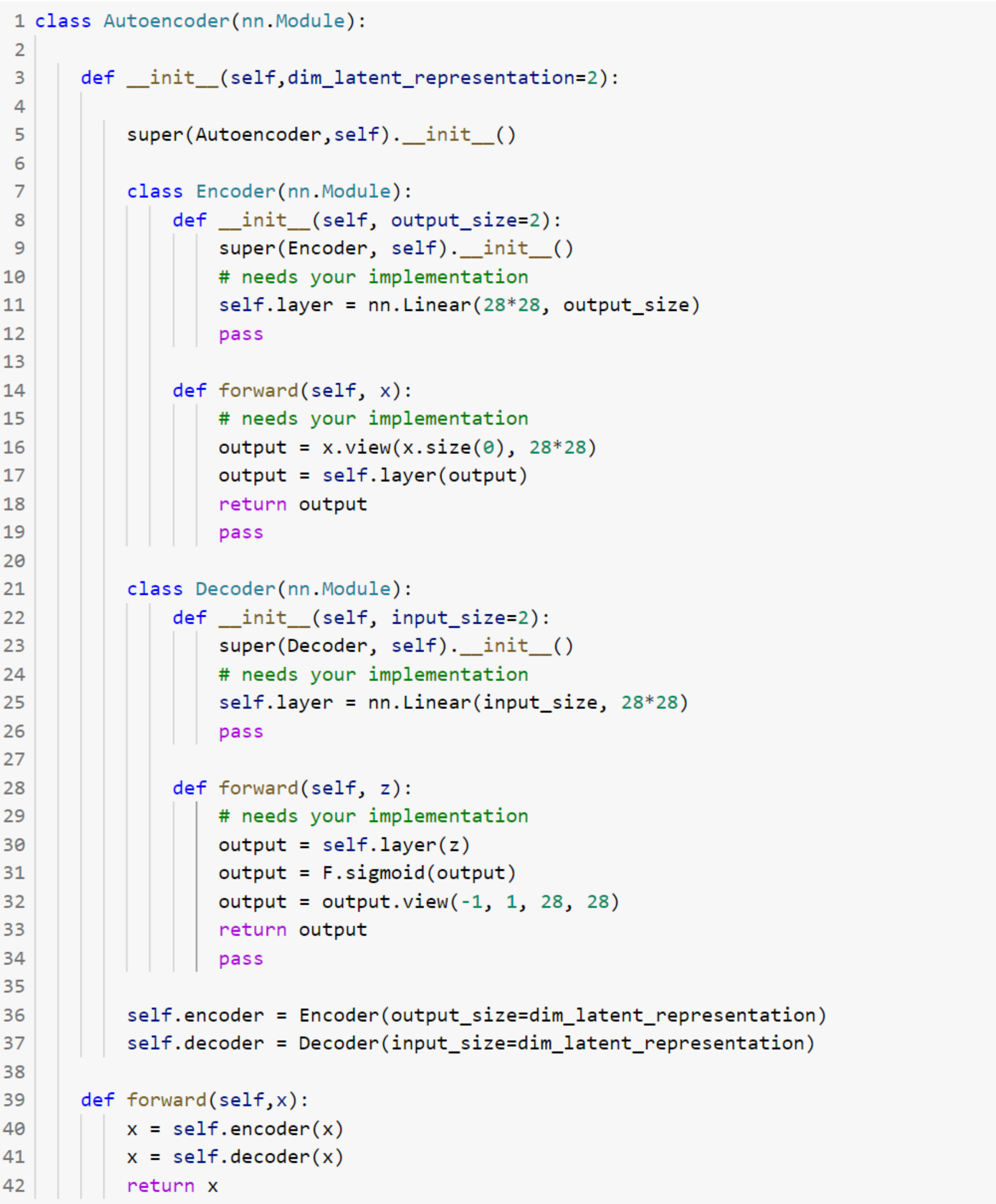
I certify that all solutions are entirely in my own words and that I have not looked at another student’s solutions. I have given credit to all external sources I consulted. 黄郅鑫

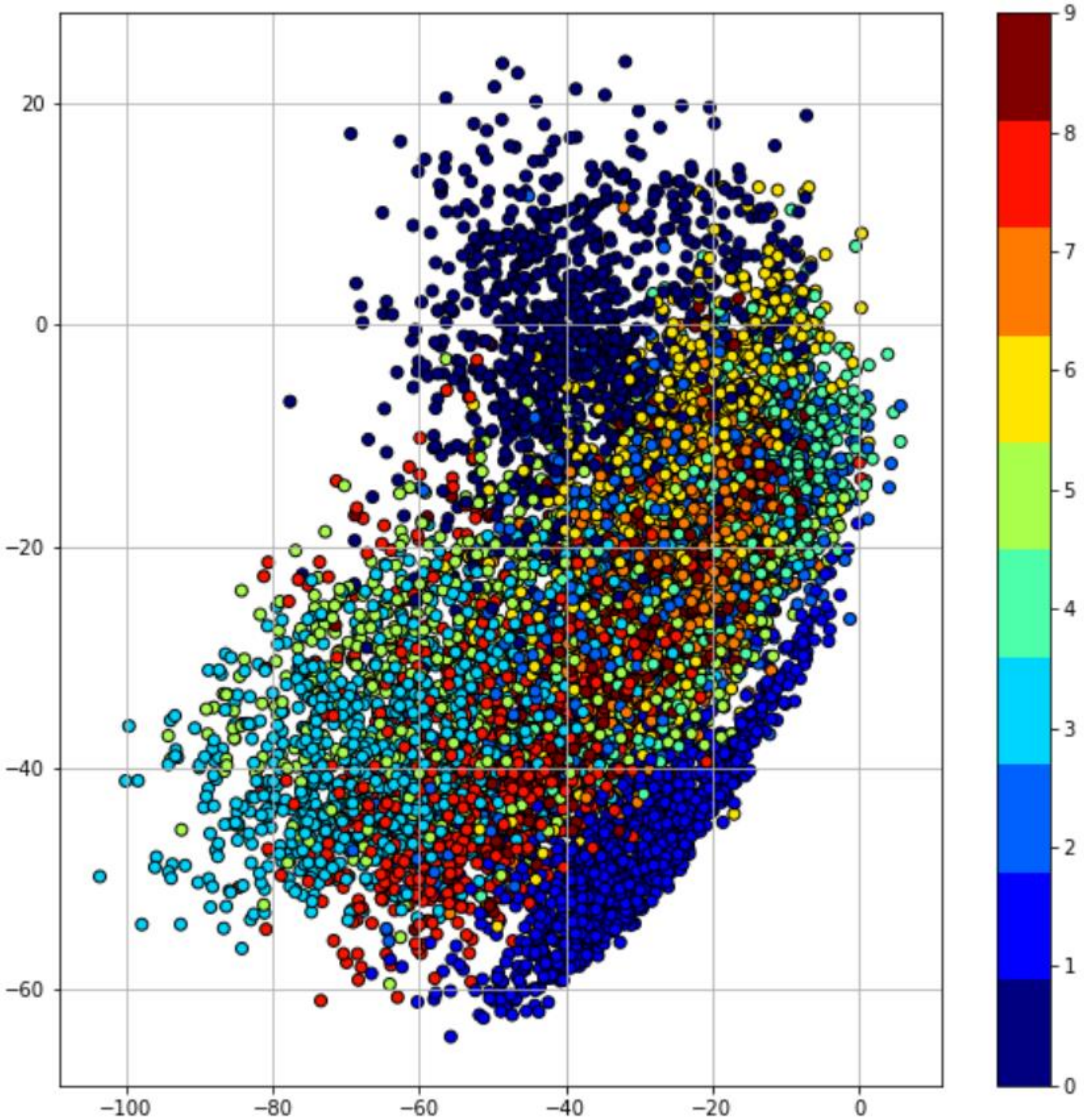
2.

a) reconstruction losses on the train sets = 0.7374

reconstruction losses on the validation sets = 0.7427

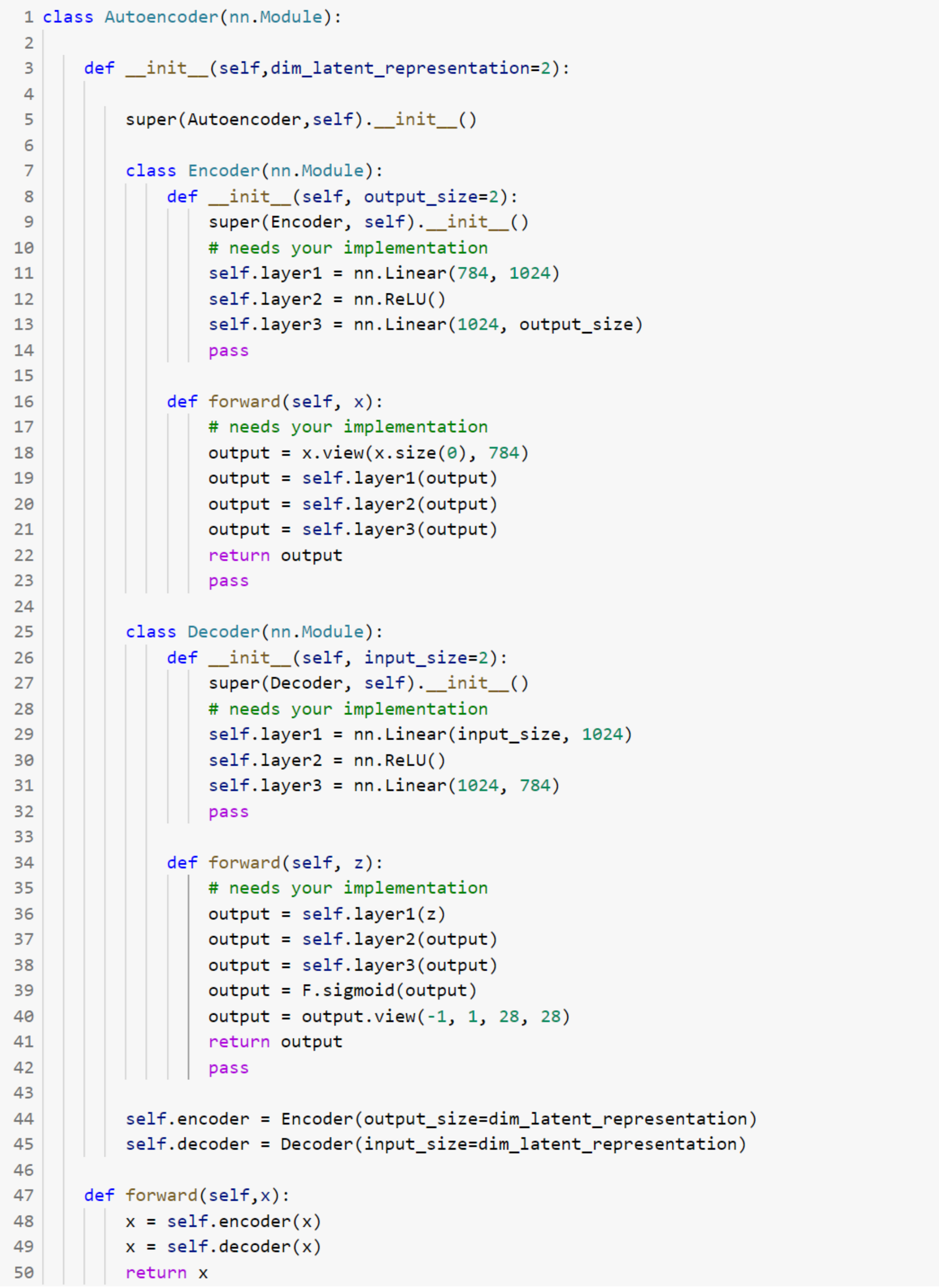


```python
class Autoencoder(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(Autoencoder,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layer = nn.Linear(28*28, output_size)
                pass

            def forward(self, x):
                # needs your implementation
                output = x.view(x.size(0), 28*28)
                output = self.layer(output)
                return output
                pass

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layer = nn.Linear(input_size, 28*28)
                pass

            def forward(self, z):
                # needs your implementation
                output = self.layer(z)
                output = F.sigmoid(output)
                output = output.view(-1, 1, 28, 28)
                return output
                pass

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)

    def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

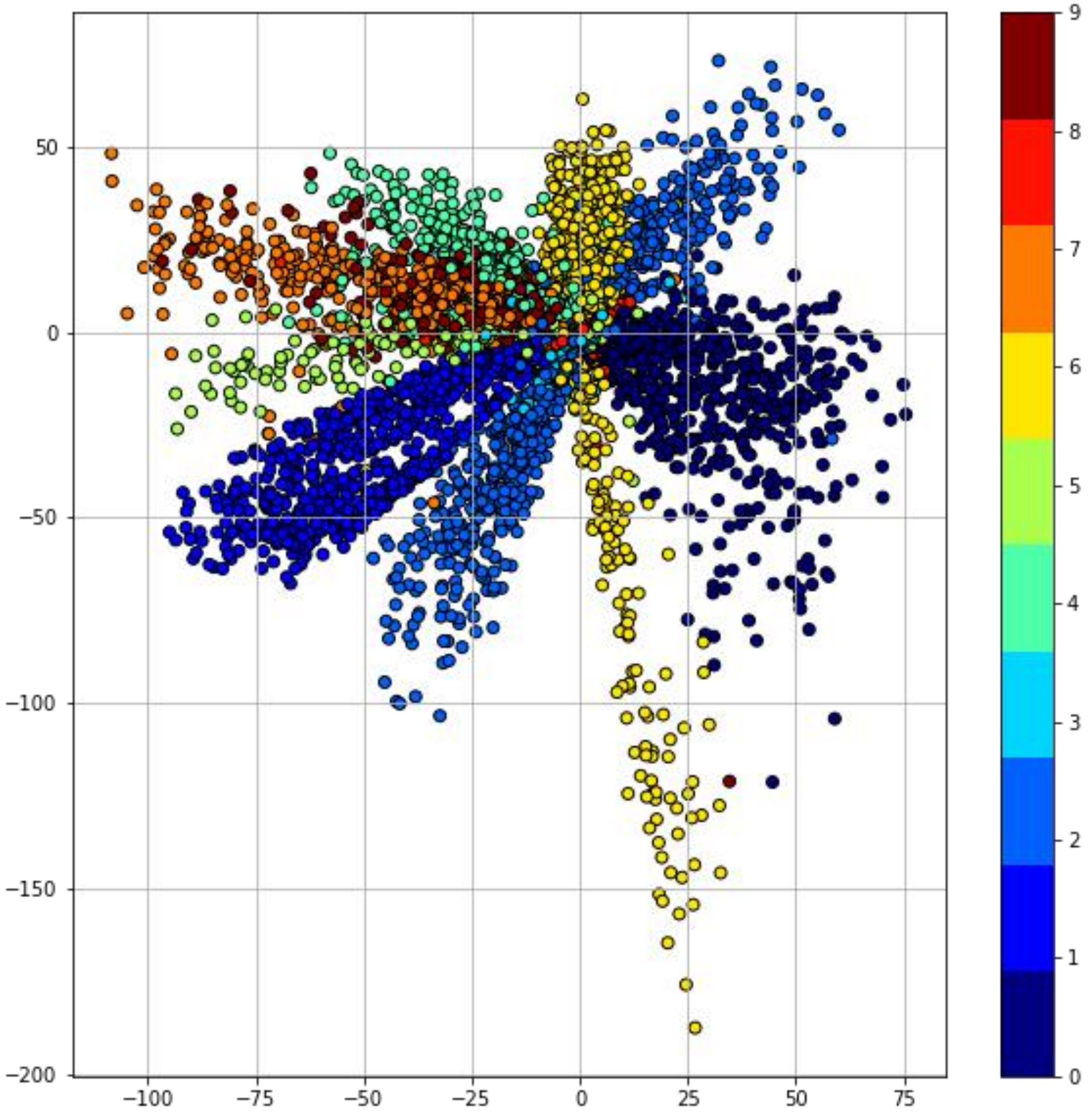

20
0
−20
−40
−60
−100
−80
−60
−40
−20
0
9
8
7
6
5
4
3
2
1
0

b) reconstruction losses on the train sets = 0.6703

reconstruction losses on the validation sets = 0.6773



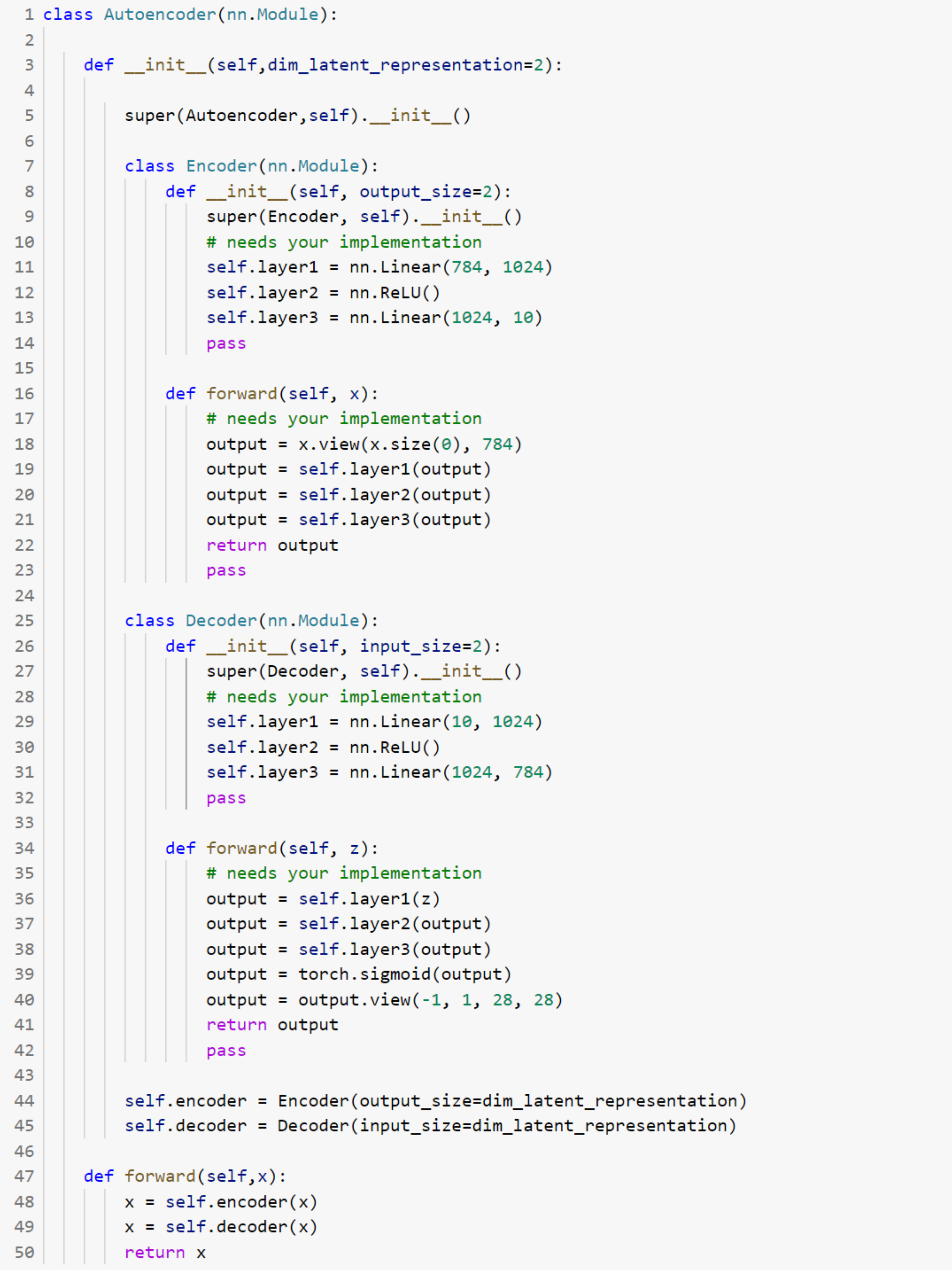
```python
class Autoencoder(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(Autoencoder,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(784, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, output_size)
                pass

            def forward(self, x):
                # needs your implementation
                output = x.view(x.size(0), 784)
                output = self.layer1(output)
                output = self.layer2(output)
                output = self.layer3(output)
                return output
                pass

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(input_size, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, 784)
                pass

            def forward(self, z):
                # needs your implementation
                output = self.layer1(z)
                output = self.layer2(output)
                output = self.layer3(output)
                output = F.sigmoid(output)
                output = output.view(-1, 1, 28, 28)
                return output
                pass

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)

    def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```



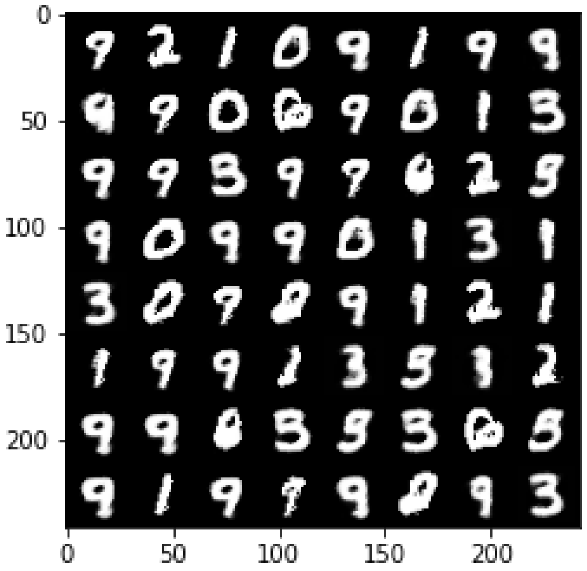
The plot in part a is more random than in part b, the points in part b are getting closer to (0, 0). The architecture in part b is more accurate than part a, because it has more layers.

c) reconstruction losses on the train sets = 0.5259

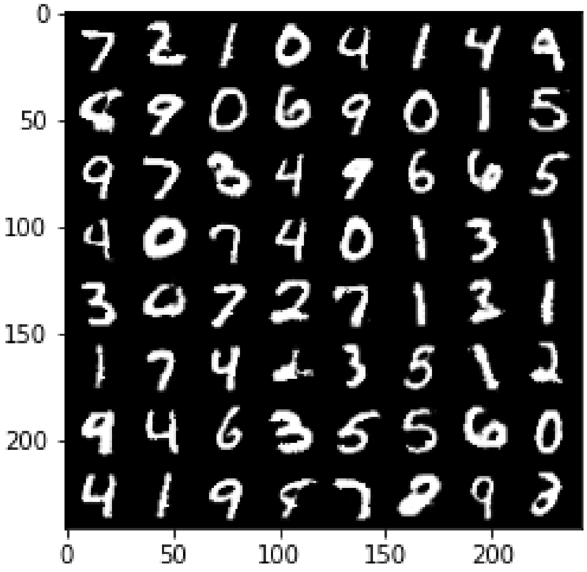
reconstruction losses on the validation sets = 0.5387



```
class Autoencoder(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(Autoencoder,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(784, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, 10)
                pass

            def forward(self, x):
                # needs your implementation
                output = x.view(x.size(0), 784)
                output = self.layer1(output)
                output = self.layer2(output)
                output = self.layer3(output)
                return output
                pass

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(10, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, 784)
                pass

            def forward(self, z):
                # needs your implementation
                output = self.layer1(z)
                output = self.layer2(output)
                output = self.layer3(output)
                output = torch.sigmoid(output)
                output = output.view(-1, 1, 28, 28)
                return output
                pass

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)

    def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```



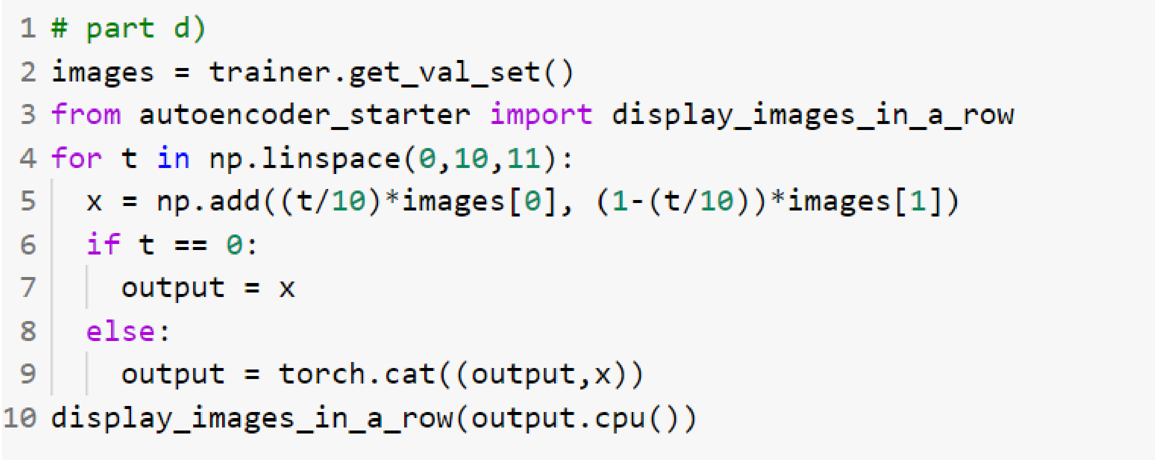
bottleneck feature with 2 dimensions



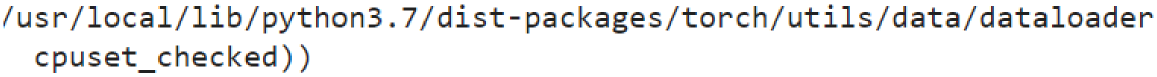
bottleneck feature with 10 dimensions

The image in part c is more accurate than part b compares to the original image, because the bottleneck feature in part c has more dimensions.

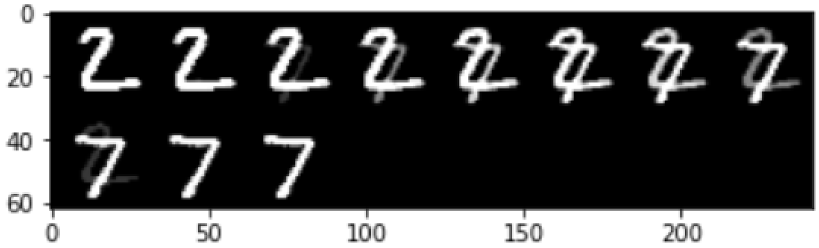
d)



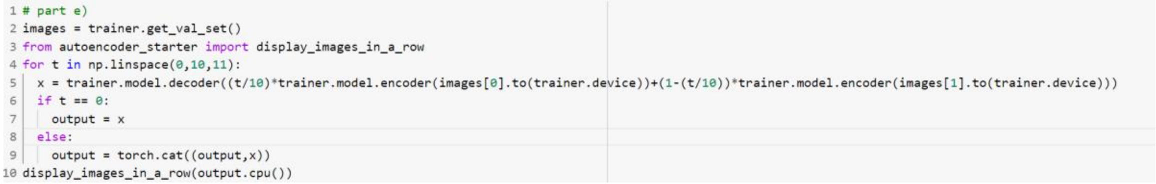
```
# part d)
images = trainer.get_val_set()
from autoencoder_starter import display_images_in_a_row
for t in np.linspace(0,10,11):
  x = np.add((t/10)*images[0], (1-(t/10))*images[1])
  if t == 0:
    output = x
  else:
    output = torch.cat((output,x))
display_images_in_a_row(output.cpu())
```



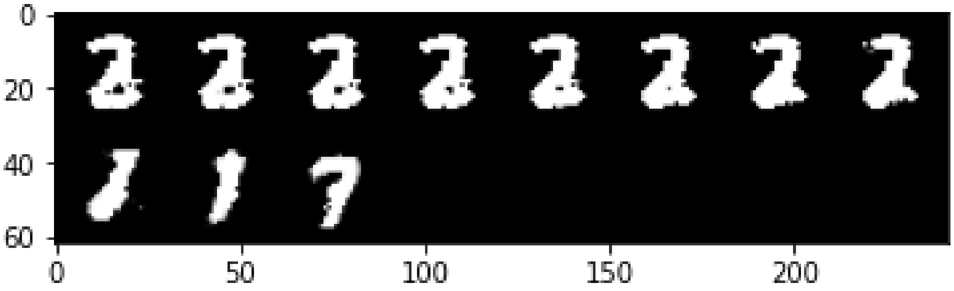
```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader
  cpuset_checked))
```



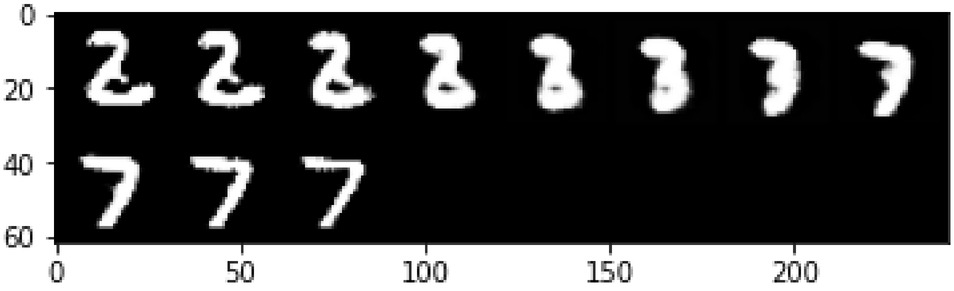
e)



```
# part e)
images = trainer.get_val_set()
from autoencoder_starter import display_images_in_a_row
for t in np.linspace(0,10,11):
  x = trainer.model.decoder((t/10)*trainer.model.encoder(images[0].to(trainer.device))+(1-(t/10))*trainer.model.encoder(images[1].to(trainer.device)))
  if t == 0:
    output = x
  else:
    output = torch.cat((output,x))
display_images_in_a_row(output.cpu())
```



Model from part b

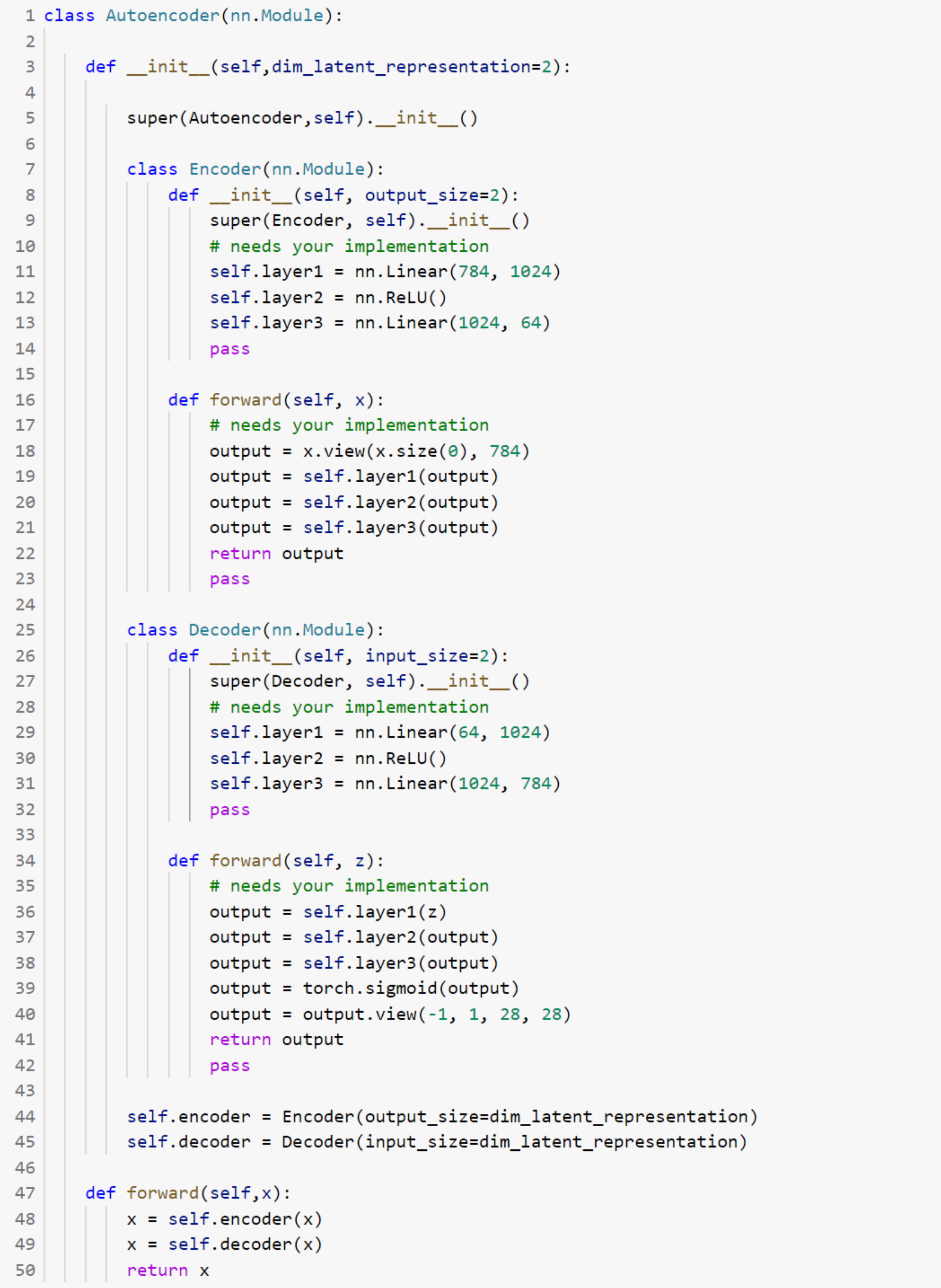


Model from part c

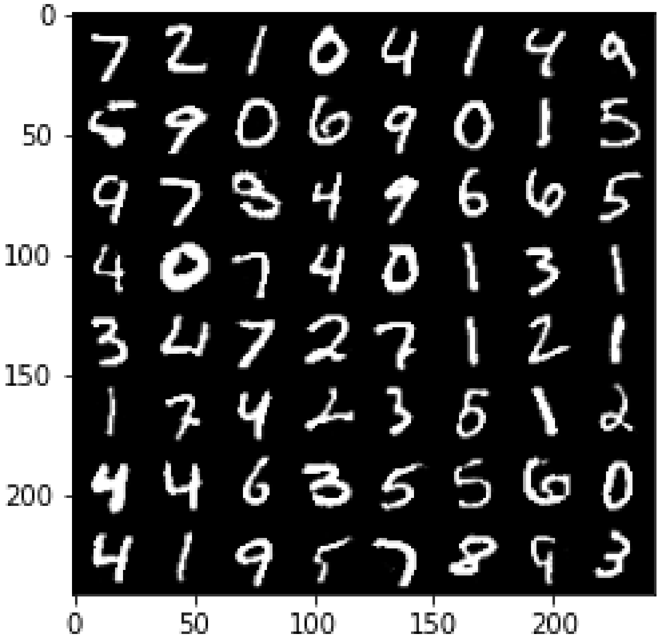
These image in bottleneck feature space are more like a single number, the images in raw pixel space from part (d) are more like two numbers combiner together. I think this is because part e used the encoder and decoder, the decoder can convert some random bottleneck feature to an image of a single number. The image of model from part c is more like numbers than the model from part b, because it has more dimensions, accuracy is higher.

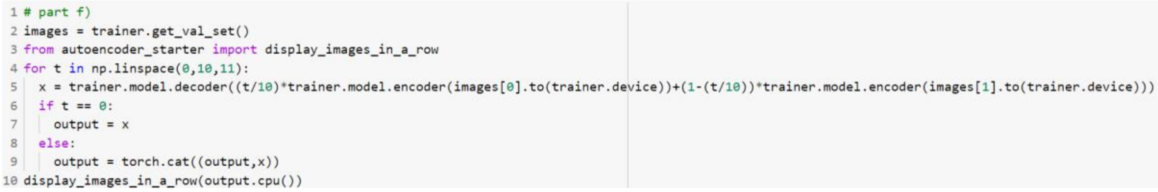
f) reconstruction losses on the train sets = 0.4810

reconstruction losses on the validation sets = 0.4926

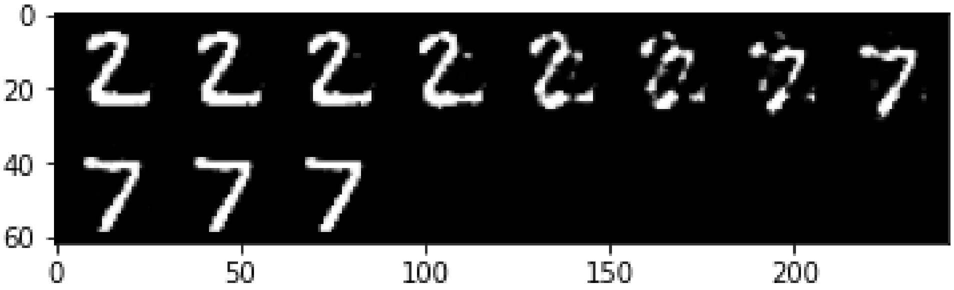


```
class Autoencoder(nn.Module):

    def __init__(self,dim_latent_representation=2):

        super(Autoencoder,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(784, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, 64)
                pass

            def forward(self, x):
                # needs your implementation
                output = x.view(x.size(0), 784)
                output = self.layer1(output)
                output = self.layer2(output)
                output = self.layer3(output)
                return output
                pass

        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.layer1 = nn.Linear(64, 1024)
                self.layer2 = nn.ReLU()
                self.layer3 = nn.Linear(1024, 784)
                pass

            def forward(self, z):
                # needs your implementation
                output = self.layer1(z)
                output = self.layer2(output)
                output = self.layer3(output)
                output = torch.sigmoid(output)
                output = output.view(-1, 1, 28, 28)
                return output
                pass

        self.encoder = Encoder(output_size=dim_latent_representation)
        self.decoder = Decoder(input_size=dim_latent_representation)

    def forward(self,x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```





```
# part f)
images = trainer.get_val_set()
from autoencoder_starter import display_images_in_a_row
for t in np.linspace(0,10,11):
  x = trainer.model.decoder((t/10)*trainer.model.encoder(images[0].to(trainer.device))+(1-(t/10))*trainer.model.encoder(images[1].to(trainer.device)))
  if t == 0:
    output = x
  else:
    output = torch.cat((output,x))
display_images_in_a_row(output.cpu())
```



The image of model from part f is even more like numbers than the model from part e, the first four images in part f look exactly like “2”, and the last four images in part f look exactly like “7”, because it has even more dimensions, so that the accuracy is higher.