Programming Problem 1

Structure of Model

VAE Model

The decoder is composed of fully connected layers that gradually increase in size from the latent dimension to the output dimension. ReLu activation function is used in between the fully connected layers with a Tanh activation for the last layer. The entire structure is seen in the screenshot below.

```
class Decoder(nn.Module):
    def __init__(self, latent_dim, output_dim):
        super(Decoder, self).__init__()
        # build your model here
        # your output should be of dim (batch_size, output_dim)
        # you can use tanh() as the activation for the last layer
        # since y coord of airfoils range from -1 to 1
        mid_dim = int((latent_dim+output_dim)/2)
        quarter_dim = int((latent_dim +mid_dim)/2)
        three_quarter_dim = int((output_dim+ mid_dim)/2)

        self.fc1 = nn.Linear(latent_dim,quarter_dim)
        self.fc12 = nn.Linear(quarter_dim,mid_dim)
        self.fc23 = nn.Linear(mid_dim,three_quarter_dim)
        self.fc2 = nn.Linear(three_quarter_dim,output_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc12(x)
        x = self.relu(x)
        x = self.fc23(x)
        x = self.relu(x)
        x = self.fc2(x)
        return self.tanh(x)
```

Similarly, the encoder is composed of fully connected layers that decrease in size from the input dimension to the latent dimension. The first layer has a Tanh activation and all the other layers have a ReLu activation.

```python
class Encoder(nn.Module):
    def __init__(self, input_dim, latent_dim):
        super(Encoder, self).__init__()
        # build your model here
        # your model should output a predicted mean and a predicted
        # both should be of dim (batch_size, latent_dim)
        mid_dim = int((input_dim+latent_dim)/2)
        quarter_dim = int((latent_dim +mid_dim)/2)
        three_quarter_dim = int((input_dim+ mid_dim)/2)

        self.fc1 = nn.Linear(input_dim,three_quarter_dim)
        self.fc2 = nn.Linear(three_quarter_dim,mid_dim)
        self.fc3 = nn.Linear(mid_dim,quarter_dim)
        #self.fc4 = nn.Linear(mid_dim,mid_dim)
        self.fc_mean = nn.Linear(quarter_dim,latent_dim)
        self.fc_logvar = nn.Linear(quarter_dim,latent_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()

    def forward(self, x):
        # define your feedforward pass
        x = self.fc1(x)
        x = self.tanh(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        # x = self.fc4(x)
        # x = self.relu(x)
        return self.fc_mean(x),self.fc_logvar(x)
```

GAN Model

The Gan Generator is connected using fully connected layers. The size of the layers increases from the latent dimension to the airfoil dimension. The layers all have a ReLu activation function except for the last layer which has a Tanh activation function.

```
class Generator(nn.Module):
    def __init__(self, latent_dim, airfoil_dim):
        super(Generator, self).__init__()
        # build your model here
        # your output should be of dim (batch_size, airfoil_dim)
        # you can use tanh() as the activation for the last layer
        # since y coord of airfoils range from -1 to 1
        mid_dim = int((latent_dim+airfoil_dim)/2)
        quarter_dim = int((latent_dim +mid_dim)/2)
        three_quarter_dim = int((airfoil_dim+ mid_dim)/2)
        self.fc1 = nn.Linear(latent_dim,quarter_dim)
        self.fc2 = nn.Linear(quarter_dim,mid_dim)
        self.fc3 = nn.Linear(mid_dim,three_quarter_dim)
        self.fc4 = nn.Linear(three_quarter_dim,airfoil_dim)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()


    def forward(self, x):
        # define your feedforward pass
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.fc3(x)
        x = self.relu(x)
        x = self.fc4(x)
        return self.tanh(x)
```

Finally, the discriminator has a similar structure to the generator. It decreases in size from the input dimension to a boolean output to predict whether the generated image is real or fake. The discriminator has relu activation function for its fully connected layers except for the last layer which has a sigmoid activation. Also, to ensure that the discriminator does not overpower the generator, I have added a dropout layer with probability 0.2 before the last linear layer.

```python
class Discriminator(nn.Module):
    def __init__(self, input_dim):
        super(Discriminator, self).__init__()
        # build your model here
        # your output should be of dim (batch_size, 1)
        # since discriminator is a binary classifier
        self.model = nn.Sequential(
            nn.Linear(input_dim,int(3*input_dim/4)),
            nn.ReLU(),
            nn.Linear(int(3*input_dim/4),int(input_dim/2)),
            nn.ReLU(),
            nn.Linear(int(input_dim/2),int(input_dim/4)),
            nn.ReLU(),
            nn.Dropout(0.2),
            nn.Linear(int(input_dim/4),1),
            nn.Sigmoid(),
        )

    def forward(self, x):
        # define your feedforward pass
        x = self.model(x)
        return x
```

Hyperparameter Tuning

VAE

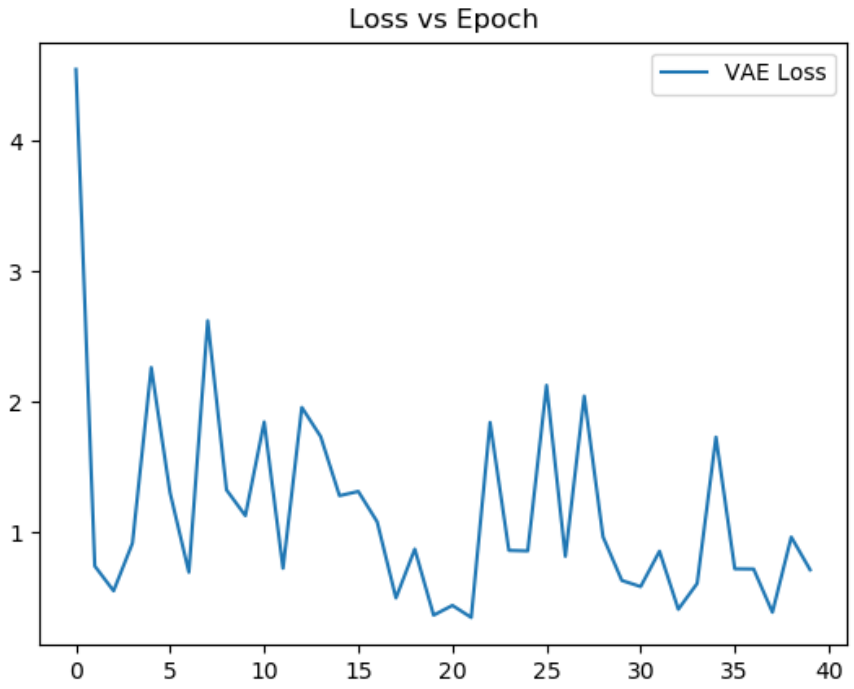
For the VAE I tweaked around with the loss function in particular with the ratio of the reconstruction and the KL divergence loss. After careful tweaking, I settled with a factor of 0.008. For the learning rate I tried multiple rates and then settled with a rate of 0.003. Higher learning rates resulted in better reconstruction but made the synthesized airfoils worse. I also tried to use a learning rate scheduler (reduce lr on plateau) but those were not giving good results. Finally I also tweaked with the number of epochs. Too high of epoch numbers (hundreds) was resulting in overfitting and most of the airfoils were very similar to each other. Hence I found a good result for the training with about 40 epochs. The VAE uses the adam optimizer.

GAN

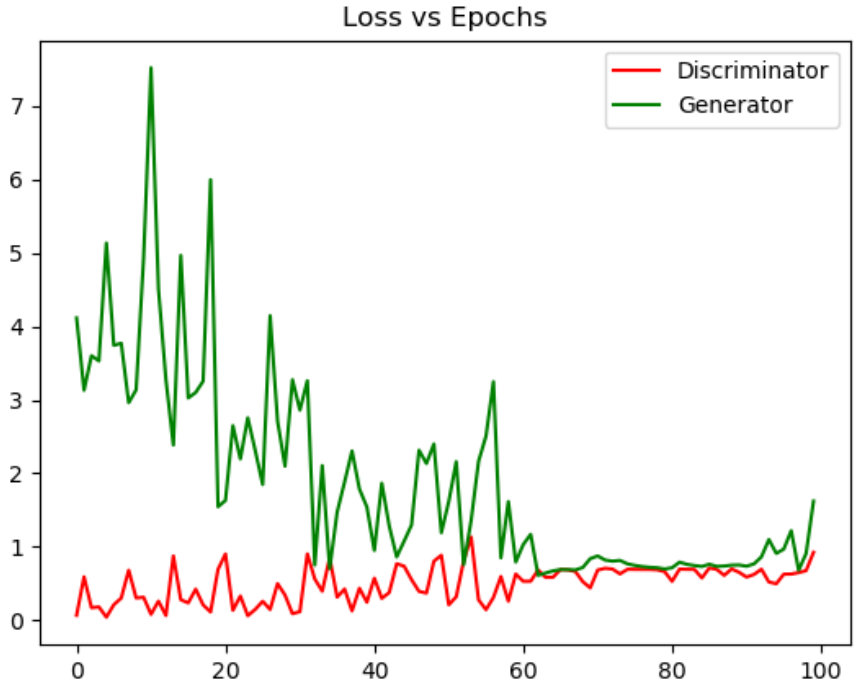
For the GAN I tweaked around with the hyperparameters for quite a while to get good results. For the learning rate I tried multiple rates and then settled with a rate of 0.001 for the discriminator and 0.0002 for the generator. During my trials I found that having the discriminator learning rate to be slightly higher than the generator was giving good results. This is likely because a good discriminator forces the generator to learn quickly providing good results. I also tried to use a learning rate scheduler (reduce lr on plateau) but those were not giving good results. Finally I also tweaked with the number of epochs. Too high of epoch numbers (~500) was resulting in overfitting and most of the airfoils were very similar to each other. Hence I found a good result for the training with about 100 epochs. Furthermore to ensure that the gan would not fail to converge I have added dropout layers in the discriminator to ensure that the discriminator and the generator learn at a similar rate. Both models use the Adam optimizer.

Loss vs Epoch

VAE

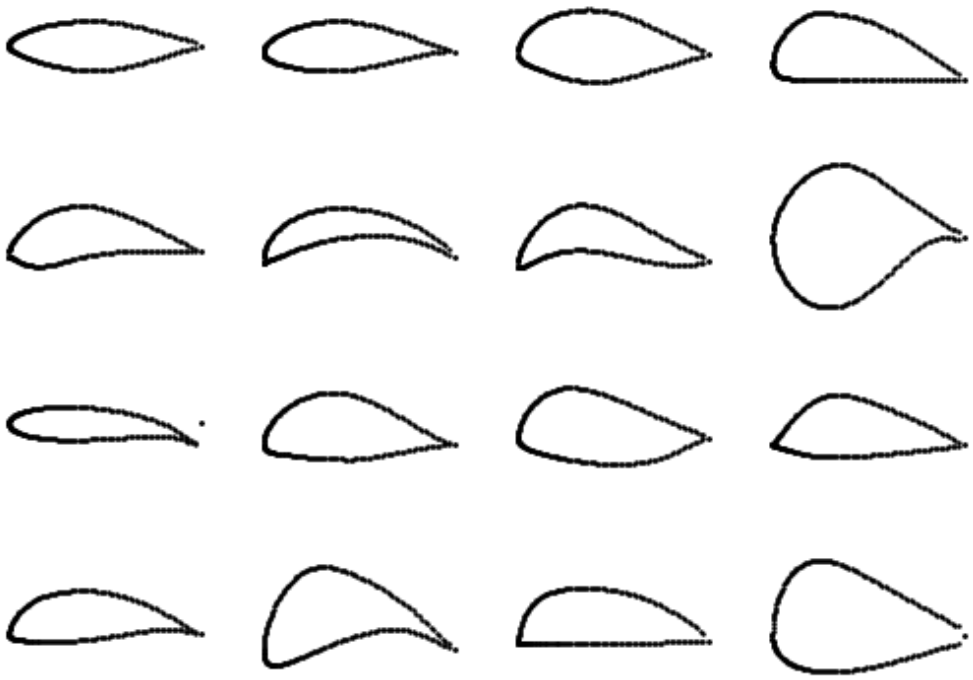


GAN

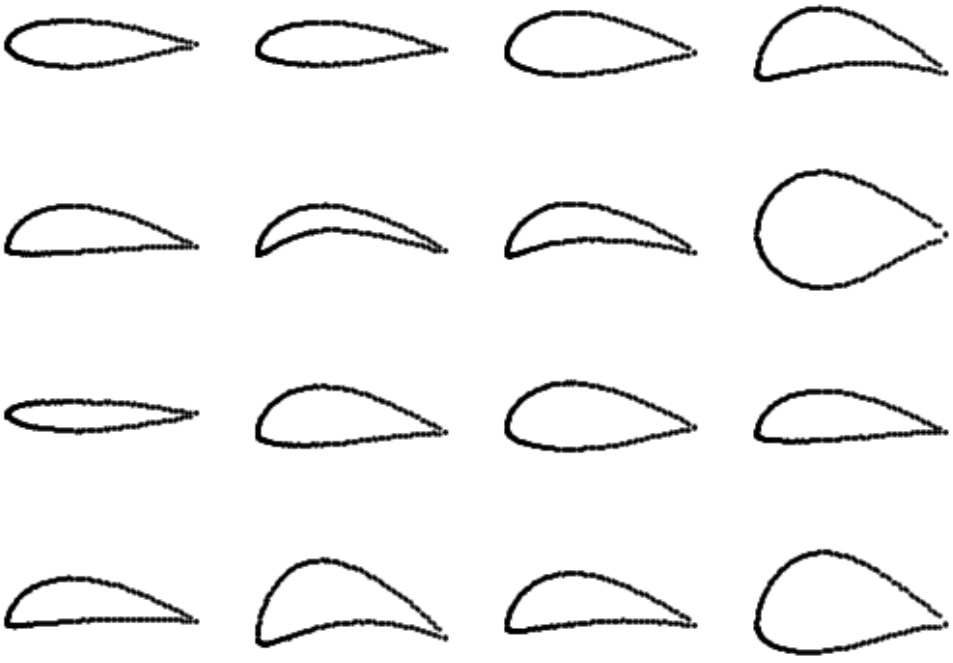


VAE Visualization

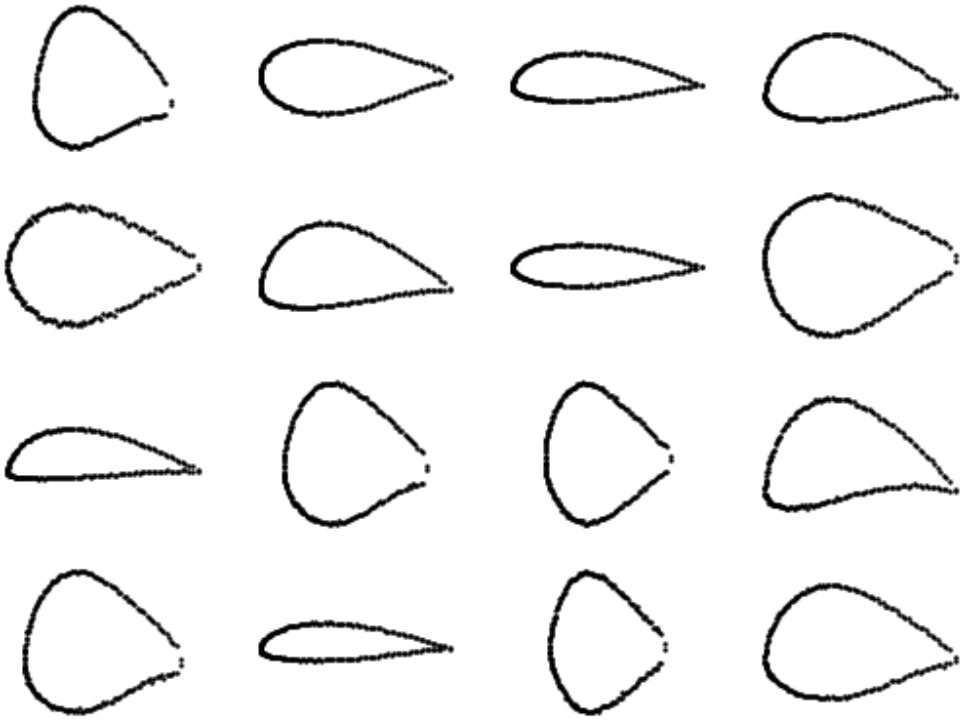
Real Airfoils



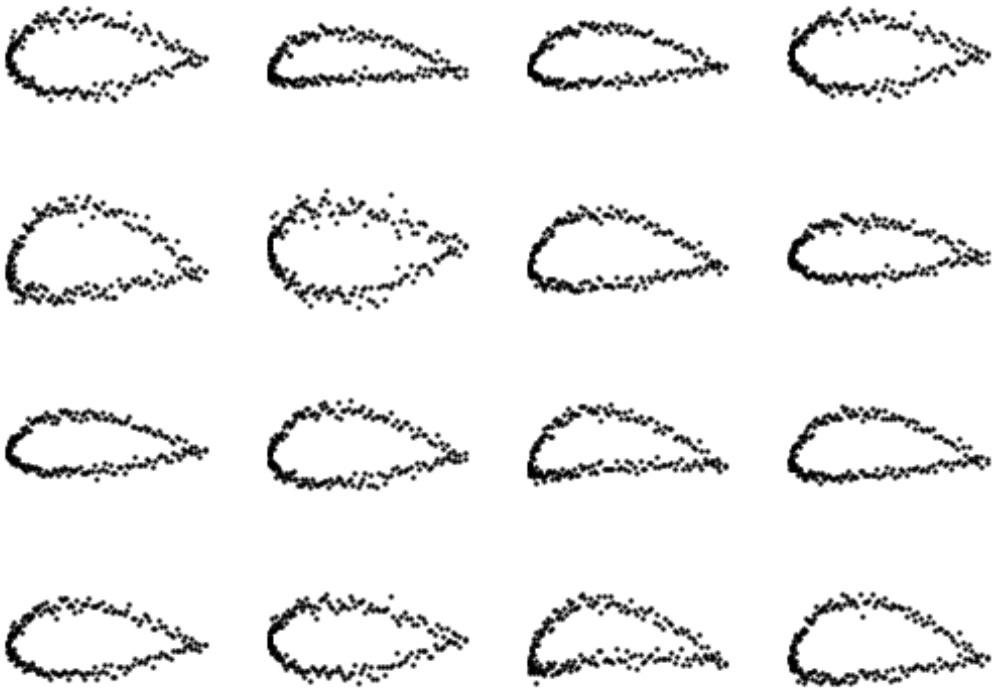
Reconstructed Airfoils



Synthesized airfoils from noise



GAN Synthesis



Comparison of Airfoils between VAE and GAN

The VAE is trained on real data compared to the GAN which always gets noise as its input. Hence it is natural that the VAE can synthesize and reconstruct airfoils much more accurately. This can be seen in the outputs for the GAN which are noisier and there are significantly more outliers in the airfoil points for the GAN compared to the VAE. Because the GAN is trained from random noise it is satisfactory for the generator to generate an airfoil and hence a large number of the generated airfoils have a similar shape.