Student Name: Arash Sajjadi
Student ID: 400422096

Artificial Neural Network (ANN)
Assignment IV

Question 1

A variational autoencoder (VAE) is a type of autoencoder that is able to generate new data points by sampling from a latent space. An associative autoencoder, on the other hand, is not able to generate new data points because it is designed to reconstruct the input data as closely as possible rather than generating new data.

The VAE architecture allows it to generate new data points because it is trained to learn the distribution of the data, rather than just reconstructing the input data. It does this by learning a set of latent variables, which are lower-dimensional representations of the data. The VAE then learns to map these latent variables to the original data points and vice versa.

To generate new data points, the VAE can sample from this latent space and use the decoder to map the latent variables back to the original data space. Because the VAE has learned the distribution of the data, the generated data points will be similar to the original data points.

In contrast, an associative autoencoder is not able to generate new data points because it is only trained to reconstruct the input data. It does not learn the distribution of the data and, therefore cannot generate new data points by sampling from latent space.

Question 2

- The KL-divergence term in the loss function of a Variational Autoencoder (VAE) is used to ensure that the approximate posterior distribution of the latent variables (often called the "encoding") is similar to a prior distribution, which is often chosen to be a standard normal distribution. The KL-divergence is a measure of the difference between two probability distributions, and in the context of a VAE, it is used to encourage the approximate posterior to be similar to the prior. This helps to ensure that the encoding is not too far away from the prior and that the model has a good trade-off between reconstruction error and the regularization provided by the KL divergence term.
- Modeling $p_\theta(z)$ and $q_\phi(z|x^{(i)})$ as normal distributions with diagonal covariance matrices in the loss function of a VAE has several advantages:
  1. **Computational efficiency**: Diagonal covariance matrices are computationally more efficient to work with than full covariance matrices, as they have fewer parameters to estimate.
  2. **Independence assumption**: By assuming that the variables in the latent space are independent, a diagonal covariance matrix enforces a factorization of the joint distribution, which can make it easier to learn the dependencies between the variables in the data.

3. **Stable optimization**: The KL divergence term in the loss function of VAE can be intractable and unstable to optimize, assuming diagonal covariance matrices for $q_\phi(z|x^{(i)})$ can help to make the loss function more stable during the optimization process.

4. **Better generalization**: Assumptions of independence between latent variables can lead to a better generalization performance as they reduce the risk of overfitting.

5. **Scalability**: VAE with diagonal covariance matrices are more scalable as the number of dimensions in the latent space increases, as the computational complexity of the model is linear with the number of dimensions.

However, it's important to notice that this assumption might be too strong in some cases and it can lead to suboptimal results, in those cases you can use full covariance matrices or other distributions to model your latent variables.

- The first term in the loss function of a Variational Autoencoder (VAE) is often referred to as the "reconstruction loss" or "reconstruction error". It measures the difference between the original input $x(i)$ and the reconstructions $x'(i)$ generated by the decoder. The reconstruction loss is usually computed as the negative log-likelihood of the data, given the encoder and decoder parameters. In mathematical terms, the reconstruction loss can be written as:

$$-L(x^{(i)}) = -\log p_\phi(x^{(i)}|z)$$

  This term is the main objective of the VAE. The VAE is trained to minimize this term with respect to the encoder and decoder parameters θ and ϕ. This encourages the VAE to produce reconstructions that are similar to the original inputs.

  The effect of this term on the latent space is that it forces the VAE to learn a meaningful and compact representation of the data in the latent space. The VAE learns to map the data to the latent space in such a way that the data can be reconstructed with minimal loss. In other words, the VAE learns a good trade-off between the reconstruction error and the regularization provided by the KL divergence term, which in turn leads to a meaningful and compact representation of the data in the latent space.

Question 3

According to the agreement with the assistant professor of the course, it was decided to use equation $\frac{x_1 - x_2}{2}$ instead of the average for this problem.

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
```

[2]:
```
# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(device)
# Hyper-parameters
num_epochs = 1201
batch_size = 192
learning_rate = 0.0005
```

```
cpu
```

[3]:
```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
test_dataset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True, transform=transform)
train_loader = torch.utils.data.DataLoader(train_dataset,batch_size=50000 ,shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                         shuffle=False)

def imshow(img):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

def imshowD(img,img2):
    img = img / 2 + 0.5  # unnormalize
    npimg = img.numpy()
    img2 = img2 / 2 + 0.5  # unnormalize
    npimg2 = img2.numpy()
    f, axarr = plt.subplots(1,2)
    axarr[0].imshow(np.transpose(npimg, (1, 2, 0)))
    axarr[1].imshow(np.transpose(npimg2, (1, 2, 0)))
    axarr[0].set_title("First Image")
    axarr[1].set_title("Second")
    plt.show()

dataiter = iter(train_loader)
images, labels = next(dataiter)
train_dataset=images[:1000]
valid_dataset=images[1000:2000]
train_loader = torch.utils.data.DataLoader(train_dataset,batch_size=batch_size ,shuffle=True)
valid_loader = torch.utils.data.DataLoader(valid_dataset,batch_size=1000,shuffle=True)
dataiter = iter(train_loader)
images = next(dataiter)
dataiter_v = iter(valid_loader)
images_v = next(dataiter_v)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

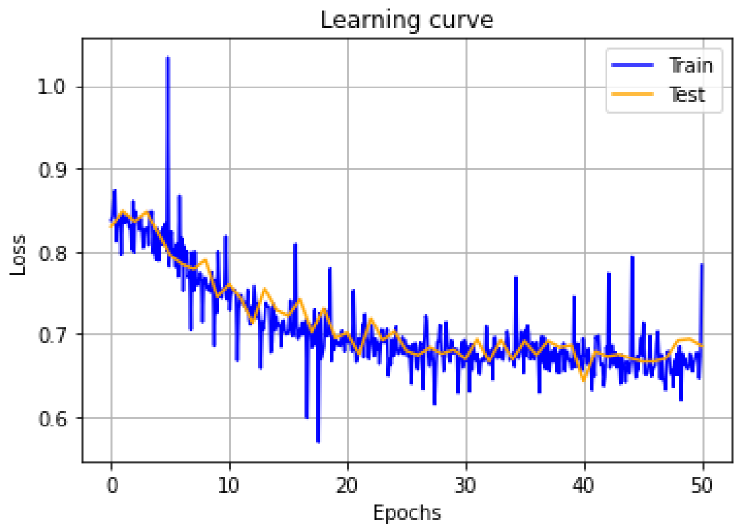
[4]:
```
def AVG(img1,img2):
  return (0.5*(img1-img2)+1)/2

def plot_loss(Loss,Loss_v,ep):
  plt.plot(np.linspace(0,ep,len(Loss)),Loss,label="Train",c="blue")
  plt.plot(np.linspace(0,ep,len(Loss_v)),Loss_v,label="Test",c="orange")
  plt.legend()
  plt.grid()
  plt.xlabel('Epochs')
  plt.ylabel('Loss')
  plt.title('Learning curve')
  plt.show()
```

[5]:
```
class Autoencoder1(nn.Module):
    def __init__(self):
        super().__init__()
        # N, 3, 32, 32
        self.encoder = nn.Sequential(
            nn.Conv2d(3, 32, 4, stride=2, padding=1), # -> N, 16, 16, 16
            nn.LeakyReLU(),
            nn.Conv2d(32, 64, 4, stride=1, padding=1), # -> N, 32, 8, 8
            nn.LeakyReLU(),
            nn.Conv2d(64, 64, 4, stride=1, padding=1),
            nn.LeakyReLU(),
            nn.Conv2d(64, 64, 4, stride=2, padding=1),
            nn.LeakyReLU(),
            nn.Conv2d(64, 128, 6) # -> N, 64, 1, 1
        )

        self.decoder1 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 2, stride=1), # -> N, 32, 7, 7
            nn.LeakyReLU(),
            nn.ConvTranspose2d(64, 64, 4, stride=2, padding=1, output_padding=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.Dropout2d(p=0.3,inplace=False),
            nn.ConvTranspose2d(64, 64, 4, stride=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.ConvTranspose2d(64, 32, 4, stride=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.ConvTranspose2d(32, 3, 4, stride=2, padding=1, output_padding=1), # N, 1, 28, 28  (N,1,27,27)
            nn.UpsamplingNearest2d(scale_factor=32/27),
            nn.Tanh()
        )

        self.decoder2 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 2, stride=1), # -> N, 32, 7, 7
            nn.LeakyReLU(),
            nn.ConvTranspose2d(64, 64, 4, stride=2, padding=1, output_padding=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.ConvTranspose2d(64, 64, 4, stride=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.Dropout2d(p=0.3,inplace=False),
            nn.ConvTranspose2d(64, 32, 4, stride=1), # N, 16, 14, 14 (N,16,13,13 without output_padding)
            nn.LeakyReLU(),
            nn.ConvTranspose2d(32, 3, 4, stride=2, padding=1, output_padding=1), # N, 1, 28, 28  (N,1,27,27)
            nn.UpsamplingNearest2d(scale_factor=32/27),
            nn.Tanh()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded1 = self.decoder1(encoded)
        decoded2 = self.decoder2(encoded)
        return decoded1,decoded2
```

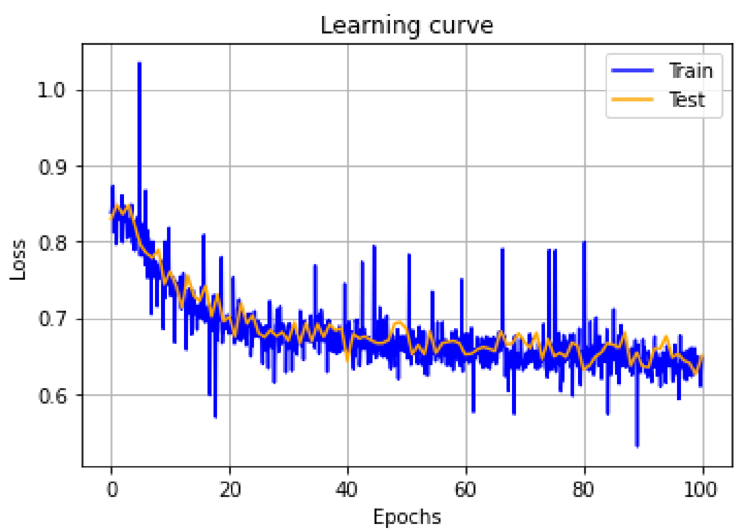
[6]:
```
model = Autoencoder1()

criterion = nn.L1Loss()
optimizer = torch.optim.Adam(model.parameters(),lr=learning_rate)
```

[7]:
```
# Point to training loop video
Loss = []
Loss_valid = []
for epoch in range(num_epochs):
    c=0
    for img in train_loader:
      local_batch_size=img.shape[0]
      rand_perm=torch.randperm(local_batch_size)
      img=img[rand_perm]
      avgimg=AVG(img[:int(local_batch_size/2)],img[int(local_batch_size/2):local_batch_size])
      recon1,recon2 = model(avgimg)
      loss1 = criterion(recon1, img[:int(local_batch_size/2)])
      loss2 = criterion(recon2,img[int(local_batch_size/2):local_batch_size])
      loss=loss1+loss2
      Loss.append(float(loss.detach().cpu().resolve_conj().resolve_neg().numpy()))
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

      c=c+1
      if c%8==0:
        with torch.no_grad():
          for img_v in valid_loader:
            local_batch_size_v=img_v.shape[0]
            rand_perm_valid=torch.randperm(local_batch_size_v)
            img_v=img_v[rand_perm_valid]
            img_v=img_v[:256]

            local_batch_size_v=img_v.shape[0]
            rand_perm_valid=torch.randperm(local_batch_size_v)
            img_v=img_v[rand_perm_valid]

            avgimg_v=AVG(img_v[:int(local_batch_size_v/2)],img_v[int(local_batch_size_v/2):
↪local_batch_size_v])
            recon1_v,recon2_v = model(avgimg_v)
            loss1_v = criterion(recon1_v, img_v[:int(local_batch_size_v/2)])
            loss2_v = criterion(recon2_v,img_v[int(local_batch_size_v/2):local_batch_size_v])
            loss_v=loss1_v+loss2_v
            Loss_valid.append(float(loss_v.detach().cpu().resolve_conj().resolve_neg().numpy()))
            break

    if epoch%50==0:
      print(f'Epoch:{epoch} \t\t Train Loss:{Loss[len(Loss)-1]:.5f}\t\t Test␣
↪Loss{Loss_valid[len(Loss_valid)-1]:.5f}\n')
      if epoch!=0:
        plot_loss(Loss,Loss_valid,epoch)
```

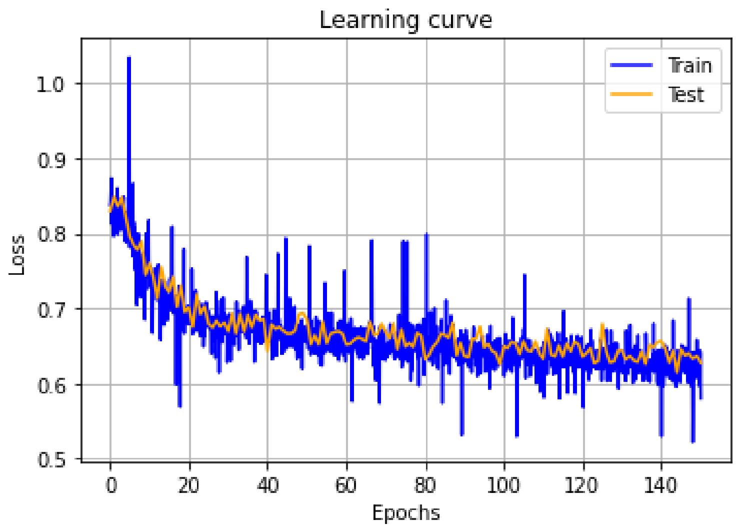
```
Epoch:0          Train Loss:0.79621          Test Loss0.82947

Epoch:50         Train Loss:0.78280          Test Loss0.68533
```



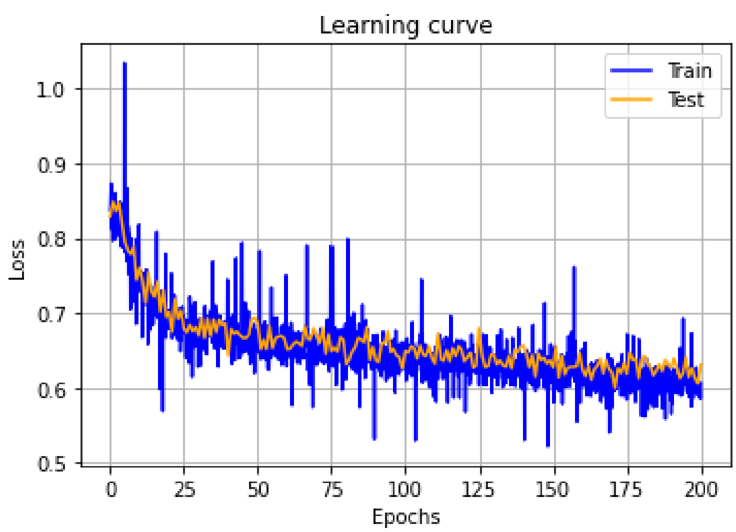
`Epoch:100 Train Loss:0.64710 Test Loss0.64971`



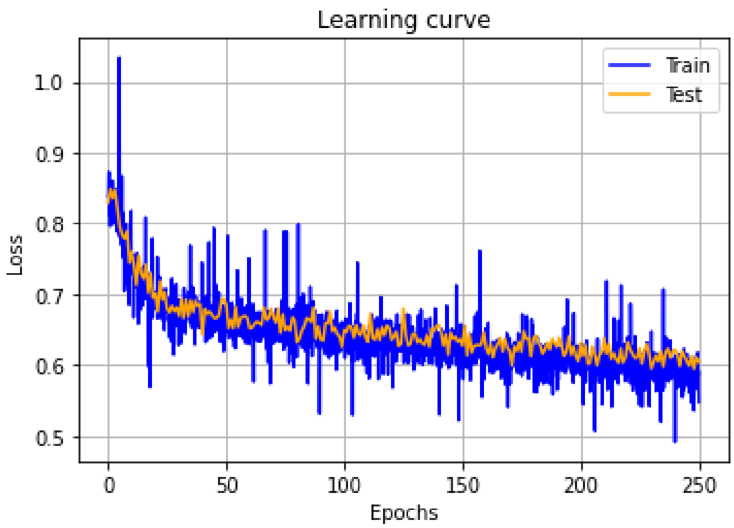
`Epoch:150 Train Loss:0.58020 Test Loss0.62745`



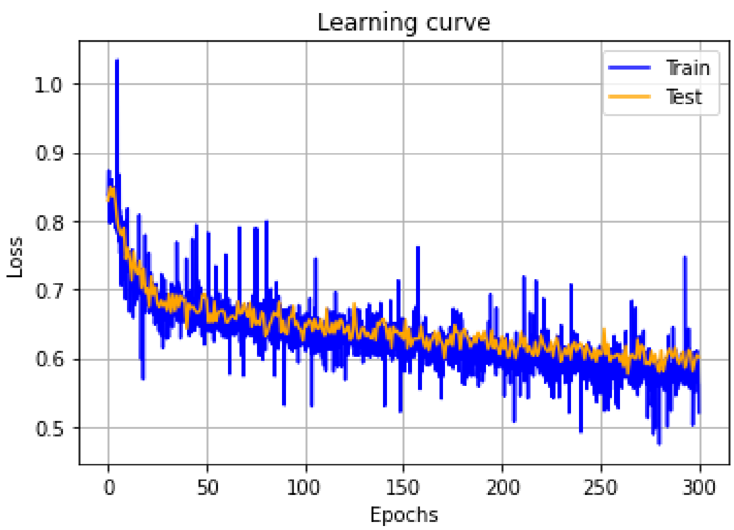
`Epoch:200            Train Loss:0.60641           Test Loss0.63124`



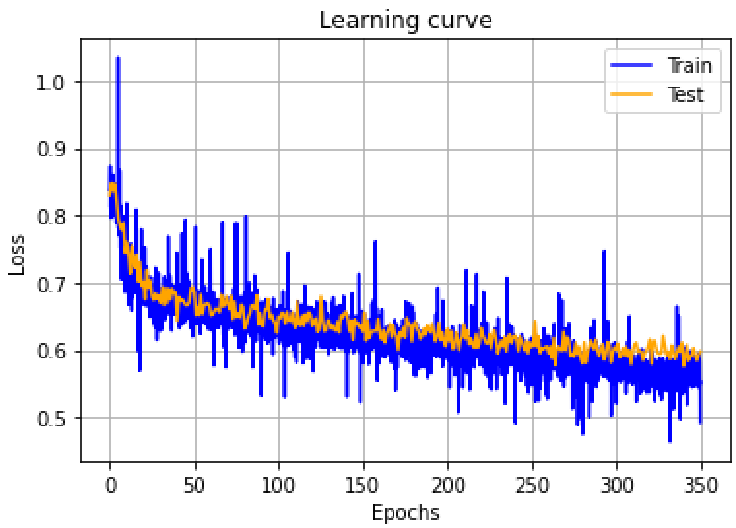
`Epoch:250            Train Loss:0.54783           Test Loss0.60442`



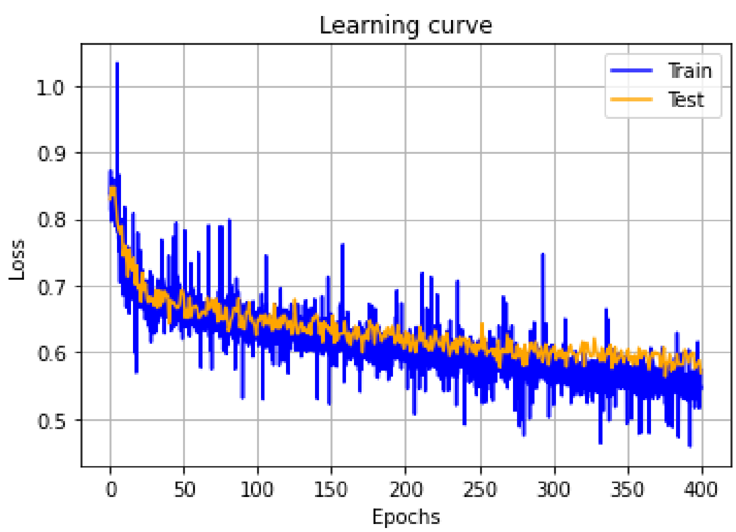
`Epoch:300          Train Loss:0.52058          Test Loss0.60132`



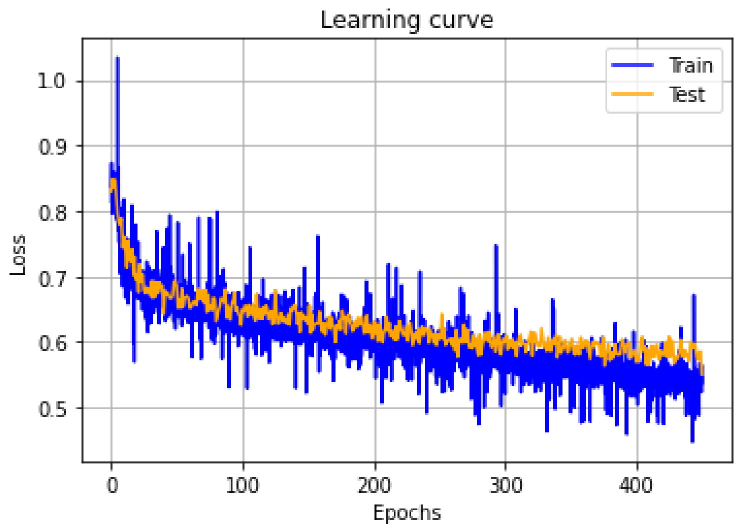
`Epoch:350          Train Loss:0.49249          Test Loss0.59774`



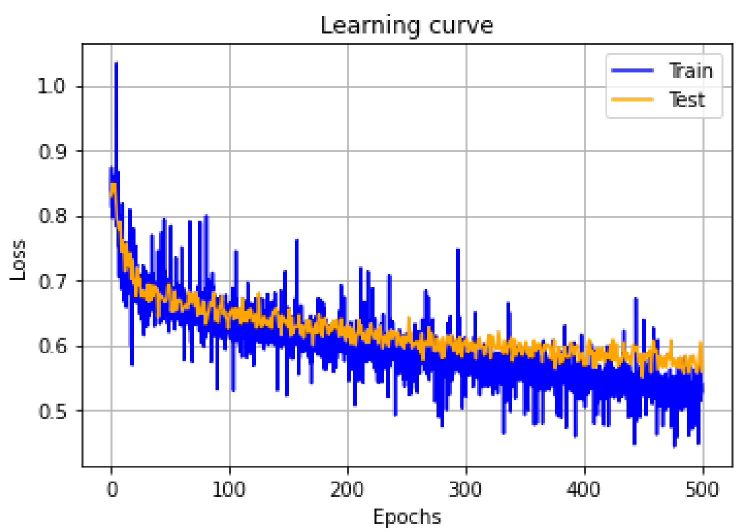

```
Epoch:400                Train Loss:0.57309              Test Loss0.56896
```



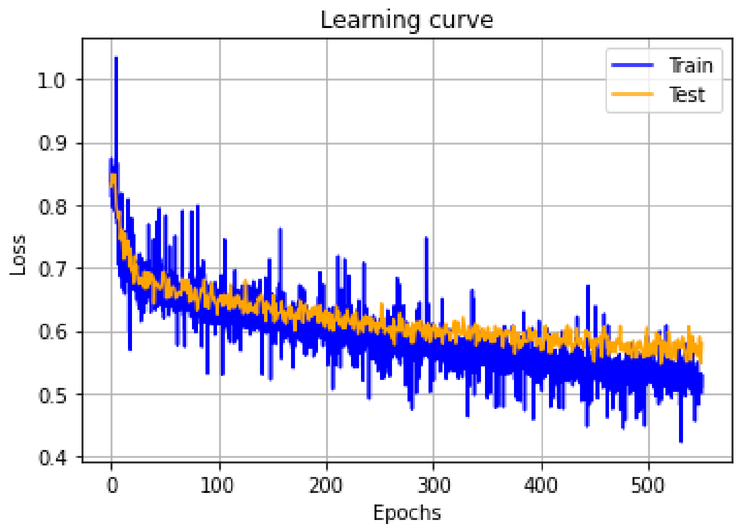

```
Epoch:450                Train Loss:0.53689              Test Loss0.55060
```



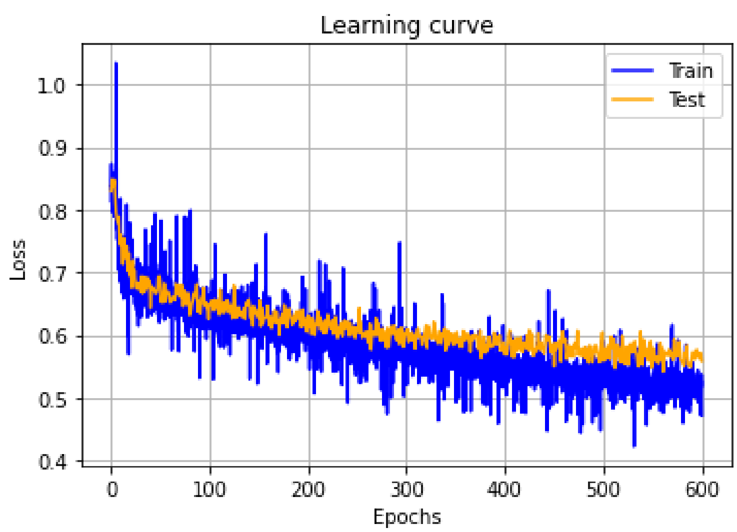
Epoch:500 Train Loss:0.53085 Test Loss0.55895



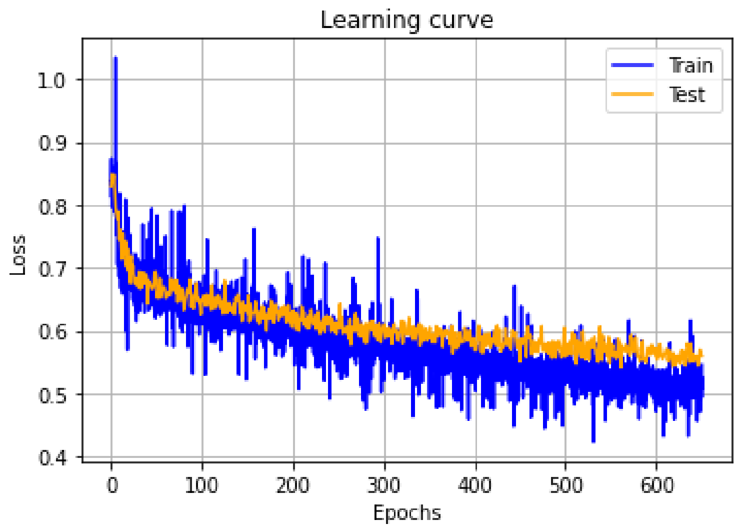
Epoch:550 Train Loss:0.50084 Test Loss0.57970



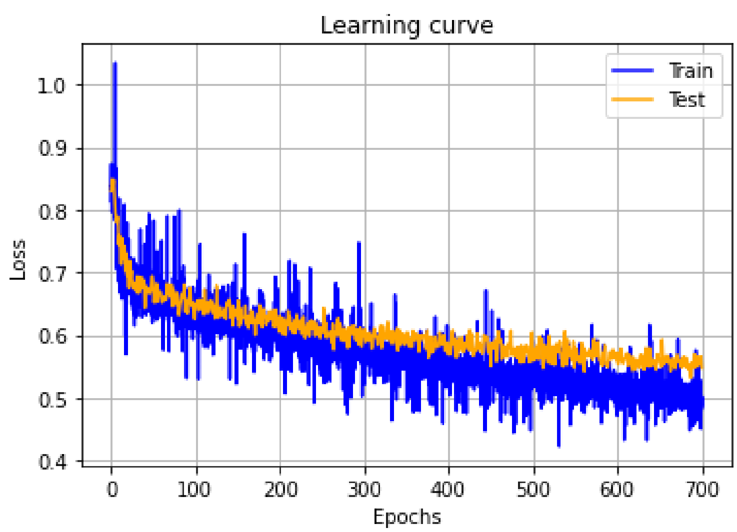

```
Epoch:600                Train Loss:0.47153              Test Loss0.55886
```



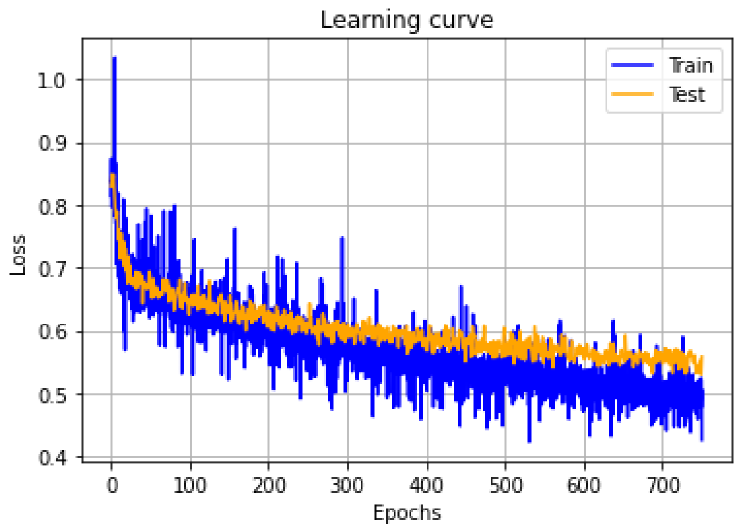

```
Epoch:650                Train Loss:0.54518              Test Loss0.55987
```



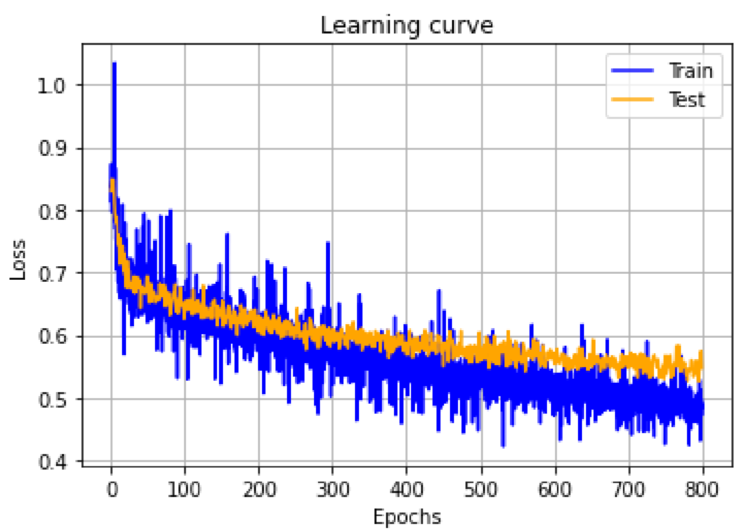

```
Epoch:700                Train Loss:0.48743              Test Loss0.54751
```



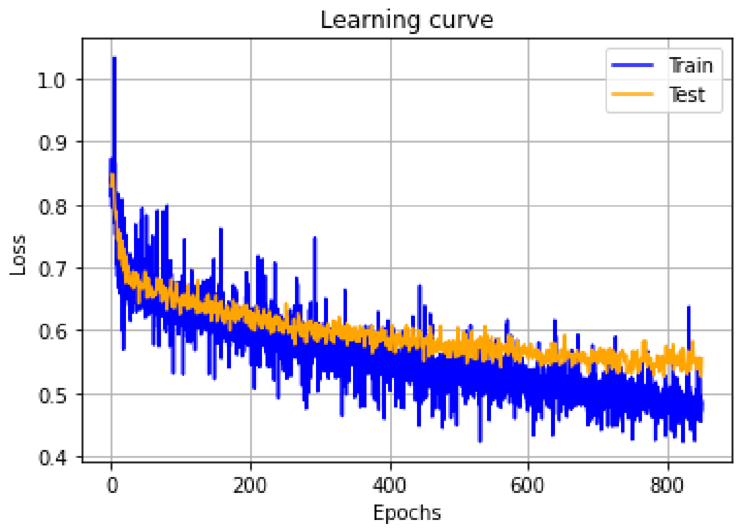

```
Epoch:750                Train Loss:0.42612              Test Loss0.55884
```



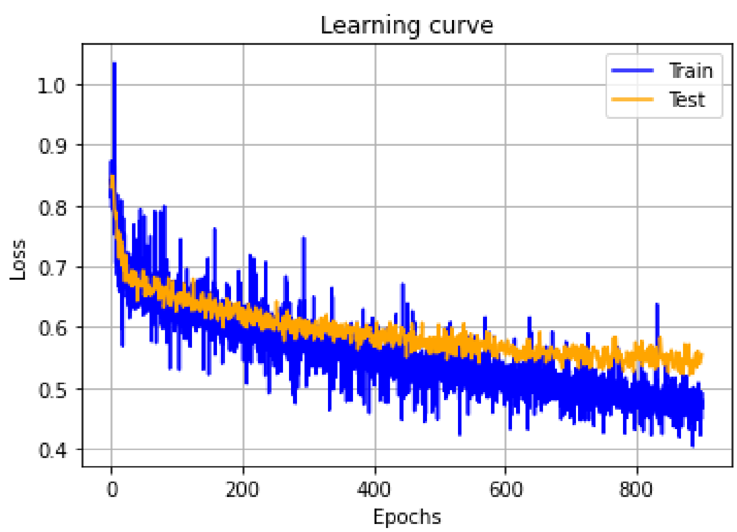
`Epoch:800 Train Loss:0.49039 Test Loss0.54734`



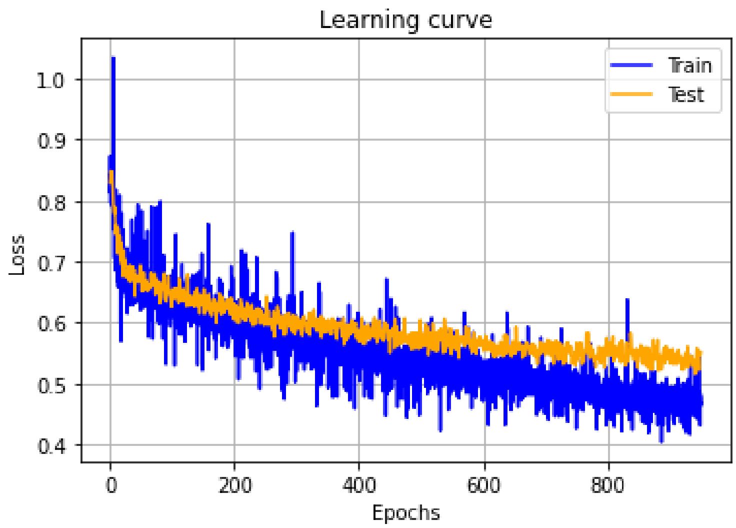
`Epoch:850 Train Loss:0.47232 Test Loss0.55523`



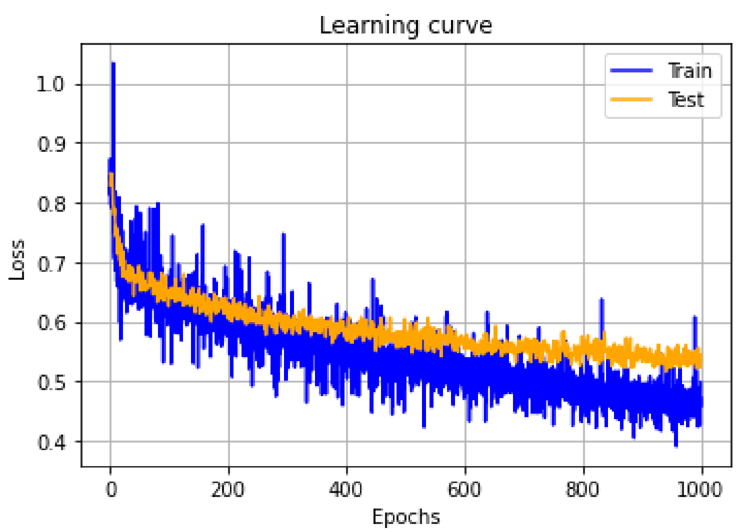
`Epoch:900 Train Loss:0.45040 Test Loss0.55532`



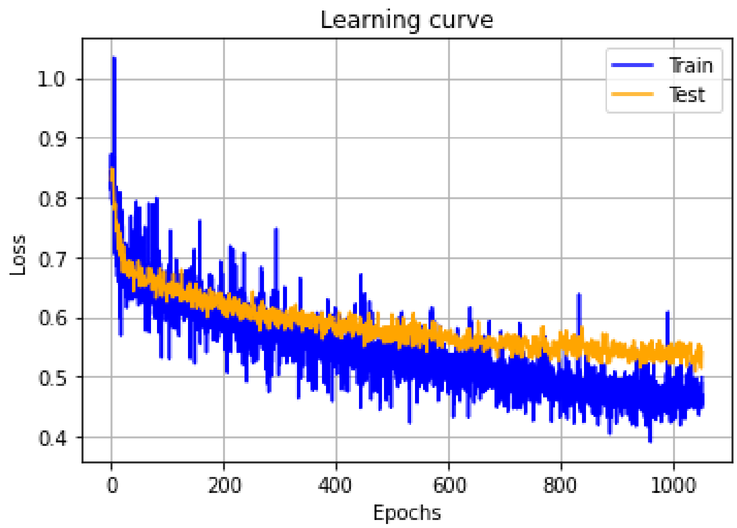
`Epoch:950 Train Loss:0.46831 Test Loss0.55107`



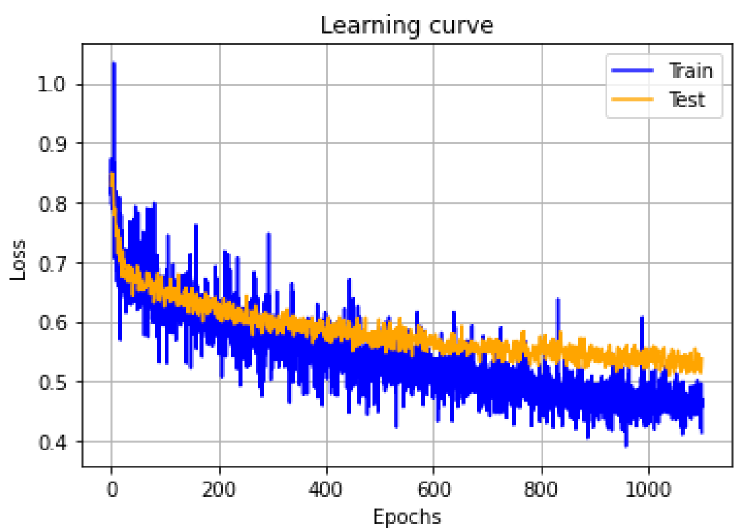

```
Epoch:1000                  Train Loss:0.49804                 Test Loss0.54339
```



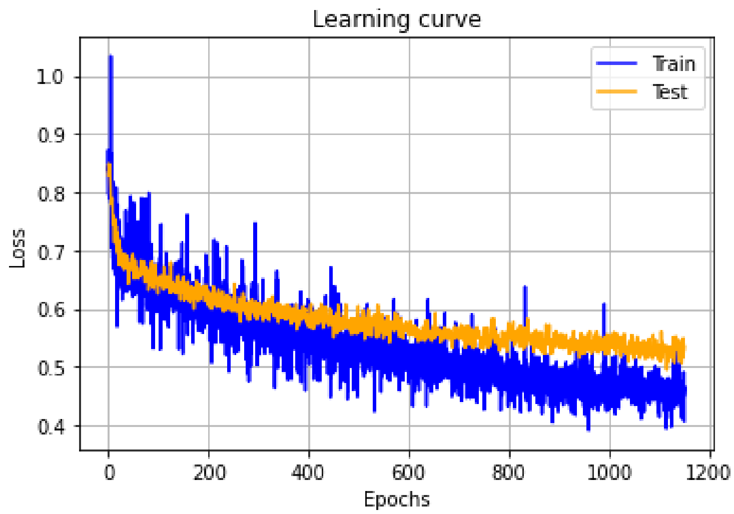

```
Epoch:1050                  Train Loss:0.49875                 Test Loss0.54089
```



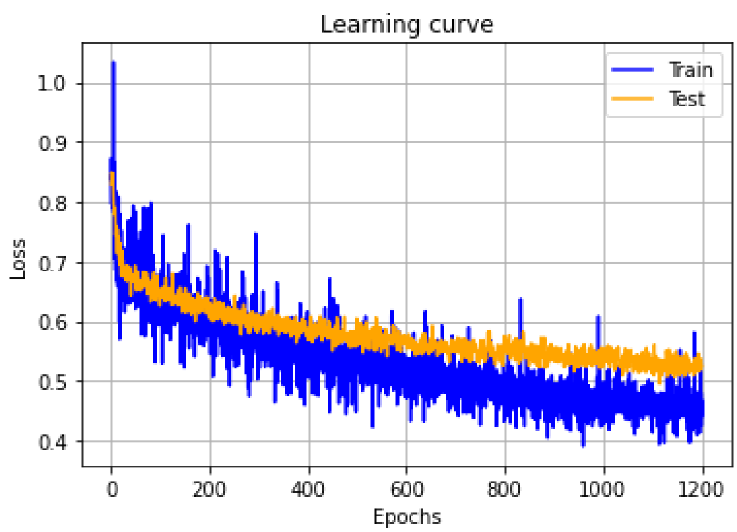
`Epoch:1100 Train Loss:0.41341 Test Loss0.53759`



`Epoch:1150 Train Loss:0.40639 Test Loss0.53541`




```
Epoch:1200                Train Loss:0.42686                Test Loss0.52573
```



0.1 Train Resualts:

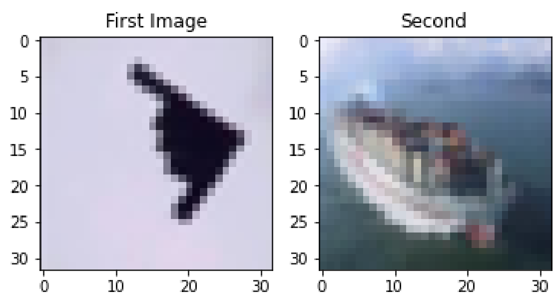
$MAE = 0.42686 \xrightarrow{\approx} MSE \approx \sqrt{0.487} \approx 0.18220946$

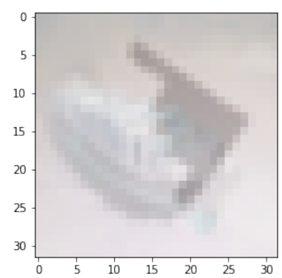
0.2 Test Resualts:

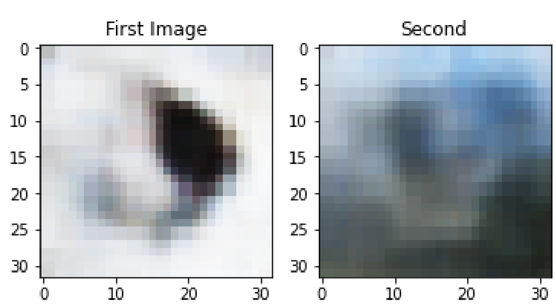
$MAE = 0.52573 \xrightarrow{\approx} MSE \approx \sqrt{0.487} \approx 0.276392033$

[11]:

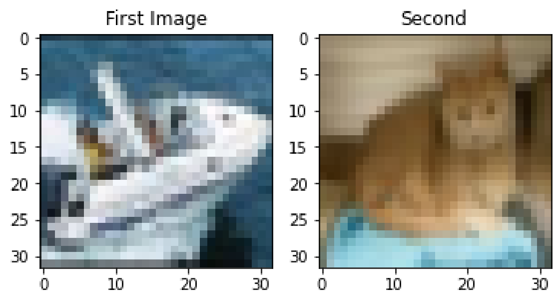
```
# visualize an example from Training set
f=4
s=7

imshowD(images[f], images[s])
print('-------------------')
imshow(AVG(images[f],images[s]))
print('-------------------')
recon1,recon2=model(AVG(images[f],images[s])[None,:, :, :])
with torch.no_grad():
  recon1,recon2=torch.tensor(recon1).squeeze(),torch.tensor(recon2).squeeze()
imshowD(recon1,recon2)
```



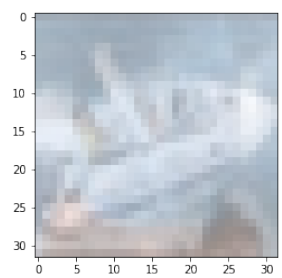




```
[14]: # visualize an example from Test set
f=55
s=19
imshowD(images_v[f], images_v[s])
print('-------------------')
imshow(AVG(images_v[f],images_v[s]))
print('-------------------')
recon1_v,recon2_v= model(AVG(images_v[f],images_v[s])[None,:, :, :])
recon1_v,recon2_v=torch.tensor(recon1_v).squeeze(),torch.tensor(recon2_v).squeeze()
imshowD(recon1_v,recon2_v)
```




```
-------------------
```



```
-------------------
```