. For each channel in a convolutional layer or neuron in a fully connected layer, the mean and standard deviation is calculated across the mini-batch. The input is then normalized by subtracting the mean and dividing by the standard deviation, and learnable parameters such as scale and shift are introduced to allow the network to adapt to the new normalized values.

For the batch normalization example in the given code, The number 16 in `nn.BatchNorm2d(16)` represents the number of channels in the input to the batch normalization layer. In the context of convolutional layers, each channel is like a feature map produced by a convolutional filter.

Part (c)

Code attached at the end of the report.

Results



Part (d)

Code attached at the end of the report.

Results

- **Accuracy before reassignment:** 11.83 %
- **Accuracy after reassignment:** 66.69 %

Index Reassignment Algorithm

The clusters are assigned according to the following steps of the algorithm:

1) **get_encoded_data (encoder, dataloader)**
    - Retrieve the encoded data using the trained encoder module on the training dataset. The encoded data is of size 48000 x 4.
2) **get_centroids (encoded_data, labels)**
    - From the given encoded dataset (size: 48000 x 4) and the respective labels, the average position of each of the given labels (0, 1, 2, …, 9) is calculated as the mean centroid position.
3) **get_kmeans_clusters (encoded_train_data, k)**
    - The k-means algorithm is used to train an unsupervised model using the encoded training dataset (without label information). The algorithm is set to identify k=10 clusters for each given label (0,

1, 2, …, 9). The function created for training k-means algorithm returns the centroids of each of the 10 clusters and their respective labels.

4) **reassign_clusters (actual_centroids, pred_centroids, actual_labels, pred_labels)**
   - The labels returned by the k-means clustering algorithm need to be reassigned since the algorithm is agnostic to the actual labels. This can be confirmed when we calculate the accuracy of the predicted labels in comparison to the actual labels.
   - To perform this reassignment, the distance from each predicted centroid to each actual centroid is calculated, and the closest actual centroid is mapped to the predicted label.
   $$\boldsymbol{Distance} = \sum_{\boldsymbol{k=1}}^{\boldsymbol{n}} (\boldsymbol{predCentroids}[\boldsymbol{i}] - \boldsymbol{actualCentroids}[\boldsymbol{j}])^{\boldsymbol{2}}$$
   - From the calculated distances, each predicted cluster centroid is mapped to the closest actual centroid. The mapping is performed for each predicted cluster centroid.
   $$\boldsymbol{labelMapping}[\boldsymbol{i}] = \boldsymbol{np.argmin}(\boldsymbol{distancesDict}[\boldsymbol{i}])$$
   - The resultant mapping is as follows: `label mapping (predicted label to actual label): {0: 2, 1: 1, 2: 9, 3: 2, 4: 7, 5: 6, 6: 0, 7: 4, 8: 5, 9: 3}`. As we can see two predicted labels (0 and 3) have the same actual label (2) mapped to it.
   - To improve this, both labels are tested for the next closest centroid that has not been assigned yet (which is 8 in this case). Hence, label 8 is assigned to either 0 or 2 based on which cluster is closest.
   - This reassignment of cluster labels leads to a one-to-one mapping of predicted to actual, which is used to modify the resulting predictions of the k-means algorithm on the given dataset as per the acquired mapping. `label mapping: {0: 2, 1: 1, 2: 9, 3: 8, 4: 7, 5: 6, 6: 0, 7: 4, 8: 5, 9: 3}`

Part (e)

Code uploaded to Box Folder.

Python Scripts

Part (C) – Generating 9 random images.

```
seed = 1
torch.manual_seed(seed)

z = torch.randn(9, d).to(device) # Random tensor of size 9 x d (9 x 4)

x_hat = decoder(z) # Feed the tensor to the decoder

# Plot the reconstructed images
plt.figure(figsize=(10,10))
for i in range(9):
    plt.subplot(3, 3, i + 1)
    plt.imshow(x_hat[i].cpu().detach().numpy().squeeze(), cmap='gist_gray')
    plt.axis('off')

plt.subplots_adjust(left=0.1,
                    bottom=0.1,
                    right=0.7,
                    top=0.9,
                    wspace=0.3,
                    hspace=0.3)
plt.show()
```

Part (D) – Reassignment Algorithm.

```
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import StandardScaler

def get_encoded_data(encoder, dataloader):
    encoded_data = []
    labels = []

    for image_batch, label_batch in dataloader:
        image_batch = image_batch.to(device)
        encoded_output = encoder(image_batch)
        encoded_output = encoded_output.cpu().detach().numpy()

        label_batch = label_batch.to(device)
        label_batch = label_batch.cpu().detach().numpy()

        for item in encoded_output:
            item = item.tolist()
```

```
            encoded_data.append(item)

        for item in label_batch:
            item = item.tolist()
            labels.append(item)

    return encoded_data, labels

def get_centroids(encoded_data, labels):
    #for each label, calculate the centroid
    centroids = []
    for i in range(10):
        #get all the data with the same label
        data = []
        for j in range(len(labels)):
            if labels[j] == i:
                data.append(encoded_data[j])
        data = np.array(data)
        #calculate the centroid
        centroid = np.mean(data, axis=0)
        centroids.append(list(centroid))

    return np.array(centroids)


def get_kmeans_clusters(encoded_train_data, k):
    kmeans = KMeans(n_clusters=k, random_state=0, max_iter=10000, n_init=20,init='k-
means++', algorithm='full')
    kmeans.fit(encoded_train_data)
    centroids = kmeans.cluster_centers_
    predicted_labels = kmeans.labels_
    return centroids, predicted_labels

def reassign_clusters(actual_centroids, pred_centroids, actual_labels, pred_labels):

    # Accuracy of the clustering : actual labels vs predicted labels
    print('Accuracy before reassignment: ', np.round(accuracy_score(actual_labels,
pred_labels)*100, 2), '%')

    label_mapping = {}
    distances_dict = {}

    for i in range(len(pred_centroids)):
        distances = []
        for j in range(len(actual_centroids)):
            # Distance between the predicted centroid and the actual centroid
            distance = np.linalg.norm(pred_centroids[i]-actual_centroids[j])
            distances.append(distance)
```

```
        distances_dict[i] = distances

    # create label mapping
    for i in range(len(distances_dict)):
        index = np.argmin(distances_dict[i])
        label_mapping[i] = index

    # in label_mapping, check for the keys that have the same value, if they have the same
value update them with the missing value
    for key1 in label_mapping:
        for key2 in label_mapping:
            if key1 != key2 and label_mapping[key1] == label_mapping[key2]:
                temp = label_mapping[key1]
                label_mapping[key1] = -1
                label_mapping[key2] = -1

                for i in range(len(distances_dict)):
                    if i not in label_mapping.values():
                        if distances_dict[key1][i] < distances_dict[key2][i]:
                            label_mapping[key1] = i
                            label_mapping[key2] = temp
                        else:
                            label_mapping[key2] = i
                            label_mapping[key1] = temp

    print ('label mapping: ', label_mapping)

    # Reassignment according to the label mapping
    new_pred_labels = []
    for label in pred_labels:
        new_pred_labels.append(label_mapping[label])

    # Accuracy of the clustering: new predicted labels vs actual labels
    print('Accuracy after reassignment: ', np.round(accuracy_score(actual_labels,
new_pred_labels)*100, 2), '%')

    return new_pred_labels

encoded_train_data, actual_train_labels = get_encoded_data(encoder, train_loader)
actual_centroids = get_centroids(encoded_train_data, actual_train_labels)
pred_centroids, predicted_labels = get_kmeans_clusters(encoded_train_data, k=10)
reassigned_pred_labels = reassign_clusters(actual_centroids, pred_centroids,
actual_train_labels, predicted_labels)
```