Assignment 4

Wanyi Su
#30144565

Q1 SVM

1. choose #4

① The problem defines a "soft-margin linear SVM", and no non-linearity is added, so the hyperplane should be linear. So we limit choice to 3, 4.

② Although this is a soft margin SVM, penalty hyperparameter C is 0.02, which is small. This means the SVM still values a relatively large margin with a little tolerance of misclassification.

2. choose #3

① The problem defines a "soft-margin linear SVM", and no non-linearity is added, so the hyperplane should be linear. So we limit choice to 3, 4.

② This is a soft-margin SVM with a large penalty hyperparameter C of 20. This means the SVM tolerates some level of misclassification with a trade-off of sacrificing maximum margin.
So compared with #4, this SVM is more generalized and has a narrower margin.

3. Choose #5

With a polynomial kernel of 2-degree, data points are transformed into a higher-dimensional feature space to be separated.

4. Choose #6.

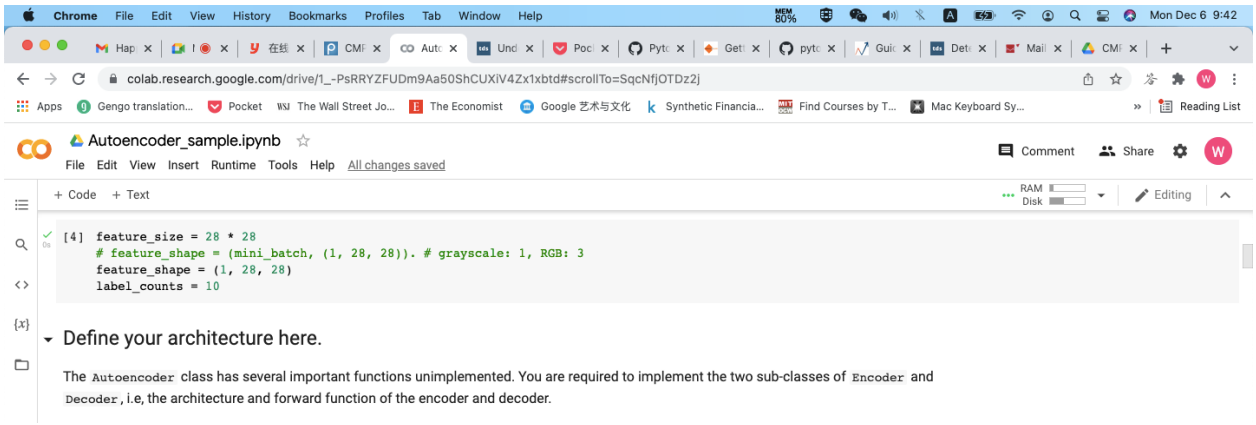
With a RBF kernel of gamma values 5, similarity radius is relatively small, which means data points need to be close to each other to be in the same class. A small noise may affect a data point's classification. This means SVM can follow too much on noises. So the shape of the decision boundary can be not so smooth.

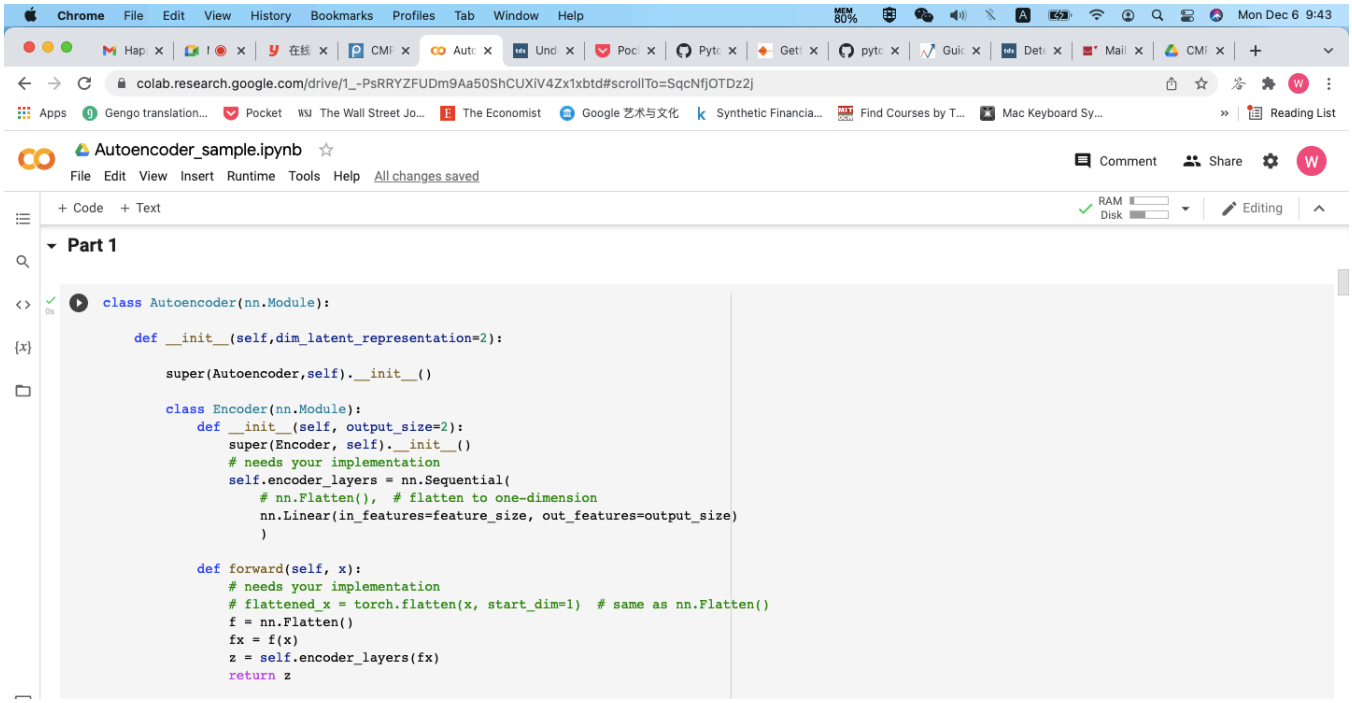
5. Choose #1.

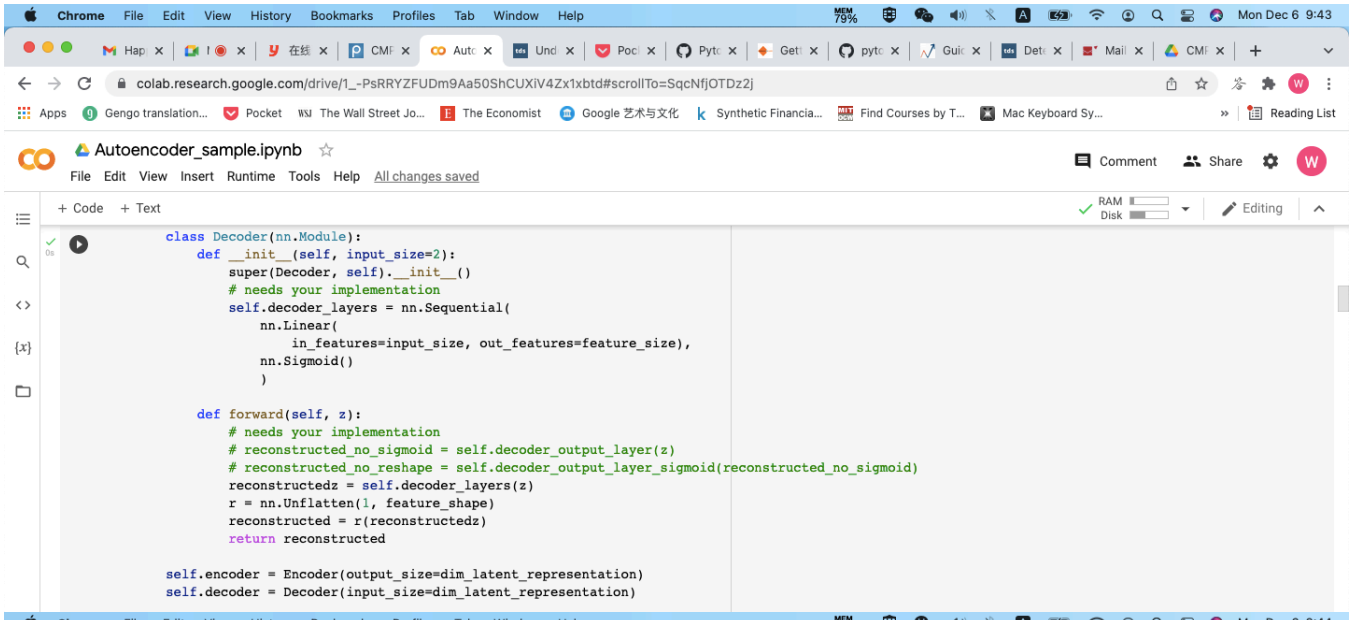
With a RBF kernel of gamma values at $\frac{1}{5}$, similarity radius is relatively larger. More data points can be classified together, and the areas on two sides of boundary are larger to encorporate more data points than #6. This SVM is more generalized and has smoother curve.

Q2

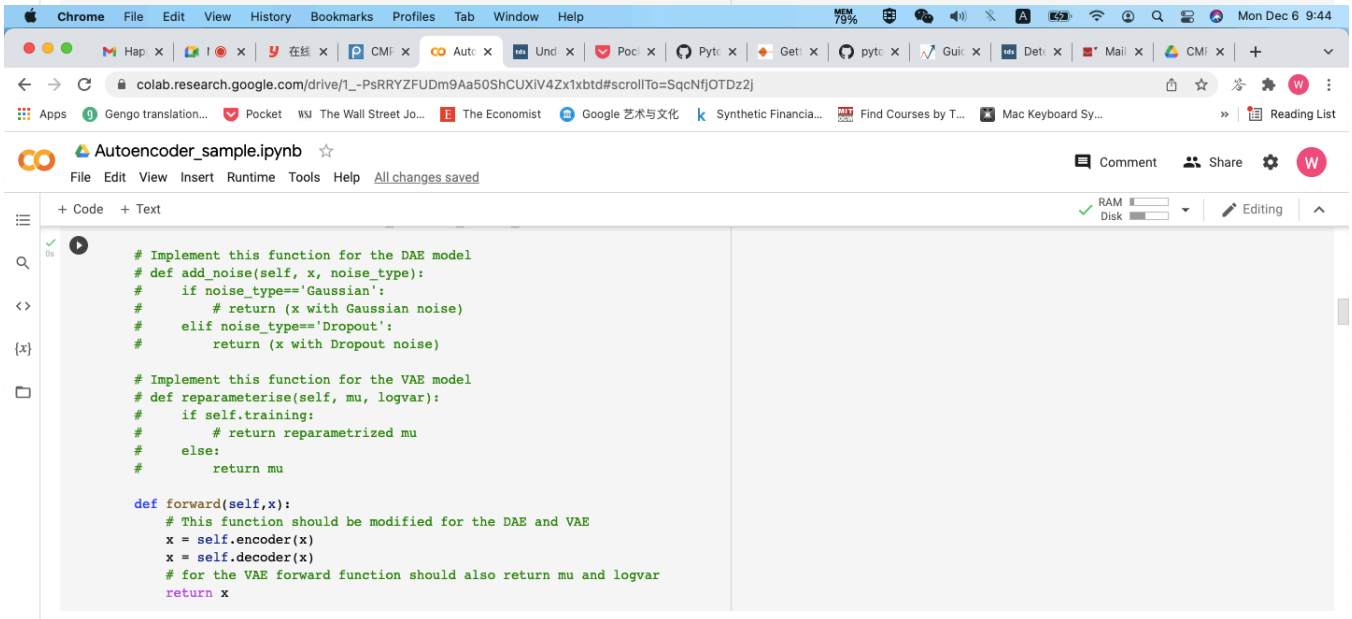
#1
Implement the architecture of the encoder and decoder and print out the reconstruction losses on the train and validation sets.
Also use the scatter plot function in autoencoder_ starter.py to plot the 2D bottleneck feature representation as a scatter plot, where different classes are represented by dots of different colours.
Include a screenshot of your code that implements the architecture and the generated scatter plot in your PDF submission.





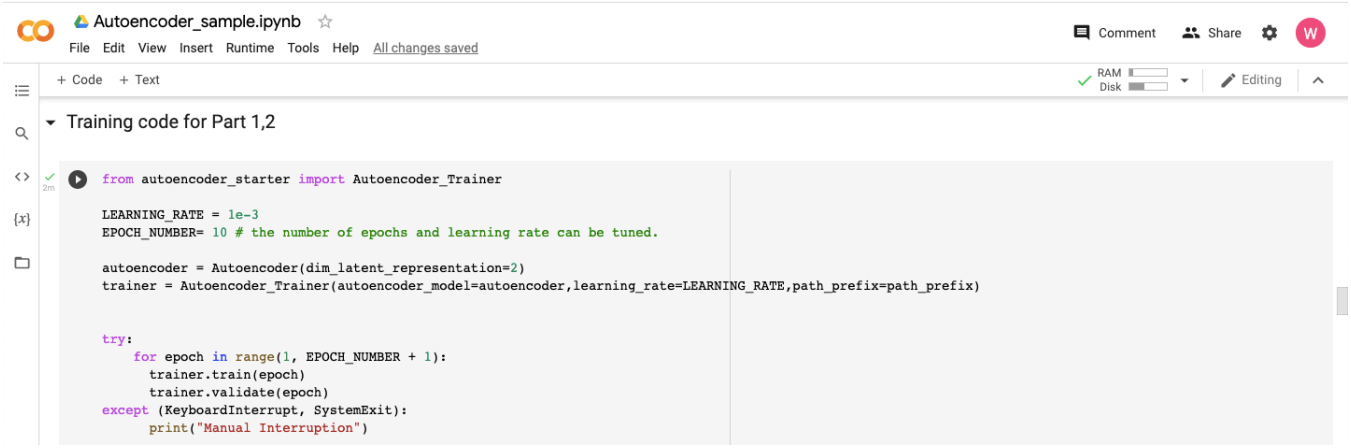


```
        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.decoder_layers = nn.Sequential(
                    nn.Linear(
                        in_features=input_size, out_features=feature_size),
                    nn.Sigmoid()
                    )

            def forward(self, z):
                # needs your implementation
                # reconstructed_no_sigmoid = self.decoder_output_layer(z)
                # reconstructed_no_reshape = self.decoder_output_layer_sigmoid(reconstructed_no_sigmoid)
                reconstructedz = self.decoder_layers(z)
                r = nn.Unflatten(1, feature_shape)
                reconstructed = r(reconstructedz)
                return reconstructed

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



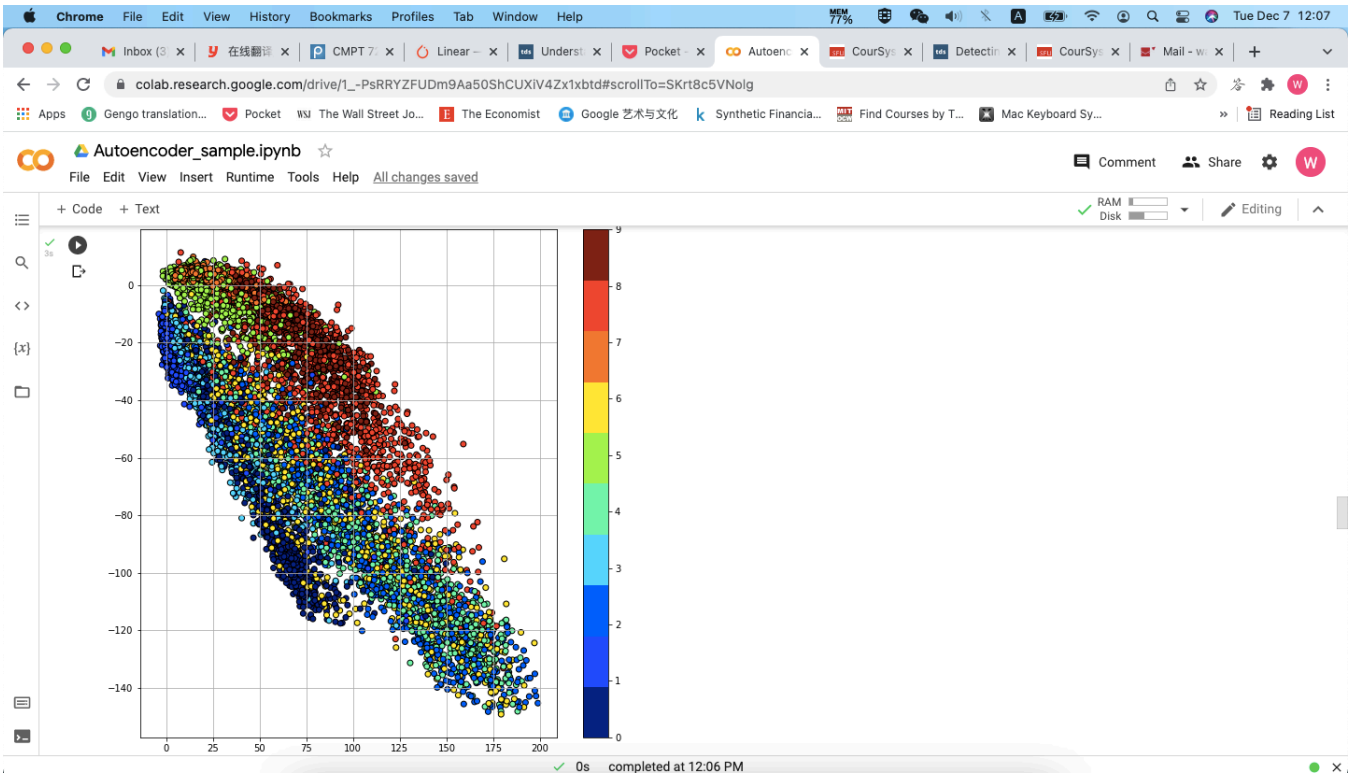
```
    # Implement this function for the DAE model
    # def add_noise(self, x, noise_type):
    #     if noise_type=='Gaussian':
    #         # return (x with Gaussian noise)
    #     elif noise_type=='Dropout':
    #         return (x with Dropout noise)

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self,x):
        # This function should be modified for the DAE and VAE
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

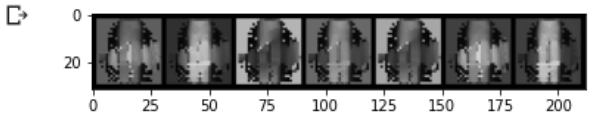
Training code:



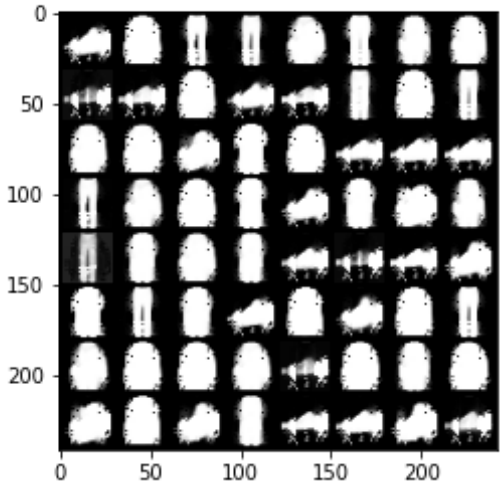
Losses:

I have tried LEARNING_RATE=[1e-4, 1e-3, 1e-2] and EPOCH_NUMBER=[10, 20], it turns out that the set of 1e-3 and 10 works relatively better to get training and validation loss around 0.75.



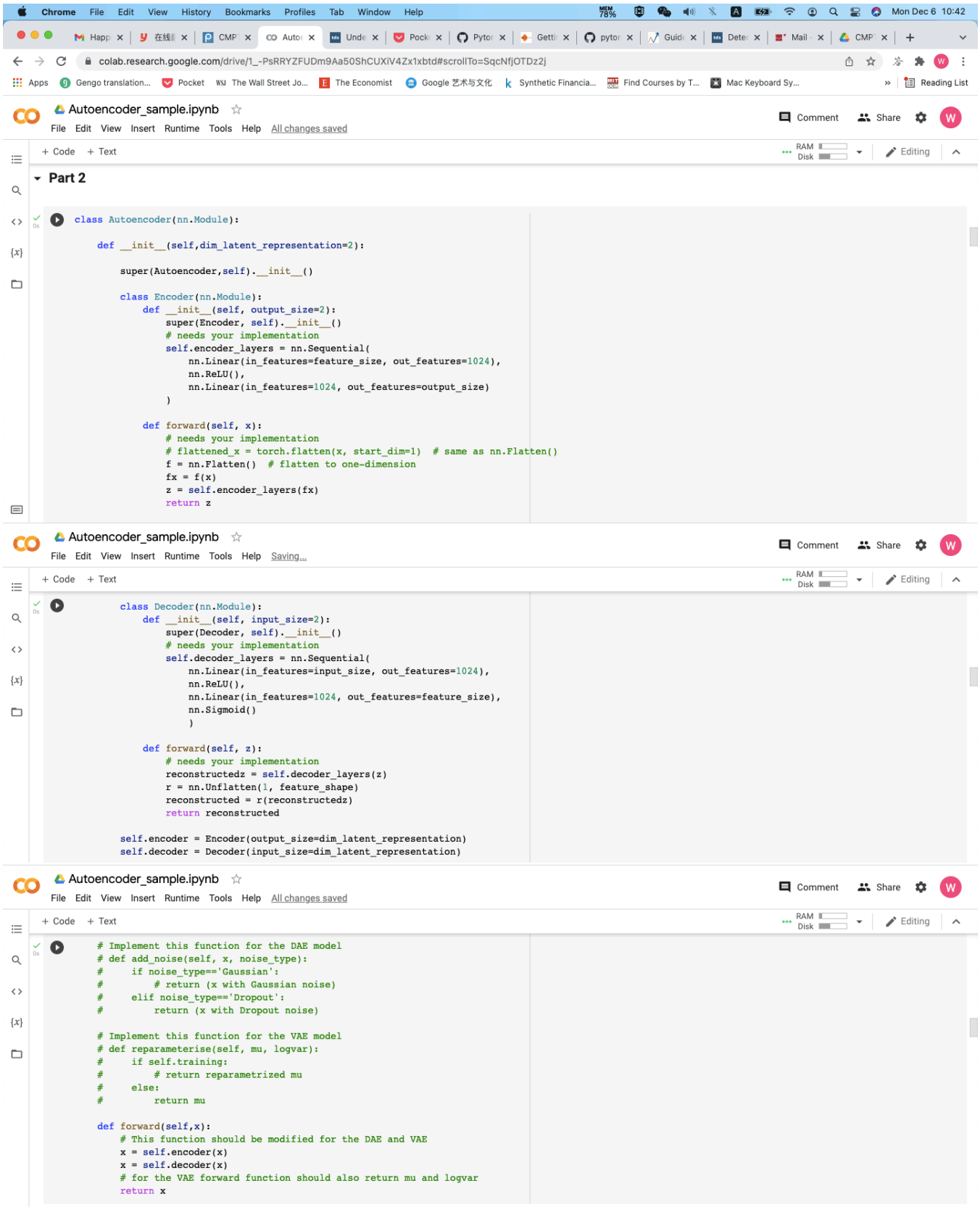


Reconstructed images



#2

Print out the reconstruction losses on the train and validation sets and generate a scatter plot of the same form as in part (1).
Include a screenshot of the code that implements the architecture and the generated scatter plot in the PDF submission.



Autoencoder_sample.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk Editing

Part 2

```
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.encoder_layers = nn.Sequential(
                    nn.Linear(in_features=feature_size, out_features=1024),
                    nn.ReLU(),
                    nn.Linear(in_features=1024, out_features=output_size)
                )

            def forward(self, x):
                # needs your implementation
                # flattened_x = torch.flatten(x, start_dim=1)  # same as nn.Flatten()
                f = nn.Flatten()  # flatten to one-dimension
                fx = f(x)
                z = self.encoder_layers(fx)
                return z
```

Autoencoder_sample.ipynb

File Edit View Insert Runtime Tools Help Saving...

Comment Share

+ Code + Text

RAM Disk Editing

```
        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.decoder_layers = nn.Sequential(
                    nn.Linear(in_features=input_size, out_features=1024),
                    nn.ReLU(),
                    nn.Linear(in_features=1024, out_features=feature_size),
                    nn.Sigmoid()
                    )

            def forward(self, z):
                # needs your implementation
                reconstructedz = self.decoder_layers(z)
                r = nn.Unflatten(1, feature_shape)
                reconstructed = r(reconstructedz)
                return reconstructed

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

Autoencoder_sample.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share

+ Code + Text

RAM Disk Editing

```
    # Implement this function for the DAE model
    # def add_noise(self, x, noise_type):
    #     if noise_type=='Gaussian':
    #         # return (x with Gaussian noise)
    #     elif noise_type=='Dropout':
    #         return (x with Dropout noise)

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self,x):
        # This function should be modified for the DAE and VAE
        x = self.encoder(x)
        x = self.decoder(x)
        # for the VAE forward function should also return mu and logvar
        return x
```

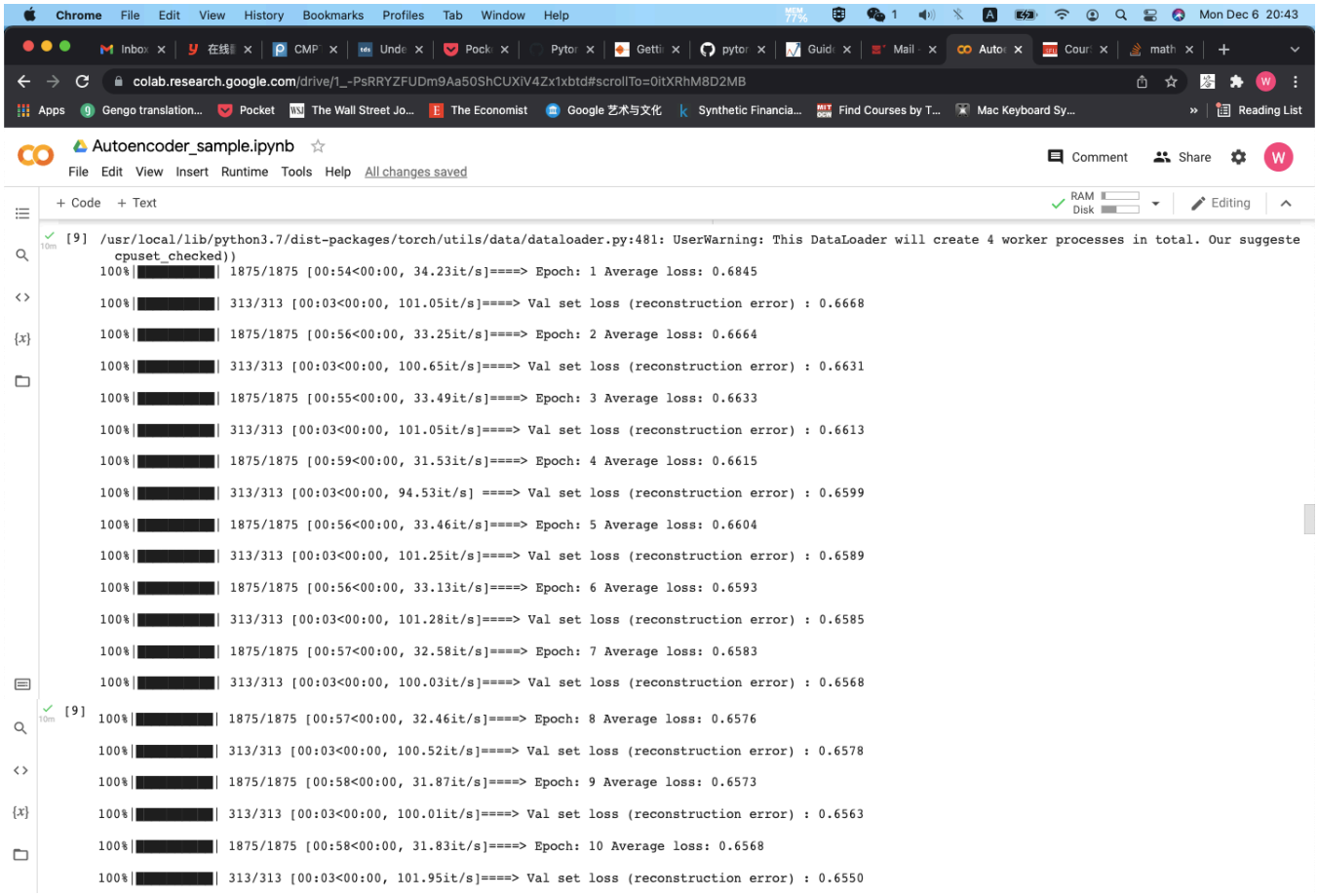
Training code:

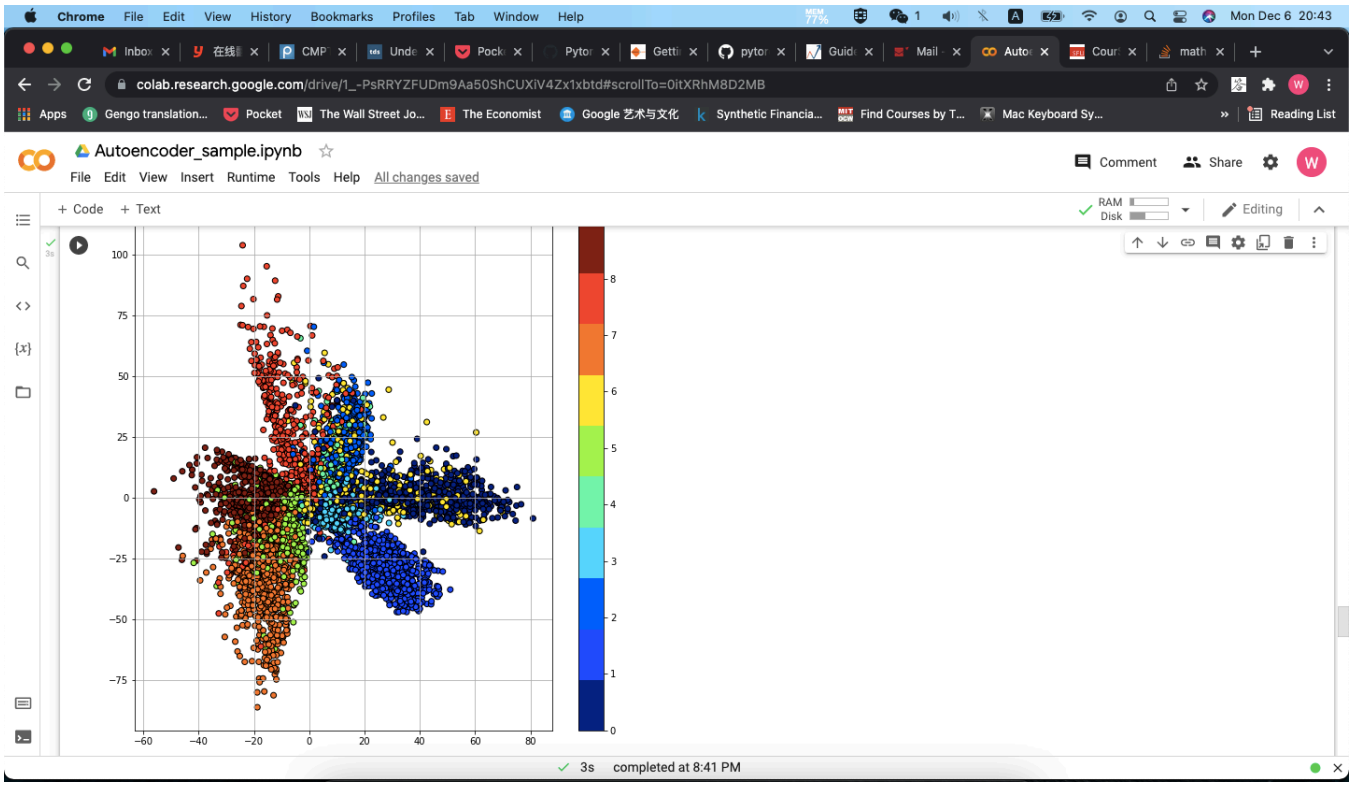
Training code for Part 1,2

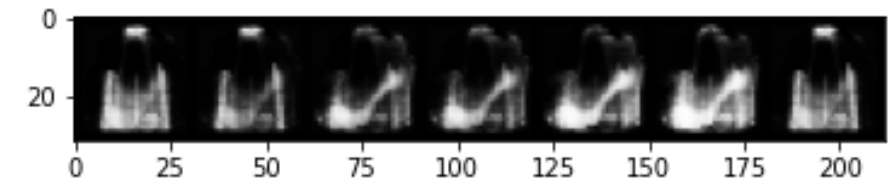
```
from autoencoder_starter import Autoencoder_Trainer

LEARNING_RATE = 1e-3
EPOCH_NUMBER= 10 # the number of epochs and learning rate can be tuned.

autoencoder = Autoencoder(dim_latent_representation=2)
trainer = Autoencoder_Trainer(autoencoder_model=autoencoder,learning_rate=LEARNING_RATE,path_prefix=path_prefix)


try:
    for epoch in range(1, EPOCH_NUMBER + 1):
      trainer.train(epoch)
      trainer.validate(epoch)
except (KeyboardInterrupt, SystemExit):
      print("Manual Interruption")
```

Losses:

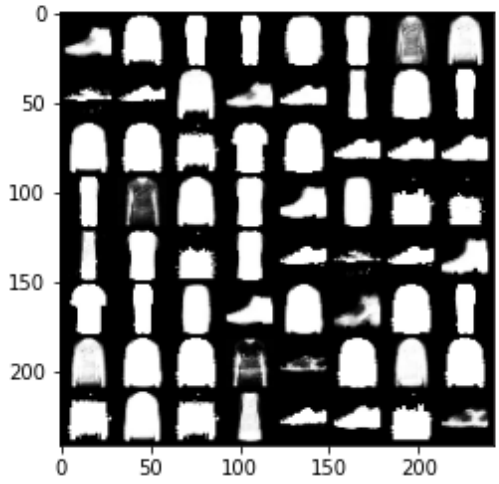
After comparison of several sets of hyperparameters, it turns out that the set of 1e-3 and 20 works relatively better to get training and validation loss around 0.66.







Reconstructed images



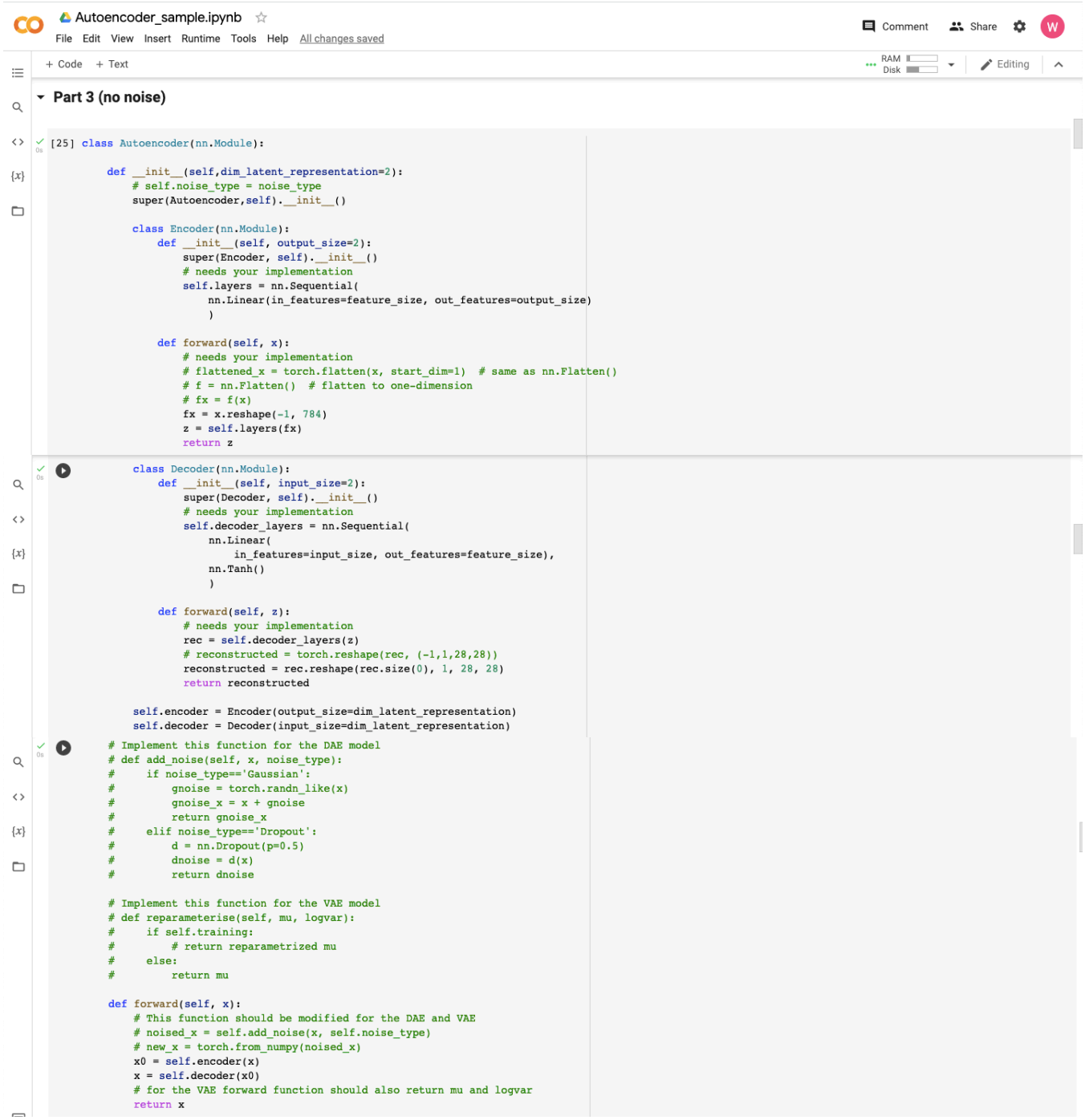
Describe how the plot differs from the one in part (1) and explain what this says about the architectures in this part and part (1). Why do you think the architecture in this part gave rise to the results shown in the scatter plot?

Points of different colors are separated more clearly and have longer between-groups distance with each other in part 2, and points of same colors gather more compactly than in part 1. Also, part 2's training and validation losses are relatively lower (around 0.65). This means part 2's architecture learns better and captures the images more accurately than part 1's architecture.

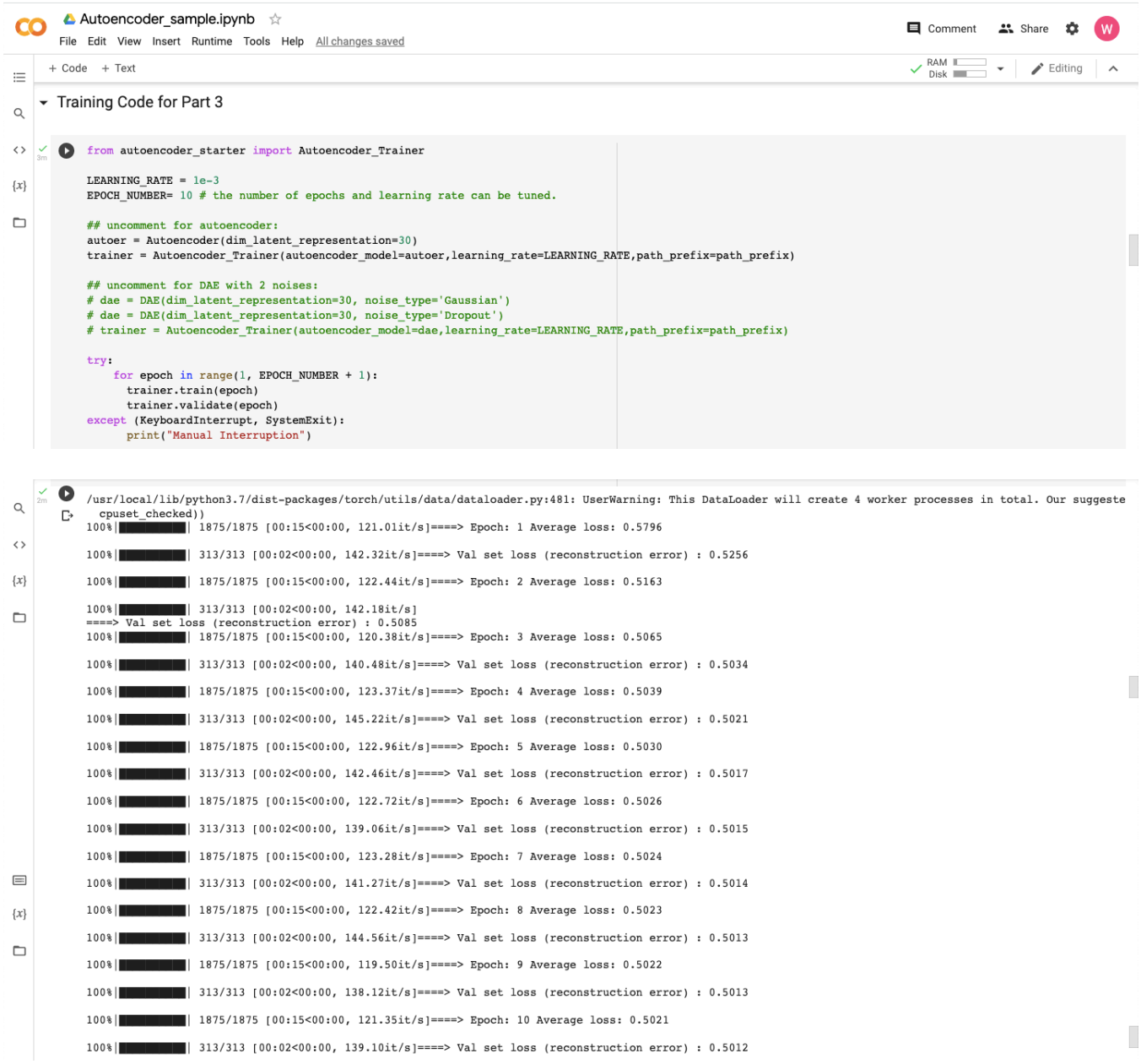
Why:
Part 1's architecture has only linear layer, while part 2's architecture has more layers and ReLU activation layer. Part 2's architecture has some non-linearity added in and is a more complex network, compared to part 1. More complex architecture with more layers enables the architecture to capture features accurately and learn better. If we only include linear activation functions in a neural network, the output will just be a linear transformation of the input, which may be not enough to form a universal function approximator.

#3

autoencoder with no noise



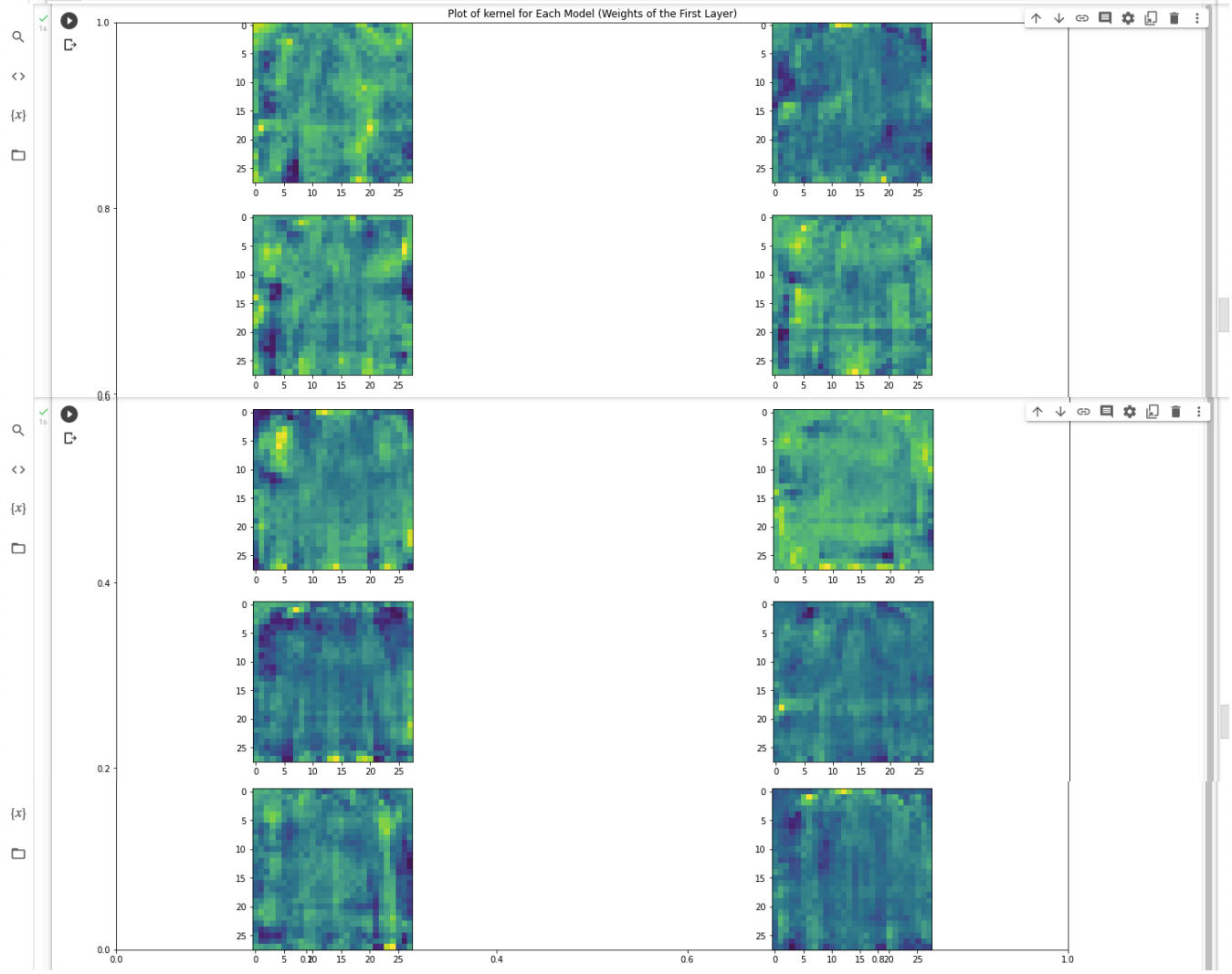
Training code & Losses (no noise):



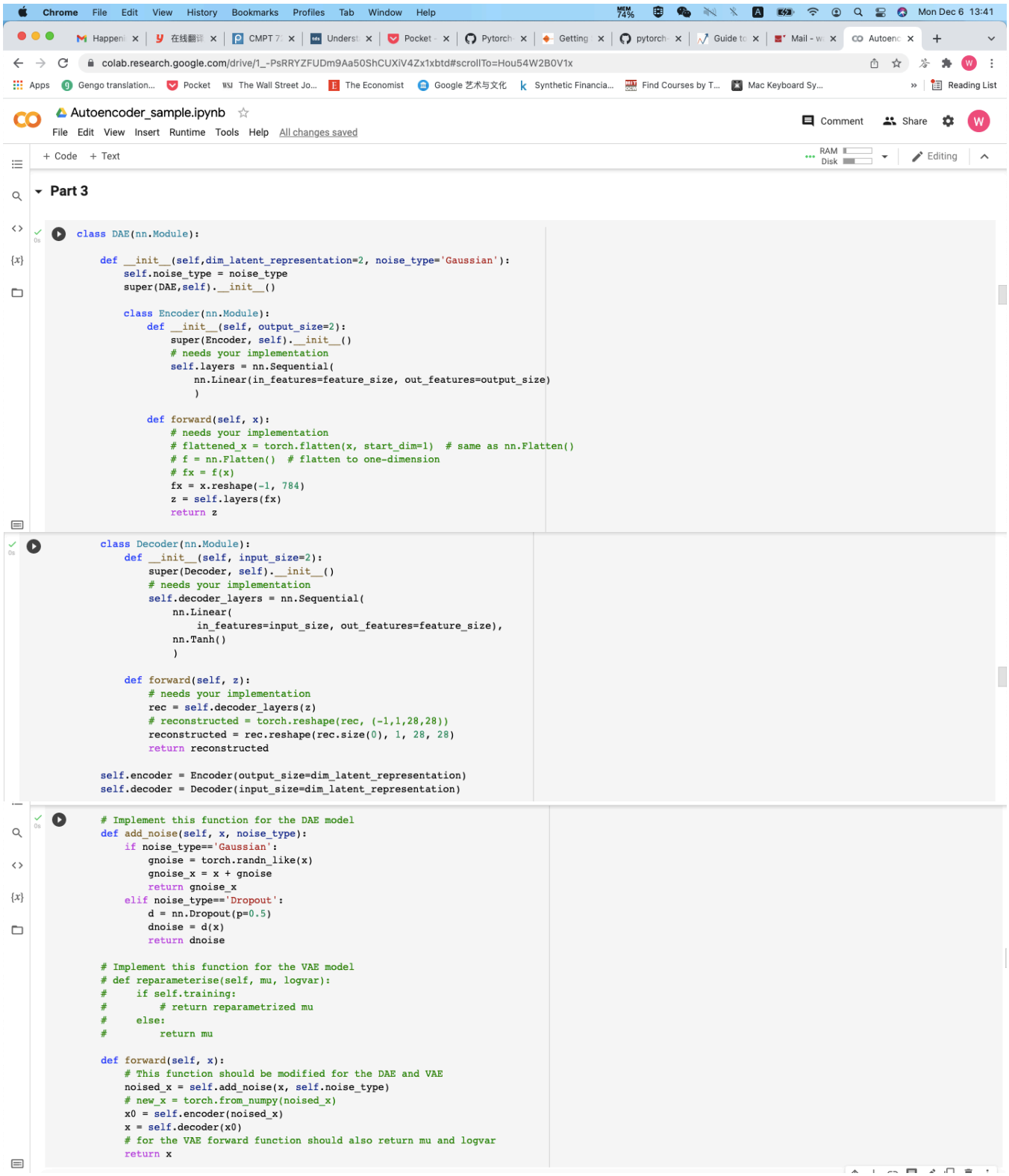
Kernel plots:

for Part 3 (kernel)

```
from autoencoder_starter import Plot_Kernel

## uncomment for autoencoder
Plot_Kernel(autoer)
## uncomment for DAE with 2 noises
# Plot_Kernel(dae)
```



DAE with Dropout noise



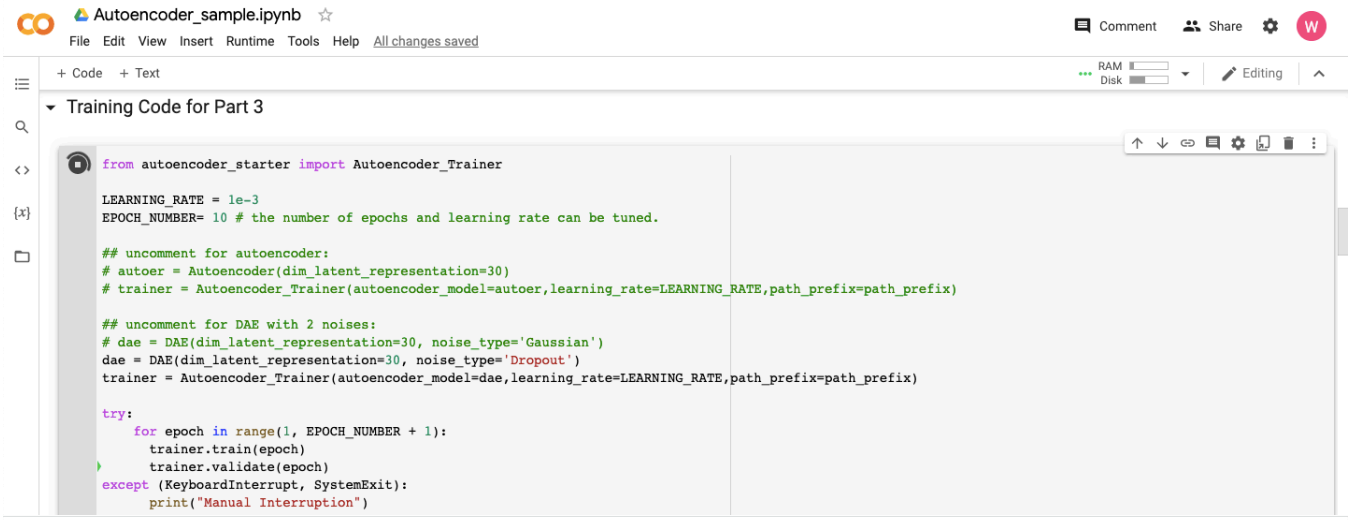
Part 3

```
class DAE(nn.Module):

    def __init__(self,dim_latent_representation=2, noise_type='Gaussian'):
        self.noise_type = noise_type
        super(DAE,self).__init__()

        class Encoder(nn.Module):
            def __init__(self, output_size=2):
                super(Encoder, self).__init__()
                # needs your implementation
                self.layers = nn.Sequential(
                    nn.Linear(in_features=feature_size, out_features=output_size)
                    )

            def forward(self, x):
                # needs your implementation
                # flattened_x = torch.flatten(x, start_dim=1)  # same as nn.Flatten()
                # f = nn.Flatten()  # flatten to one-dimension
                # fx = f(x)
                fx = x.reshape(-1, 784)
                z = self.layers(fx)
                return z
```

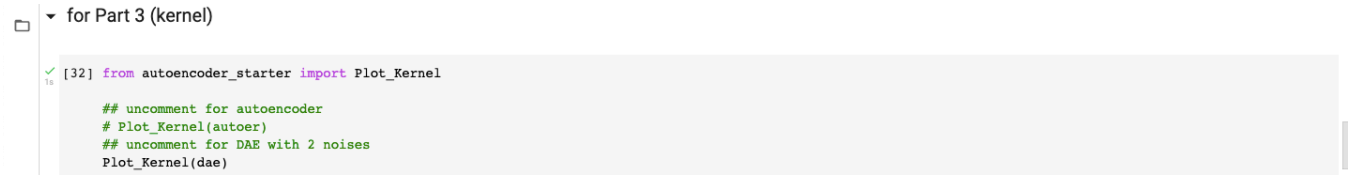
```
        class Decoder(nn.Module):
            def __init__(self, input_size=2):
                super(Decoder, self).__init__()
                # needs your implementation
                self.decoder_layers = nn.Sequential(
                    nn.Linear(
                        in_features=input_size, out_features=feature_size),
                    nn.Tanh()
                    )

            def forward(self, z):
                # needs your implementation
                rec = self.decoder_layers(z)
                # reconstructed = torch.reshape(rec, (-1,1,28,28))
                reconstructed = rec.reshape(rec.size(0), 1, 28, 28)
                return reconstructed

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

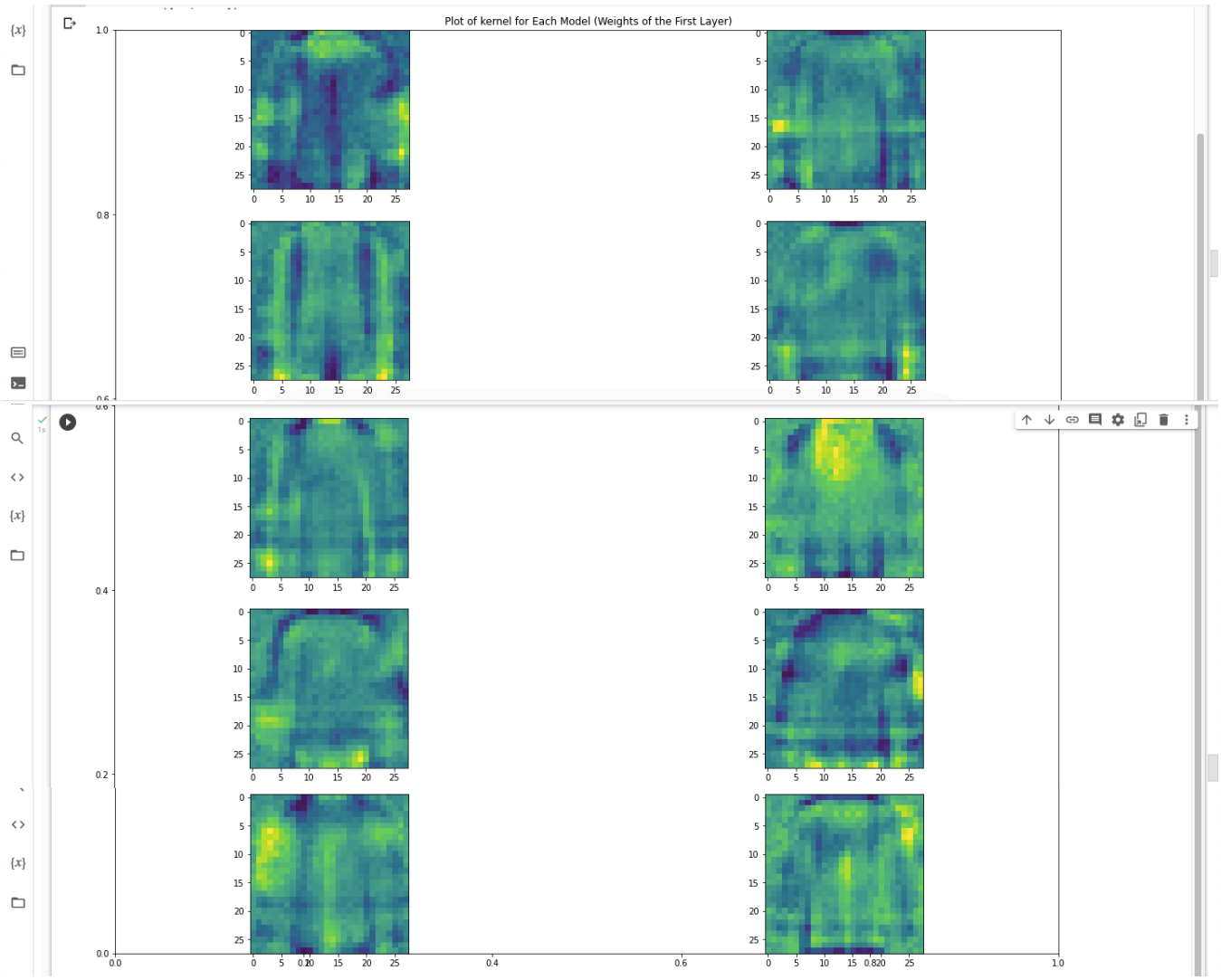
```
    # Implement this function for the DAE model
    def add_noise(self, x, noise_type):
        if noise_type=='Gaussian':
            gnoise = torch.randn_like(x)
            gnoise_x = x + gnoise
            return gnoise_x
        elif noise_type=='Dropout':
            d = nn.Dropout(p=0.5)
            dnoise = d(x)
            return dnoise

    # Implement this function for the VAE model
    # def reparameterise(self, mu, logvar):
    #     if self.training:
    #         # return reparametrized mu
    #     else:
    #         return mu

    def forward(self, x):
        # This function should be modified for the DAE and VAE
        noised_x = self.add_noise(x, self.noise_type)
        # new_x = torch.from_numpy(noised_x)
        x0 = self.encoder(noised_x)
        x = self.decoder(x0)
        # for the VAE forward function should also return mu and logvar
        return x
```

Training code & Losses for Dropout noise:



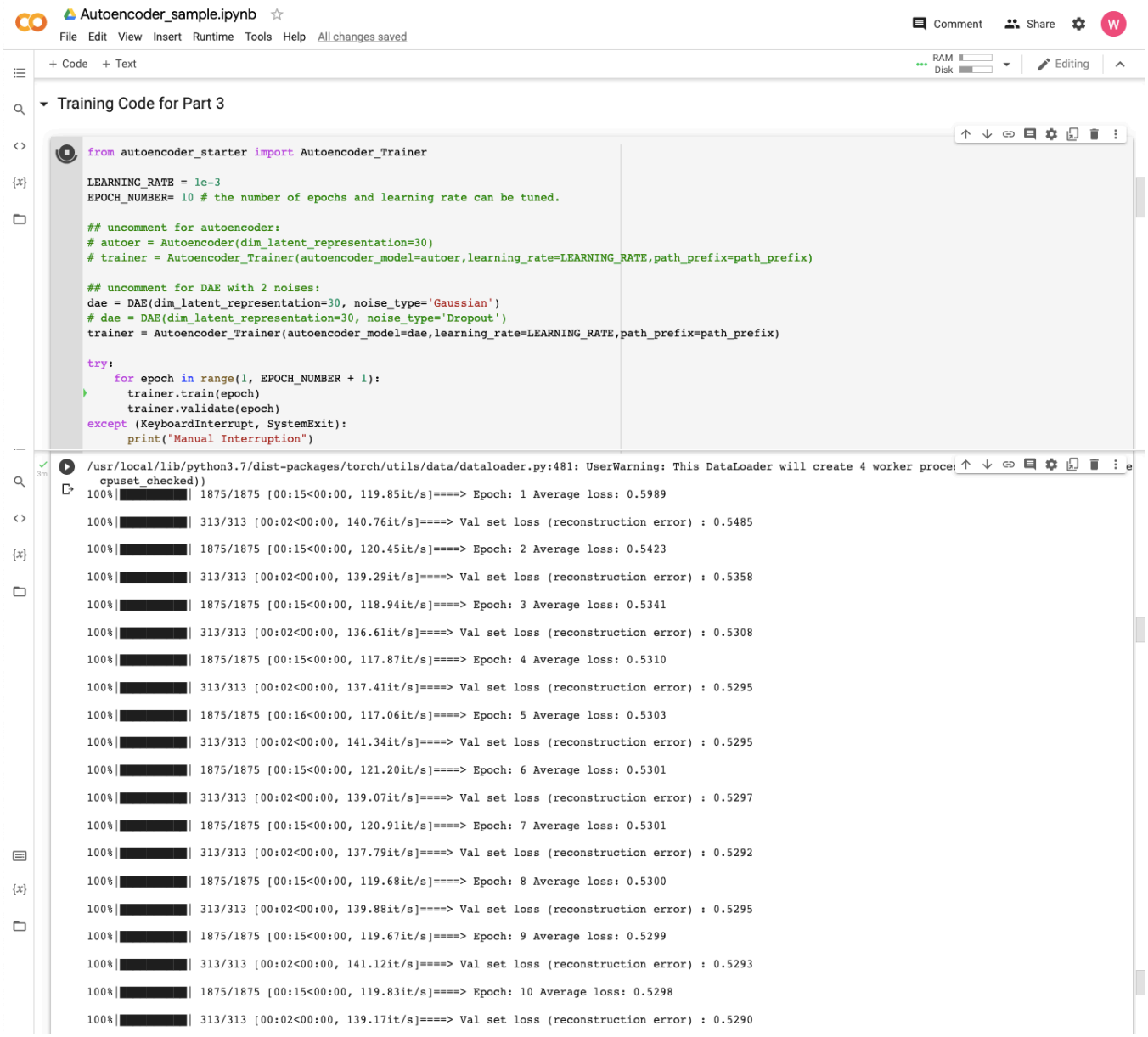
Kernel plots:



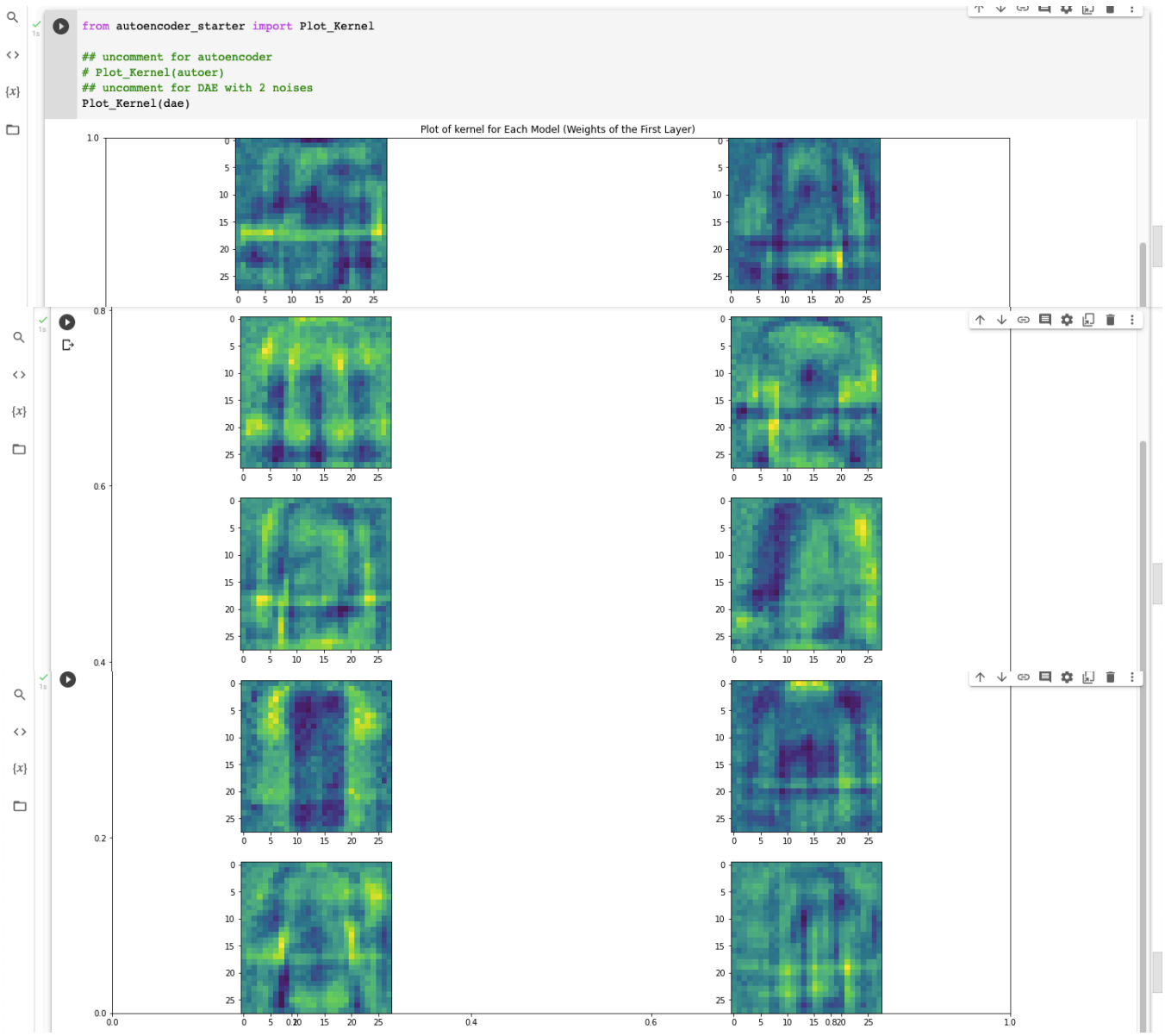


DAE with Gaussian noise

Training code & Losses for Gaussian noise:



Kernel plots:



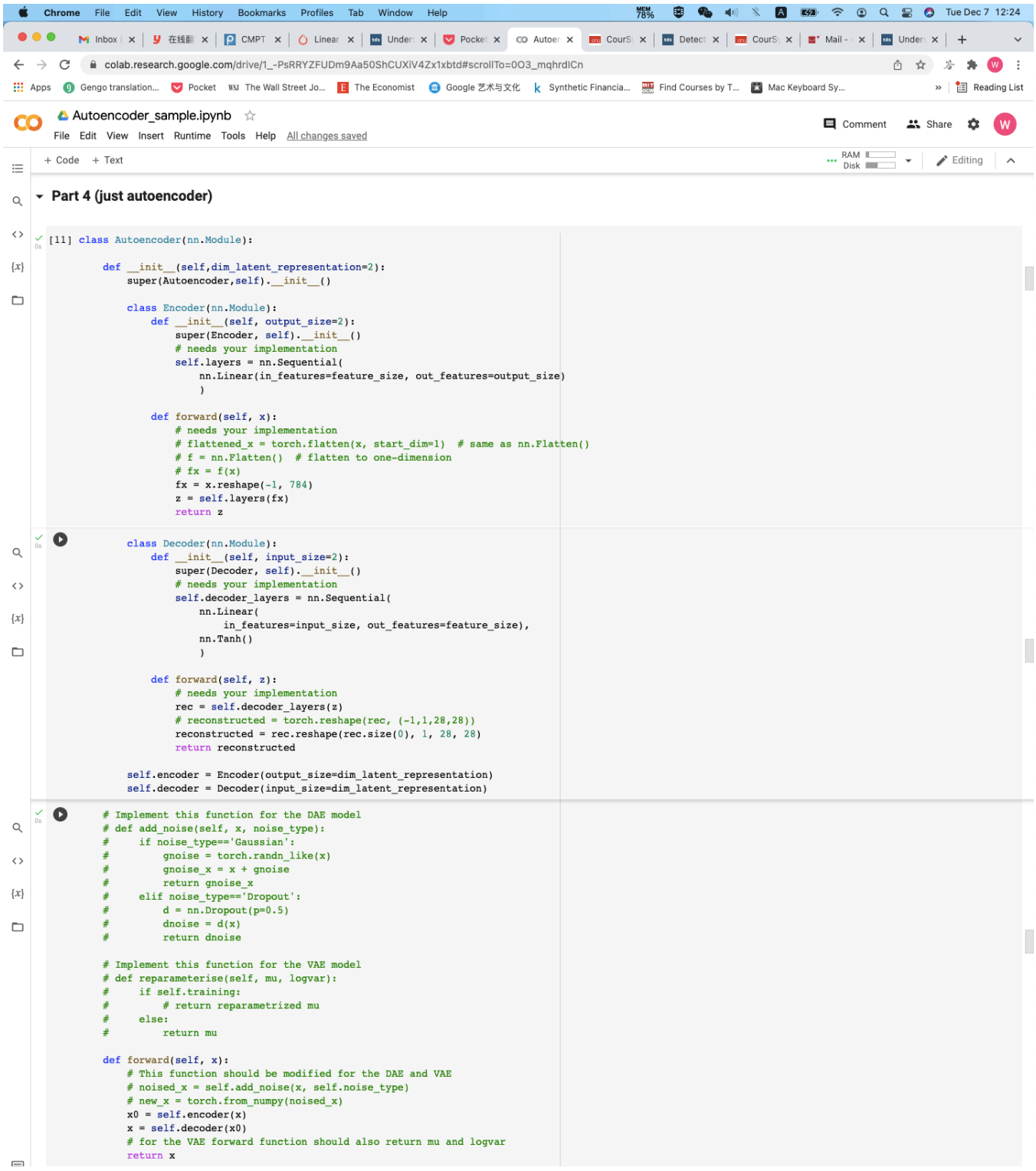
Describe how the kernels differ from each other and explain what this says about the denoising autoencoder.

Kernel of autoencoder with no noise removed is less clear and obvious than kernel of DAE. Kernel of DAE with Gaussian noise is a bit more obviously than kernel of DAE with Dropout noise. We can more easily define the shape of images in kernel of DAE.

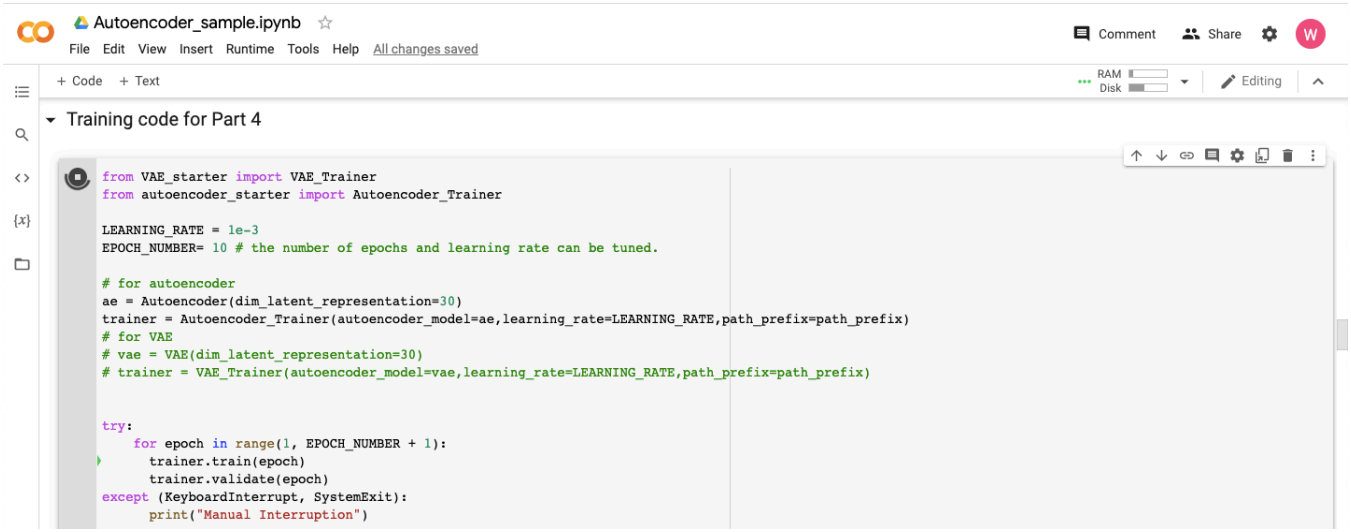
This implies that removing noise from input enables the network to ignore less important features ("noise") in images and learn more important features from the images. By removing noises, the risk of learning the identity function and learning from noises instead of extracting key features is reduced. Therefore, DAE learns and performs better than autoencoder without noises removed.

#4

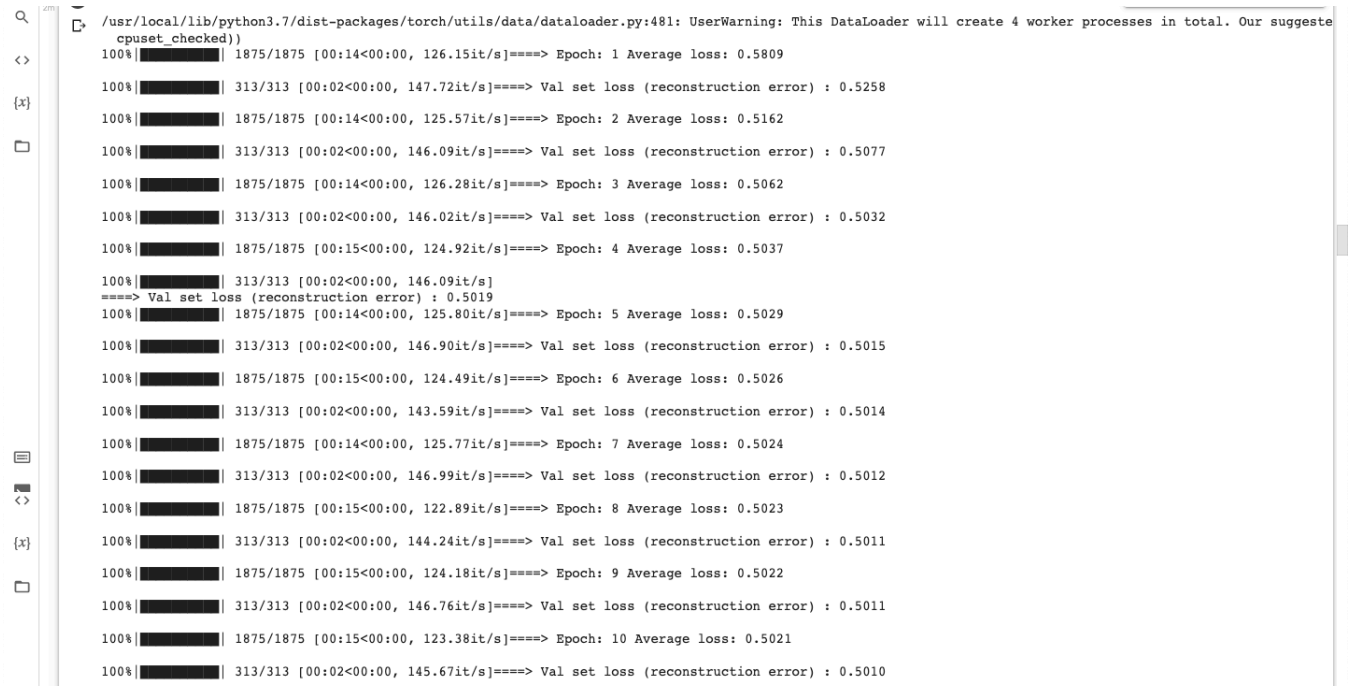
Autoencoder



Training code:



Losses:

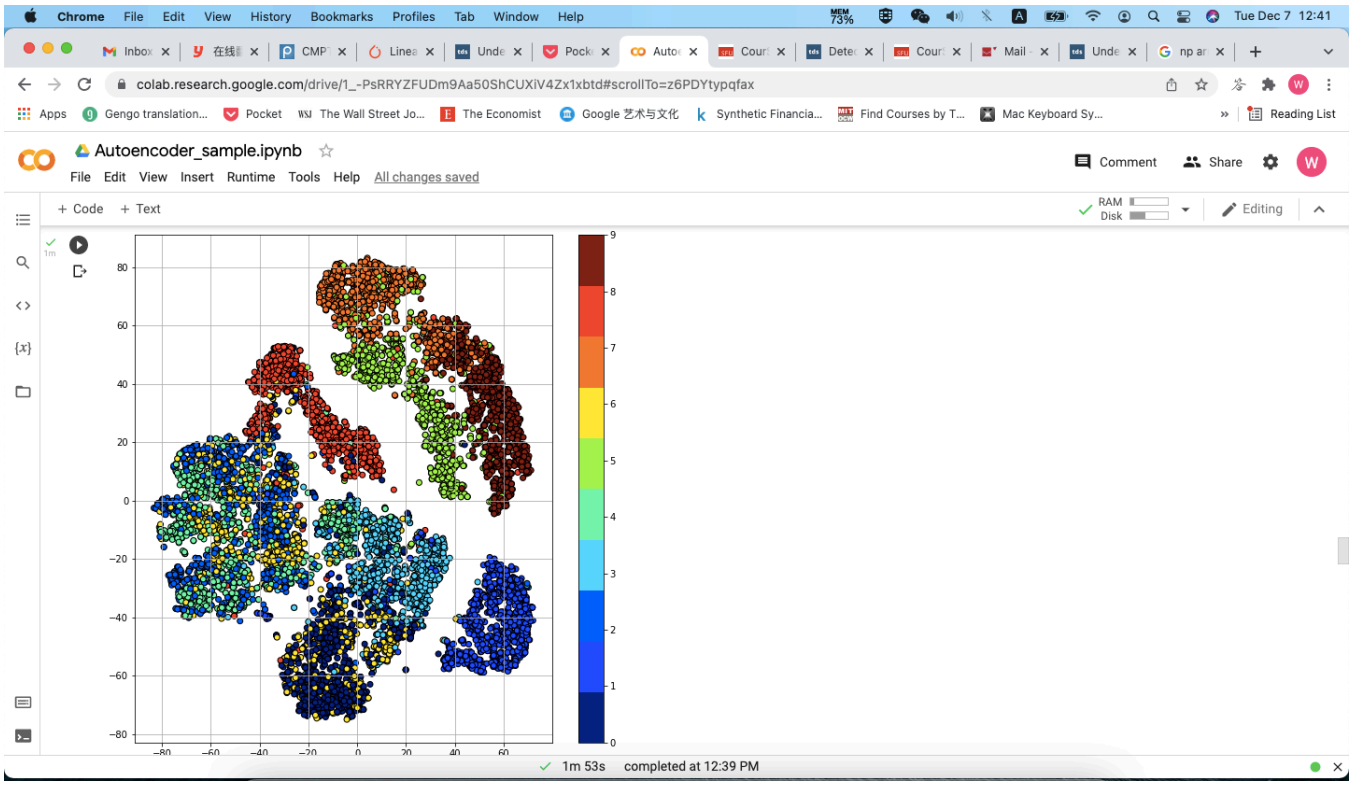


latent space plot code:

for Part 4 (TSNE - autoencoder)

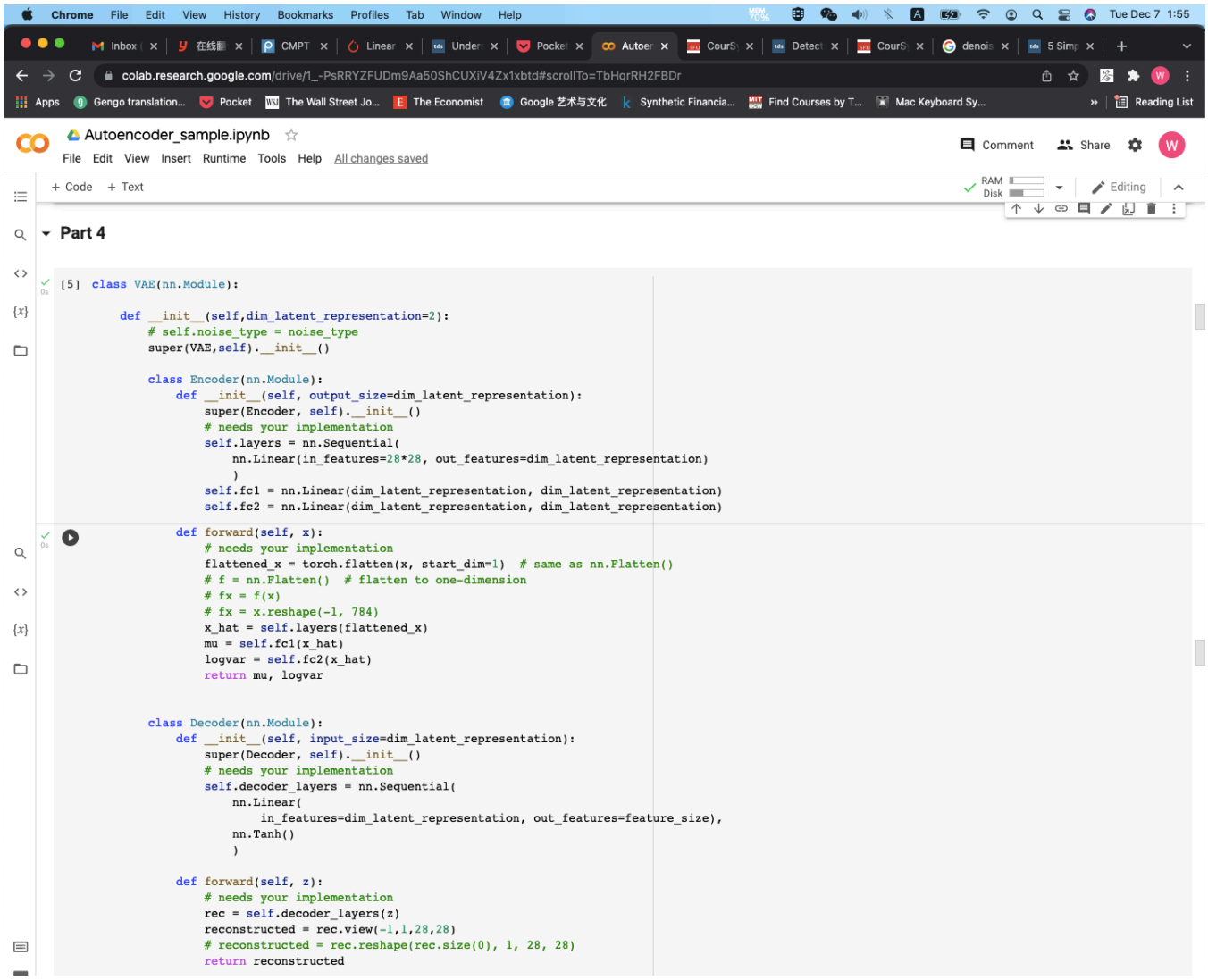
```
with torch.no_grad():
    model = trainer.model
    model.eval()
    z=[];label=[]
    for x,y in trainer.val_loader:

        z_ = model.encoder(x.to(trainer.device))
        z += z_.cpu().tolist()
        label += y.cpu().tolist()
    z = np.asarray(z)
    print('this is z shape:')
    print(z.shape)
    label = np.asarray(label)

from autoencoder_starter import scatter_plot
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2)
tsne_results = tsne.fit_transform(z)
scatter_plot(latent_representations=tsne_results,labels=label)
```

latent space plot:



VAE

Here, I use KLD loss term with mean, which is - 0.5 * torch.mean(1 + logvar - mu.pow(2) - logvar.exp()), since this definition makes more sense in the learning result than the one with sum.


Chrome File Edit View History Bookmarks Profiles Tab Window Help
Tue Dec 7 1:55
colab.research.google.com/drive/1_-PsRRYZFUDm9Aa50ShCUXiV4Zx1xbtd#scrollTo=TbHqrRH2FBDr
Autoencoder_sample.ipynb
File Edit View Insert Runtime Tools Help All changes saved
Comment
Share
+ Code + Text
RAM
Disk
Editing
Part 4
[5] class VAE(nn.Module):
def __init__(self,dim_latent_representation=2):
# self.noise_type = noise_type
super(VAE,self).__init__()
class Encoder(nn.Module):
def __init__(self, output_size=dim_latent_representation):
super(Encoder, self).__init__()
# needs your implementation
self.layers = nn.Sequential(
nn.Linear(in_features=28*28, out_features=dim_latent_representation)
)
self.fc1 = nn.Linear(dim_latent_representation, dim_latent_representation)
self.fc2 = nn.Linear(dim_latent_representation, dim_latent_representation)
def forward(self, x):
# needs your implementation
flattened_x = torch.flatten(x, start_dim=1) # same as nn.Flatten()
# f = nn.Flatten() # flatten to one-dimension
# fx = f(x)
# fx = x.reshape(-1, 784)
x_hat = self.layers(flattened_x)
mu = self.fc1(x_hat)
logvar = self.fc2(x_hat)
return mu, logvar
class Decoder(nn.Module):
def __init__(self, input_size=dim_latent_representation):
super(Decoder, self).__init__()
# needs your implementation
self.decoder_layers = nn.Sequential(
nn.Linear(
in_features=dim_latent_representation, out_features=feature_size),
nn.Tanh()
)
def forward(self, z):
# needs your implementation
rec = self.decoder_layers(z)
reconstructed = rec.view(-1,1,28,28)
# reconstructed = rec.reshape(rec.size(0), 1, 28, 28)
return reconstructed

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

    # # Implement this function for the DAE model
    # def add_noise(self, x, noise_type):
    #     if noise_type=='Gaussian':
    #         gnoise = torch.randn_like(x)
    #         gnoise_x = x + gnoise
    #         return gnoise_x
    #     elif noise_type=='Dropout':
    #         d = nn.Dropout(p=0.5)
    #         dnoise = d(x)
    #         return dnoise

    # Implement this function for the VAE model
    def reparameterise(self, mu, logvar):
        if self.training:
            std = torch.exp(logvar/2)
            eps = torch.randn_like(std)
            reparametrized_mu = mu + eps * std
            return reparametrized_mu
        else:
            return mu
```

```
[5]
    def forward(self, x):
        # This function should be modified for the DAE and VAE
        # noised_x = self.add_noise(x, self.noise_type)
        # new_x = torch.from_numpy(noised_x)
        # x0 = self.encoder(noised_x)
        # x1 = self.decoder(x0)
        # for the VAE forward function should also return mu and logvar
        mu, logvar = self.encoder(x)
        # logvar = self.encoder(x)
        z = self.reparameterise(mu, logvar)
        reconstruction = self.decoder(z)
        return reconstruction, mu, logvar
```

Training Code:

Autoencoder_sample.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Comment Share W

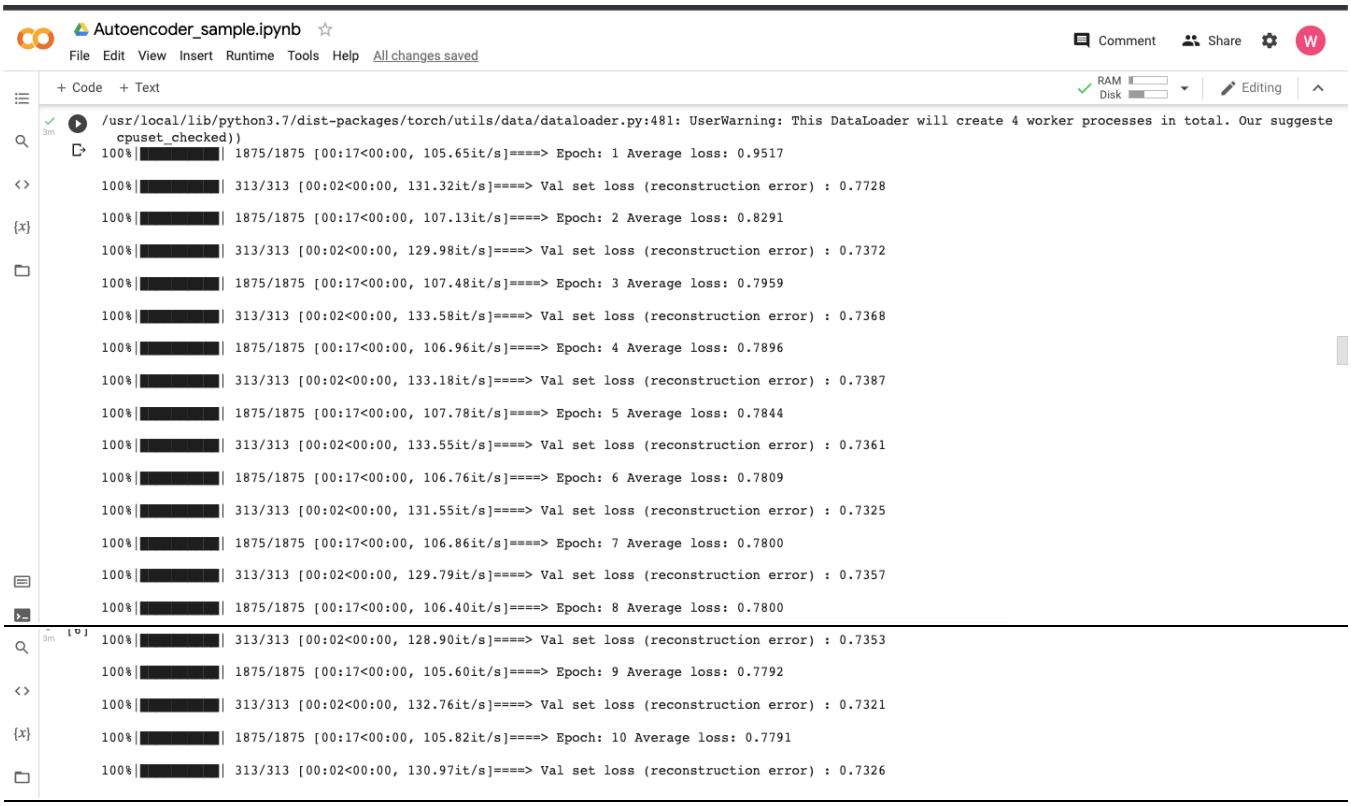
+ Code + Text

RAM Disk Editing

Training code for Part 4

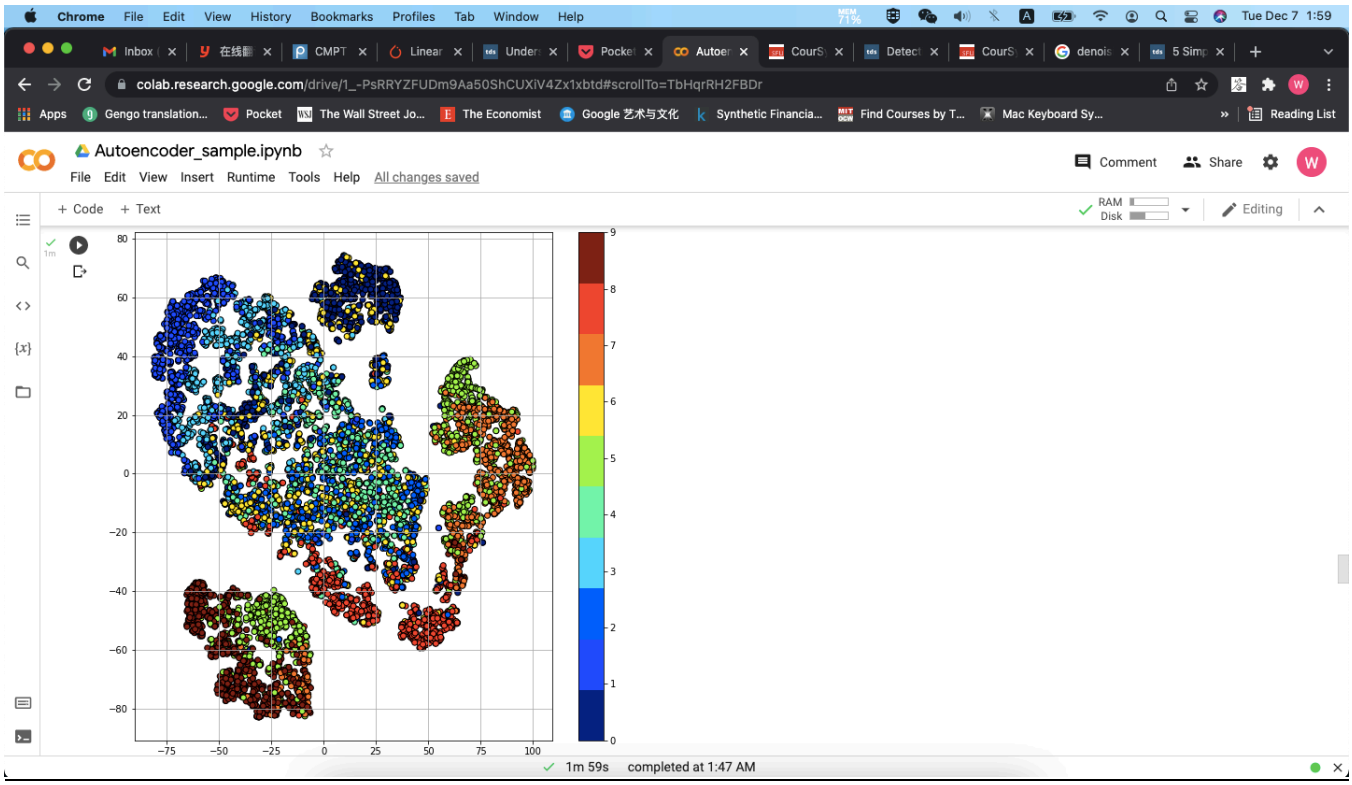
```
from VAE_starter import VAE_Trainer
from autoencoder_starter import Autoencoder_Trainer

LEARNING_RATE = 1e-3
EPOCH_NUMBER= 10 # the number of epochs and learning rate can be tuned.

# for autoencoder
ae = Autoencoder(dim_latent_representation=30)
trainer = Autoencoder_Trainer(autoencoder_model=ae,learning_rate=LEARNING_RATE,path_prefix=path_prefix)
# for VAE
# vae = VAE(dim_latent_representation=30)
# trainer = VAE_Trainer(autoencoder_model=vae,learning_rate=LEARNING_RATE,path_prefix=path_prefix)


try:
    for epoch in range(1, EPOCH_NUMBER + 1):
      trainer.train(epoch)
      trainer.validate(epoch)
except (KeyboardInterrupt, SystemExit):
      print("Manual Interruption")
```

Losses:

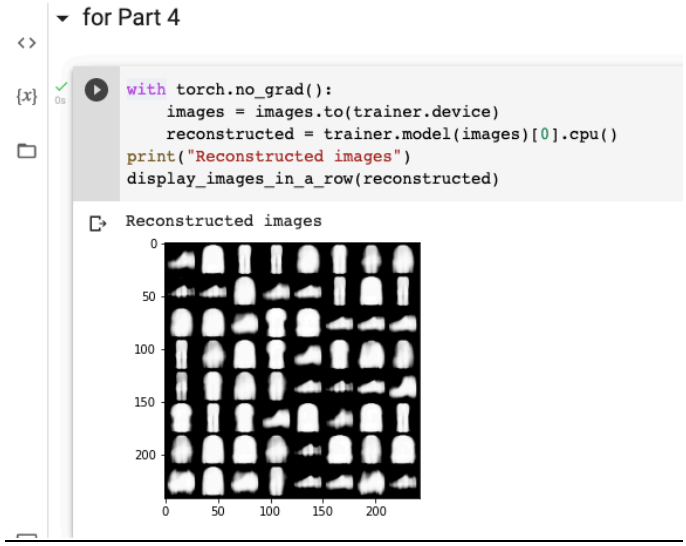


latent space plot code:

latent space plot:



Reconstructed image:



Describe how the latent features are differ from each other for two models and explain what this says about the variational autoencoders.

Points with different colors gathers more compactly and the between-groups distance is larger in autoencoder than in VAE. Although it is obvious that points with same colors gather in VAE's plot and we can clearly define different color groups of points, their in-group distance is larger than latent

features in autoencoder does. It seems that autoencoder with 30-dimension latent space performs better in capturing the images than its counterpart VAE does. Autoencoder here may be overfitting.

First, VAE modeling with Gaussians allows each dimension in the representation to push themselves as farther as possible from the other factors. Therefore, not so compactly gather as autoencoder does. It also implies that VAE introduces regularization during training process for counterpart autoencoder and avoid overfitting to ensure that the latent space has good properties that enable generative purpose.