Dan Brody
Professor Curro
Deep Learning
29 October 2020

Deep Learning Midterm Write-Up

Introduction

The paper that I chose was about the implicit rank-minimizing autoencoder. The basis of the paper was that the autoencoder was designed to implicitly minimize the rank/dimensionality of the autoencoder and learn representations with low effective dimensionality, creating a regularization effect. *An overview of low-rank matrix recovery from incomplete observations* by Mark A. Davenport and Justin Romberg showed that, given a unknown matrix W and unseen entries, if the matrix W is low rank then various algorithms can achieve approximate or exact recovery on the unseen entries. This regularization technique was achieved through having linear layers between the encoder and decoder, with corresponding latent dimensions to the data, where each linear layer is considered as a square matrix with a of size latent dimension x latent dimension. The reason that this process works is the following: *Implicit regularization in matrix factorization* by Suriya Guneskar, et al. studied implicit regularization on shallow (depth-2) matrix factorization by considering recovery of a positive semidefinite matrix from sensing via symmetric measurements (represented by the equation below)

$$min_{W \epsilon S_+^d} l(W) \coloneqq \sum_{i=1}^{m} (y_i - < A_i, W >) \text{ [1]}$$

(A_i = symmetric and linearly independent matrix $\in \mathbb{R}^{d,d}$, S_+^d stands for the set of symmetric and positive semidefinite matrices $\in \mathbb{R}^{d,d}$, $l = loss$)

on the objective:

$$\psi(Z) = l(ZZ^T) = \frac{1}{2} \sum_{i=1}^{m} (y_i - < A_i, ZZ^T >)^2 \text{ [1]}$$

(ZZ^T = final solution, Z = symmetric linear matrix $\in \mathbb{R}^{d,d}$)

Given that $W_{shallow,inf}(\alpha) \coloneqq \lim_{t \to \infty} (Z(t)Z(t)^t)$, *Implicit regularization in matrix factorization* proved that $W_{shallow}$ is a global optimum with minimal nuclear norm. If we instead take the objective function and change it to, instead of multiplying two symmetric linear matrices, multiplying multiple linear square matrices then we receive the following:

$$\psi(W_1 W_2 \ldots \ldots W_N) = l(W_N W_{N-1} \ldots \ldots W_1) = \frac{1}{2} \sum_{i=1}^{m} (y_i - < A_i, W_N W_{N-1} \ldots \ldots W_1 >)^2 \text{[1]}$$

($(W_1 W_2 \ldots \ldots W_N)$ = square matrices $\in \mathbb{R}^{d,d}$)

Since this condition gives the same properties of that for the shallow matrix factorization described above, then the following holds: for depth greater than or equal to 3, W_{deep} is a global optimum with minimal nuclear norm. I have henceforth shown the regularizing effect. Nuclear norm minimization combined with a low rank solution results in a quicker convergence (since the nuclear norm is the sum of the singular values). A faster nuclear norm minimization, therefore will result in a quicker convergence. As the number of linear layers between the encoder and decoder increase so does the regularizing effect.

Specifications

In this paper I have replicated figure 2 and 3 (shown below)

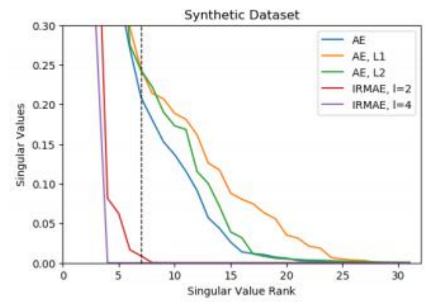


Figure 2: Singular values of the latent space of each model on synthetic shape dataset. Each curve represents singular values of the covariance matrix of the code computed on the validation set. IRMAE $l = 2$ is able to approach the minimal theoretical rank of 7.

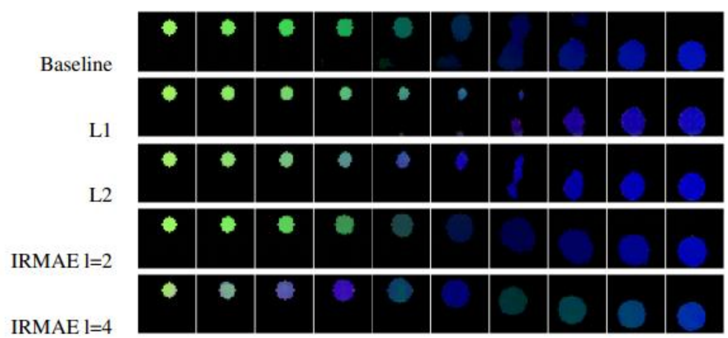


Figure 3: Linear interpolation between two randomly generated samples. From top to bottom are results from baseline unregularized AE, AE with L1 regularization, AE with L2 regularization, IRMAE $l = 2$, IRMAE $l = 4$.

Figure 1: *IRMAE Figures to Replicate*

On the synthetic shape dataset the following model architecture was described and the following instructions were described:

Dataset	Shape
Encoder	$x \in \mathcal{R}^{32x32x3}$ $\rightarrow$ Conv$_{32}$ $\rightarrow$ ReLU $\rightarrow$ Conv$_{64}$ $\rightarrow$ ReLU $\rightarrow$ Conv$_{128}$ $\rightarrow$ ReLU $\rightarrow$ Conv$_{256}$ $\rightarrow$ ReLU $\rightarrow$ Conv$_{32}$ $\rightarrow$ ReLU $\rightarrow z \in \mathcal{R}^{32}$
Decoder	$z \in \mathcal{R}^{32}$ $\rightarrow$ ConvT$_{256}$ $\rightarrow$ ReLU $\rightarrow$ ConvT$_{128}$ $\rightarrow$ ReLU $\rightarrow$ ConvT$_{64}$ $\rightarrow$ ReLU $\rightarrow$ ConvT$_{32}$ $\rightarrow$ ReLU $\rightarrow$ ConvT$_{3}$ $\rightarrow$ Tanh $\rightarrow \hat{x} \in \mathcal{R}^{32x32x3}$

(where all convolutional layers had padding of 1, stride 0f of 2 and a 4x4 kernel size).

Figure 2: *Model Architecture*

For the synthetic shape dataset, we generate shape images on the fly. The size of each shape is uniformly sampled between 3 and 8, inclusively. The color is uniformly sampled in RGB. The coordinate of the center of the shape is randomly sampled with x and y between 8 and 24, inclusively.

Figure 3*: Synthetic Shape Dataset Instructions*

Dataset	Shape
learning rate	0.0001
epochs	100
latent dimension	32
batch size	32
training examples	50000
evaluation examples	10000

Figure 4: *Synthetic Shape Dataset Hyperparameters*

Experimentation

The tanh in the architecture was particularly interesting because images should have pixels between 0 and 1 (as floats) or 0 and 255 (as integers). Additionally, in the interpolation I employed, when the negative values were filtered out the images were reconstructed very well, showing that there may be some mechanism that filters out incorrect values as negative values. I suspect that there is a possibility that inputs used by the paper contained negative pixels and this was allowed because the inputs and outputs were converted to PIL (which accepts negative pixels), however this pre-processing technique was never denoted in the paper. Because of this, I decided to do some experimentation. I found a paper that had Yann LeCun (one of the authors *of Implicit Rank-Minimizing Autoencoder*) as one of the authors of a paper (*Efficient Backprop*) that denoted a method to pre-process the inputs where the inputs transform to be in both the positive and negative regions. I decided to employ that. The three steps are converting the inputs to 0-mean, decorrelating the inputs, and doing covariance equalization. I was only able to accomplish one out of the three steps because the others would not make sense considering the dataset I was using. Since the dataset consisted of two shapes of varying scale decorrelating would defeat the purpose. Given that decorrelating the inputs would not be practical, any steps afterward are nullified. Below is an example of an image I employed 0-mean on (using StandardScaler) and another that I tried to decorrelate.

```
In [83]: to_pil_image= transforms.ToPILImage()
         x = createSquareNewestTry()
         to_pil_image(x)
Out[83]:
```

```
In [84]: torch.mean(x)
Out[84]: tensor(-1.5522e-10)

In [85]: torch.min(x)
Out[85]: tensor(-1.2863)
```

```
from sklearn.decomposition import PCA
pca = PCA()
new_train_tensor = torch.FloatTensor(np.resh

#not possible to decorrelate
to_pil_image= transforms.ToPILImage()
to_pil_image(new_train_tensor[10])
```

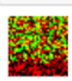


Figure 5: *0-mean Image*

Figure 6: *Decorrelated Image*

Unfortunately I did not have enough time to train the inputs but, given more time, the results may have been interesting.

Another method that I thought would be interesting to experiment with would be to use a sigmoid at the end of the architecture (suggested by professor Curro) so that the output values would be between 0 and 1. Unfortunately I did not have enough time to see this through, however given more time the results may have been interesting.

Project

As for the figures that were assigned to me, I employed the paper's instructions and architecture for the shape dataset as well as used OpenCV to create the circles and squares. I found, however, for the shape dataset that followed the instructions the paper had laid out, the interpolation was not very accurate. Due to this, I minimized the shape of the squares by half. This produced better results because the smaller squares would be more similar to the circles and could act as better building blocks (since the circles are made up of squares). In this case, I propose two implementations of figure 2 and figure 3. . I also could not find the regularization coefficient for L1 and L2 regularization anywhere in the paper so I experimented with the regularization coefficients to find one that was not too large (to prevent underfitting) and not too small (to create meaningful results) and settled on 1e-10.

The first implementation will be using smaller squares.

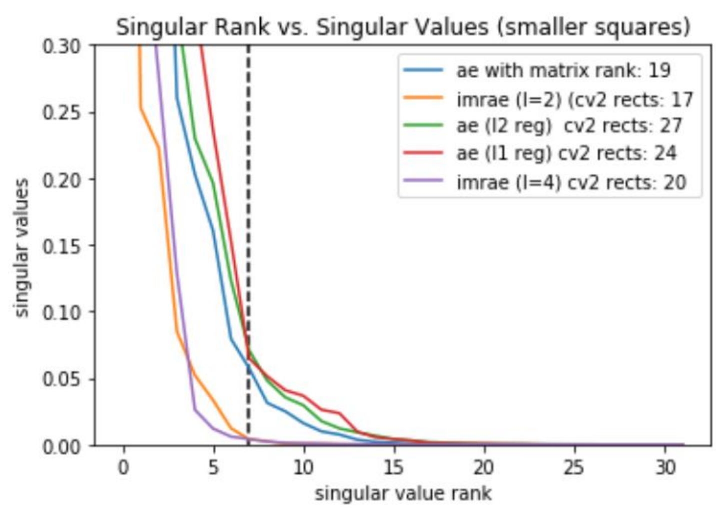


Figure 7: *Singular Value Rank vs. Singular Values for Synthetic Shape Dataset with Smaller Squares*

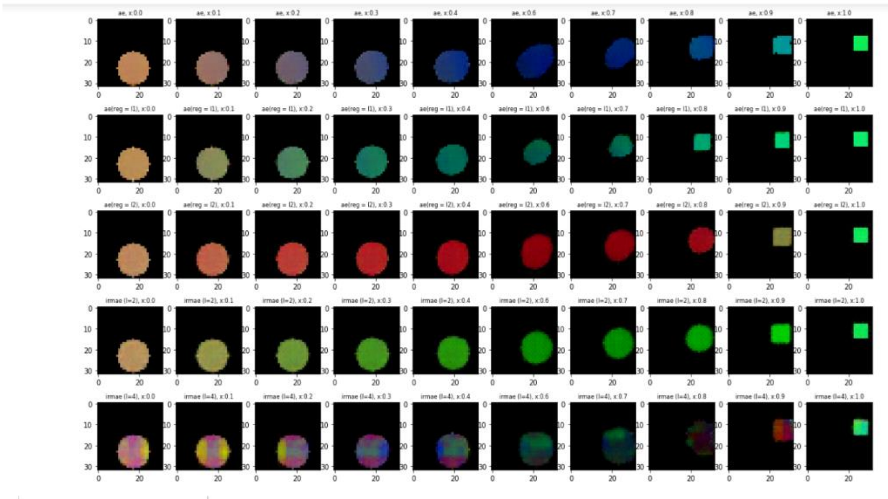


Figure 8: *Linear Interpolation on Test Set for Synthetic Shape Dataset with Smaller Squares*

For IRMAE(l=4) the results were incorrect because they overfit. When I used the model against the training set I was able to get excellent results but, as seen here in figure 8, the model does not work well on the test set. I suspect this is because the dataset is easier to learn and, therefore, more prone to overfitting.

The second implementation will be using the dataset outlined in the paper

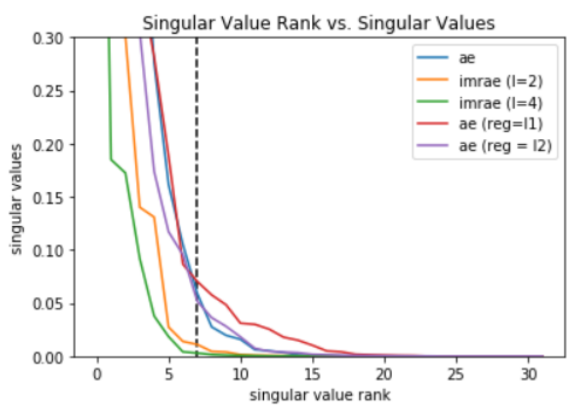


Figure 9: *Singular Value Rank vs. Singular Values for Synthetic Shape Dataset*

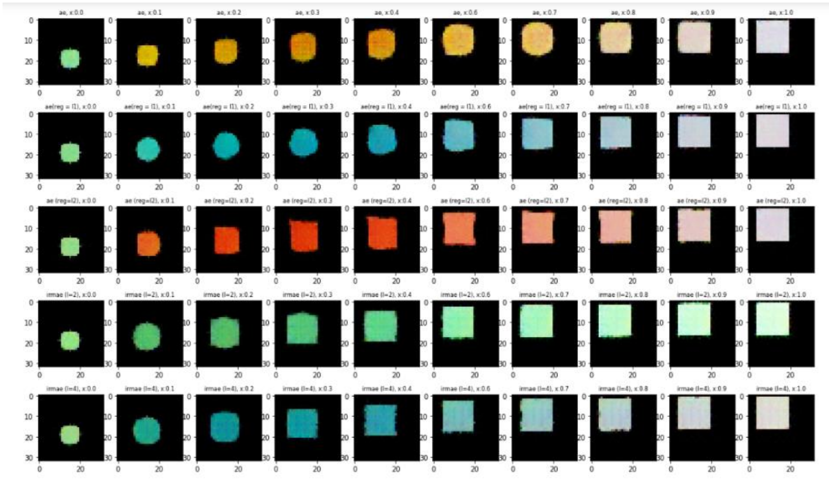


Figure 10: *Linear Interpolation on Test Set for Synthetic Shape Dataset*

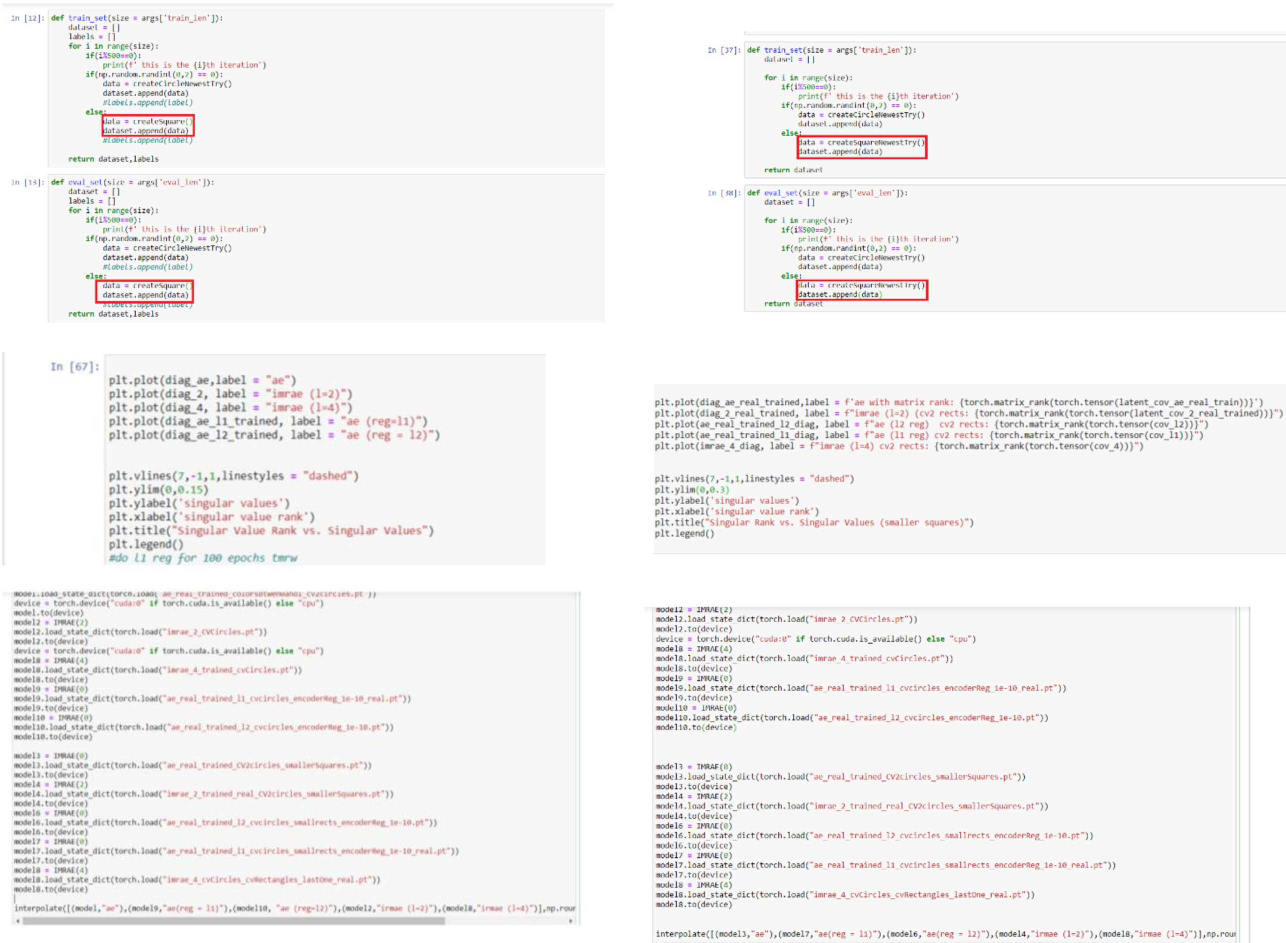
Both of my implementations were successful in that they showed that using Implicit Rank-Minimizing Autoencoder (IRMAE) would result in a low-rank and low dimensionality for the autoencoder while L1 and L2 regularization on the hidden code would take up a large latent space with a higher matrix rank than the baseline autoencoder (through figures 7 and 9). Both implementations also showed that IRMAE (l=2) approached the minimum rank of 7 (through figures 7 and 9). Figures 8 and 10 also showed that IRMAE has better representation than the baseline autoencoder (AE) where the autoencoder had some artifacts. The results could have been improved if the regularization coefficients were specified and the shape dataset that the creators had made was given.

Works Cited

[1] Sanjeev Arora, Nadav Cohen, Wei Hu, and Yuping Luo. Implicit regularization in deep matrix factorization. In Advances in Neural Information Processing Systems (NeurIPS ’19), pages 7413–7424, 2019.

Appendix:

The only difference between the two files for the two different datasets is the following:



```
In [12]: def train_set(size = args['train_len']):
    dataset = []
    labels = []
    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
            #labels.append(label)
        else:
            data = createSquare()
            dataset.append(data)
            #labels.append(label)

    return dataset,labels

In [13]: def eval_set(size = args['eval_len']):
    dataset = []
    labels = []
    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
            #labels.append(label)
        else:
            data = createSquare()
            dataset.append(data)
            #labels.append(label)
    return dataset,labels
```

```
In [37]: def train_set(size = args['train_len']):
    dataset = []

    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
        else:
            data = createSquareNewestTry()
            dataset.append(data)

    return dataset

In [38]: def eval_set(size = args['eval_len']):
    dataset = []

    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
        else:
            data = createSquareNewestTry()
            dataset.append(data)
    return dataset
```

```
In [67]:
plt.plot(diag_ae,label = "ae")
plt.plot(diag_2, label = "imrae (l=2)")
plt.plot(diag_4, label = "imrae (l=4)")
plt.plot(diag_ae_l1_trained, label = "ae (reg=l1)")
plt.plot(diag_ae_l2_trained, label = "ae (reg = l2)")


plt.vlines(7,-1,1,linestyles = "dashed")
plt.ylim(0,0.15)
plt.ylabel('singular values')
plt.xlabel('singular value rank')
plt.title("Singular Value Rank vs. Singular Values")
plt.legend()
#do l1 reg for 100 epochs tmrw
```

```
plt.plot(diag_ae_real_trained,label = f'ae with matrix rank: {torch.matrix_rank(torch.tensor(latent_cov_ae_real_train))}')
plt.plot(diag_2_real_trained, label = f"imrae (l=2) (cv2 rects: {torch.matrix_rank(torch.tensor(latent_cov_2_real_trained))}")
plt.plot(ae_real_trained_l2_diag, label = f"ae (l2 reg)  cv2 rects: {torch.matrix_rank(torch.tensor(cov_l2))}")
plt.plot(ae_real_trained_l1_diag, label = f"ae (l1 reg) cv2 rects: {torch.matrix_rank(torch.tensor(cov_l1))}")
plt.plot(imrae_4_diag, label = f"imrae (l=4) cv2 rects: {torch.matrix_rank(torch.tensor(cov_4))}")


plt.vlines(7,-1,1,linestyles = "dashed")
plt.ylim(0,0.3)
plt.ylabel('singular values')
plt.xlabel('singular value rank')
plt.title("Singular Rank vs. Singular Values (smaller squares)")
plt.legend()
```

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)
model2 = IMRAE(2)
model2.load_state_dict(torch.load("imrae_2_CVCircles.pt"))
model2.to(device)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model8 = IMRAE(4)
model8.load_state_dict(torch.load("imrae_4_trained_cvCircles.pt"))
model8.to(device)
model9 = IMRAE(0)
model9.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_encoderReg_1e-10_real.pt"))
model9.to(device)
model10 = IMRAE(0)
model10.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_encoderReg_1e-10.pt"))
model10.to(device)

model3 = IMRAE(0)
model3.load_state_dict(torch.load("ae_real_trained_CV2circles_smallerSquares.pt"))
model3.to(device)
model4 = IMRAE(2)
model4.load_state_dict(torch.load("imrae_2_trained_real_CV2circles_smallerSquares.pt"))
model4.to(device)
model6 = IMRAE(0)
model6.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_smallrects_encoderReg_1e-10.pt"))
model6.to(device)
model7 = IMRAE(0)
model7.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_smallrects_encoderReg_1e-10_real.pt"))
model7.to(device)
model8 = IMRAE(4)
model8.load_state_dict(torch.load("imrae_4_cvCircles_cvRectangles_lastOne_real.pt"))
model8.to(device)

interpolate([(model,"ae"),(model9,"ae(reg - l1)"),(model10, "ae (reg=l2)"),(model2,"irmae (l=2)"),(model8,"irmae (l=4)")],np.rour
```

```
model2 = IMRAE(2)
model2.load_state_dict(torch.load("imrae_2_CVCircles.pt"))
model2.to(device)
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model8 = IMRAE(4)
model8.load_state_dict(torch.load("imrae_4_trained_cvCircles.pt"))
model8.to(device)
model9 = IMRAE(0)
model9.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_encoderReg_1e-10_real.pt"))
model9.to(device)
model10 = IMRAE(0)
model10.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_encoderReg_1e-10.pt"))
model10.to(device)



model3 = IMRAE(0)
model3.load_state_dict(torch.load("ae_real_trained_CV2circles_smallerSquares.pt"))
model3.to(device)
model4 = IMRAE(2)
model4.load_state_dict(torch.load("imrae_2_trained_real_CV2circles_smallerSquares.pt"))
model4.to(device)
model6 = IMRAE(0)
model6.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_smallrects_encoderReg_1e-10.pt"))
model6.to(device)
model7 = IMRAE(0)
model7.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_smallrects_encoderReg_1e-10_real.pt"))
model7.to(device)
model8 = IMRAE(4)
model8.load_state_dict(torch.load("imrae_4_cvCircles_cvRectangles_lastOne_real.pt"))
model8.to(device)


interpolate([(model3,"ae"),(model7,"ae(reg = l1)"),(model6,"ae(reg = l2)"),(model4,"irmae (l=2)"),(model8,"irmae (l=4)")],np.rour
```

(original dataset) (smaller squares)

Figure 1: *Synthetic Shape Dataset Code With Difference Between Two Implementations*

Code

Project:

#!/usr/bin/env python

coding: utf-8

```
# In[2]:


#Dan Brody

import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms
import torch
import argparse
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from torch.utils.data import DataLoader, Dataset
import cv2

args = {
    'latent_dim':32,
    'lr':0.0001,
    'epochs':100,
    'batch_size' : 32,
    'train_len' : 50000,
    'eval_len' : 10000
}
to_pil_image= transforms.ToPILImage()


# In[3]:
```

```
def rectangleOrCircle():
    if(np.random.uniform(0,1)) >= 0.5:
        return 'rectangle'
    else:
        return 'circle'



def randParamsNumpy():
    x = np.random.randint(low = 8,high = 25)
    y = np.random.randint(low = 8,high = 25)
    size = np.random.randint(low = 3,high = 9)
    return x,y,size


def pickColor(only_rand_blue = False):
    if(only_rand_blue == True):
        red = 0
        green = 0
        blue = np.random.uniform(0,1)
    else:
        red = np.random.uniform(0,1)
        green = np.random.uniform(0,1)
        blue = np.random.uniform(0,1)
    return red,green,blue


def randParamsPytorch():
    x = torch.randint(low = 8,high = 25,size = (1,1))[0][0]
```

```
    y = torch.randint(low = 8,high = 25,size = (1,1))[0][0]

    size = torch.randint(low = 3,high = 9,size = (1,1))[0][0]

    return x,y,size



def createSquare(only_rand_blue = False):

    x,y,size = randParamsPytorch()

    red,green,blue = pickColor(only_rand_blue)

    z = torch.zeros(3,32,32) #or torch.ones for white background

    z[0][:,x-size:x+size][y-size:y+size] = red

    z[1][:,x-size:x+size][y-size:y+size] = green

    z[2][:,x-size:x+size][y-size:y+size] = blue

    return z

#,[x,y,size]


def createCircle(only_rand_blue = False):


    x,y,size = randParamsPytorch()

    red,green,blue = pickColor(only_rand_blue)

    z = torch.zeros(3,32,32) #or torch.ones for white background


    X = np.random.multivariate_normal([0, 0], [[1, 0], [0,1]], 10000)

    Z = X / 10 + X / np.sqrt(np.square(X).sum(axis=1, keepdims=True))

    #plt.plot(np.floor(size*np.array(Z[:,0])),np.floor(size*np.array(Z[:,1])),'x')

    new_dict = {}

    grouped_x = 
pd.DataFrame({'x':np.floor(size*np.array(Z[:,0])),'y':np.floor(size*np.array(Z[:,1]))}).groupby(by='x')

    for i in grouped_x:

        val_x1 = min([j if j>0 else 20 for j in i[1]['y']] )
```

```
        val_x2 = max([j if j<0 else -20 for j in i[1]['y']])
        val_x1 = val_x1+size+(x-size)
        val_x2= val_x2+size+(x-size)

        new_dict[(int(i[0]+size+(y-size)))] = [(int(val_x2)),(int(val_x1))]
#ask if size is radius or total size
    #print(f'new_dct is {new_dict}')
    for i in range(2*size):
        y_axis = int(i+(y-size))
        bounds = new_dict[y_axis]

        #print(f'the bounds are{bounds}, size is {size} and x and y are {x},{y}')
        z[0][:,bounds[0]:bounds[1]][32-y_axis-1] = red
        z[1][:,bounds[0]:bounds[1]][32-y_axis-1] = green
        z[2][:,bounds[0]:bounds[1]][32-y_axis-1] = blue

    return z



# In[4]:



import math
from skimage.draw import circle
to_pil_image= transforms.ToPILImage()

def createCircleNewTry(only_rand_blue=False):
    x,y,size = randParamsPytorch()
    red,green,blue = pickColor(only_rand_blue)
```

```python
    img = torch.zeros((3,32,32))
    rr,cc = circle(r = x,c = y,radius = size)
    img[0][rr, cc] = red
    img[1][rr, cc] = green
    img[2][rr, cc] = blue
    return img


# In[5]:


def createCircleNewestTry(only_rand_blue = False):
    to_pil_image= transforms.ToPILImage()
    x,y,size = randParamsNumpy()
    red,green,blue = pickColor(only_rand_blue)
    z = np.zeros((32,32,3))
    cv2.circle(z,(x,y),size,(red,green,blue),-1)
    z = np.transpose(z,[2,0,1])
    z = torch.FloatTensor(z)
    return z
#,[x,y,size]

def createSquareNewestTry(only_rand_blue = False):
    to_pil_image= transforms.ToPILImage()
    x,y,size = randParamsNumpy()
    red,green,blue = pickColor(only_rand_blue)
    z = np.zeros((32,32,3))
    cv2.rectangle(z,(x,y),(x+size,y+size),(red,green,blue),-1)
    z = np.transpose(z,[2,0,1])
```

```
    z = torch.FloatTensor(z)

    return z
#,[x,y,size]


# In[6]:


to_pil_image(createSquareNewestTry())


# In[7]:


to_pil_image(createCircleNewTry())


# In[8]:



to_pil_image(createCircle())


# In[9]:


to_pil_image(createSquare() + createCircle())
```

```
# In[11]:


#creating synthetic dataset
#compress maybe?
def train_set_twoShapes(size = args['train_len']):
    dataset = []
    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        #if(np.random.randint(0,2) == 0):
        #    dataset.append(createCircle())
        #else:
        dataset.append(createSquareNewestTry()+createCircleNewestTry())

    return dataset


# In[12]:


def eval_set_twoShapes(size = args['eval_len']):
    dataset = []
    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        #if(np.random.randint(0,2) == 0):
```

```python
        #    dataset.append(createCircle())
        #else:
        dataset.append(createSquareNewestTry()+createCircleNewestTry())
    return dataset


# In[37]:


def train_set(size = args['train_len']):
    dataset = []

    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
        else:
            data = createSquareNewestTry()
            dataset.append(data)

    return dataset


# In[38]:
```

```
def eval_set(size = args['eval_len']):
    dataset = []

    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
        else:
            data = createSquareNewestTry()
            dataset.append(data)
    return dataset


# In[39]:


train_data = train_set()


# In[40]:


validation_data = eval_set()


# In[41]:
```

```
train_tensor_data = (torch.tensor(np.reshape((np.concatenate(train_data)),[50000,3,32,32])))
to_pil_image(train_tensor_data[90])


# In[42]:


eval_tensor_data = (torch.tensor(np.reshape((np.concatenate(validation_data)),[10000,3,32,32])))
to_pil_image(eval_tensor_data[109])


# In[ ]:


def gaussianNoise(train_tensor = train_tensor, mean = 0,var = 0.000009 ,size = args['train_len']):
    train_tensor_gaussian_noise = []
    #best so far has been var = 0.000009
    for i in range(size):
        if(i%500==0):
            print(f'this is the {i}th iteration')
        image = train_tensor[i]
        row,col,ch= image.shape
        gaussian_mean = float(mean)
        gaussian_var = float(var)
        sigma = float(gaussian_var**0.5)
        gauss = torch.normal(gaussian_mean,sigma,(row,col,ch))
        gauss = torch.reshape(gauss,(row,col,ch))
        noisy = image + gauss
```

```
        train_tensor_gaussian_noise.append(noisy)
    return train_tensor_gaussian_noise


# In[ ]:


def normalize(size = args['train_len'],train_tensor = train_tensor):
    normalized_tensor = []
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
    for i in range(size):
        if(i%500==0):
            print(f'this is the {i}th iteration')
        normalized_tensor.append(normalize(train_tensor[i]))
    return normalized_tensor




# In[ ]:


#probability must be 1/something
def randomizedGaussianNoise(train_tensor = train_tensor, mean = 0,var = 0.000009 ,size = 
args['train_len'], probability = 0.5):
    train_tensor_gaussian_noise = []
    #best so far has been var = 0.000009
    for i in range(size):
        if(i%500==0):
            print(f'this is the {i}th iteration')
```

```
        image = train_tensor[i]
        if(np.random.randint(0,probability**-1) == 0): #high is exclusive
            row,col,ch= image.shape
            gaussian_mean = float(mean)
            gaussian_var = float(var)
            sigma = float(gaussian_var**0.5)
            gauss = torch.normal(gaussian_mean,sigma,(row,col,ch))
            gauss = torch.reshape(gauss,(row,col,ch))
            noisy = image + gauss
            train_tensor_gaussian_noise.append(noisy)
        else:
            train_tensor_gaussian_noise.append(image)

    return train_tensor_gaussian_noise



# In[ ]:



def randomizedNormalization(train_tensor = train_tensor,size = args['train_len'], probability = 0.5):
    normalized_tensor = []
    #best so far has been var = 0.000009
    normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229, 0.224, 0.225])
    for i in range(size):
        if(i%500==0):
            print(f'this is the {i}th iteration')
        image = train_tensor[i]
        if(np.random.randint(0,probability**-1) == 0):
            normalized_tensor.append(normalize(image))
```

```
        else:
            normalized_tensor.append(image)

    return normalized_tensor



# In[21]:



class encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 3, out_channels = 32, kernel_size = 4, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(in_channels = 32, out_channels = 64,kernel_size = 4, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 4, stride = 2, padding = 
1)
        self.conv4 = nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 4, stride = 2, padding = 
1)
        self.conv5 = nn.Conv2d(in_channels = 256, out_channels = 32, kernel_size = 4, stride = 2, padding = 
1)
    def forward(self,x):
        #print(x.shape)
        x = F.relu(self.conv1(x))
        #print(x.shape)
        x = F.relu(self.conv2(x))
        #print(x.shape)
        x = F.relu(self.conv3(x))
        #print(x.shape)
        x = F.relu(self.conv4(x))
        #print(x.shape)
```

```
        x = F.relu(self.conv5(x))
        #print(x.shape)
        x = x.view(x.size(0),-1)

        return x
```

In[22]:

```
class linear_between(nn.Module):
    def __init__ (self, linear_layers):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(32,32) for i in range(linear_layers)])
    def forward(self,x):
         #maybe replace with (x.view(x.size(0),-1))
        for layer in self.layers:
            x = layer(x)
        #print(x.shape)
        #print(x.shape)
        return x
```

In[23]:

```
class decoder(nn.Module):
    def __init__ (self):
        super().__init__()
```

```
        self.convt1 = nn.ConvTranspose2d(in_channels = 32, out_channels = 256,kernel_size = 4, stride = 2, padding = 1)
        self.convt2 = nn.ConvTranspose2d(in_channels = 256, out_channels = 128, kernel_size = 4, stride = 2, padding = 1)
        self.convt3 = nn.ConvTranspose2d(in_channels = 128, out_channels = 64, kernel_size = 4, stride = 2, padding =1)
        self.convt4 = nn.ConvTranspose2d(in_channels = 64, out_channels = 32, kernel_size = 4, stride = 2, padding = 1)
        self.convt5 = nn.ConvTranspose2d(in_channels = 32, out_channels = 3, kernel_size = 4, stride = 2, padding = 1)
    def forward(self,x):
        #print(x.shape)
        x = x.view(-1,32,1,1)
        x = F.relu(self.convt1(x))
        #print(x.shape)
        x = F.relu(self.convt2(x))
        #print(x.shape)
        x = F.relu(self.convt3(x))
        #print(x.shape)
        x = F.relu(self.convt4(x))
        #print(x.shape)
        x = torch.tanh(self.convt5(x))
        #print(x.shape)

        return x

class IMRAE(nn.Module):
    def __init__(self,linear_layers):
        super().__init__()
```

```
        self.linear_layers = linear_layers
        self.encoder = encoder()
        self.linear_between = linear_between(linear_layers)
        self.decoder = decoder()

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




test_dataloader = DataLoader(eval_tensor_data, batch_size = args['batch_size'], shuffle = True)
train_dataloader = DataLoader(train_tensor_data, batch_size = args['batch_size'], shuffle = True)




#maybe decrease the size to ensure square?
def train(lr,train_dataloader,num_epochs,regularization = None, l = 0,lmbda = 1e-10):

    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

    imrae2 = IMRAE(l)
    imrae2.to(device)
    to_pil_image= transforms.ToPILImage()
```

```
optimizer = torch.optim.Adam(params=imrae2.parameters(), lr=lr)

num_epochs = num_epochs

x=[]



for epoch in range(num_epochs):

    train_loss_avg = 0

    num_batches = 0



    for batch in train_dataloader:

        optimizer.zero_grad()

        l1_regularization = 0

        l2_regularization = 0

        batch = batch.to(device)

        reconstructed = imrae2(batch)

        loss = F.mse_loss(reconstructed, batch)


        if(regularization=='l1'):

            l1_regularization = torch.norm(ae_real_trained_l1.encoder(batch),1)

        loss+= lmbda*l1_regularization

        if(regularization == 'l2'):

            l2_regularization = torch.norm(ae_real_trained_l1.encoder(batch),2)**2

        loss+= lmbda*l2_regularization


        loss.backward()

        optimizer.step()

        train_loss_avg+=(loss.item())
```

```python
            num_batches += 1
            x.append(to_pil_image(reconstructed[0].detach().cpu().clone()))

        train_loss_avg /= num_batches
        print(f'Epoch [{epoch+1} / {num_epochs}] average reconstruction error: {train_loss_avg}')

    return imrae2,x,train_loss_avg



# In[26]:


def train_trained_model(imrae2,lr,train_dataloader,num_epochs,regularization = None,lmbda = 1e-10):

    to_pil_image= transforms.ToPILImage()

    optimizer = torch.optim.Adam(params=imrae2.parameters(), lr=lr)
    num_epochs = num_epochs
    x=[]


    for epoch in range(num_epochs):
        train_loss_avg = 0
        num_batches = 0

        for batch in train_dataloader:
            l1_regularization = torch.FloatTensor(0)
            l2_regularization = torch.FloatTensor(0)
```

```
            optimizer.zero_grad()
            batch = batch.to(device)
            reconstructed = imrae2(batch)
            loss = F.mse_loss(reconstructed, batch)
            if(regularization=='l1'):
              l1_regularization = torch.norm(ae_real_trained_l1.encoder(batch),1)
            loss+= lmbda*l1_regularization
            if(regularization == 'l2'):
              l2_regularization = torch.norm(ae_real_trained_l1.encoder(batch),2)**2
            loss+= lmbda*l2_regularization

            print(loss.item())
            loss.backward()
            optimizer.step()
            train_loss_avg+=(loss.item())
            num_batches += 1
            x.append(to_pil_image(reconstructed[0].detach().cpu().clone()))

        train_loss_avg /= num_batches
        print(f'Epoch [{epoch+1} / {num_epochs}] average reconstruction error: {train_loss_avg}')

    return imrae2,x,train_loss_avg


# In[27]:


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
ae_real_trained_l2 = IMRAE(0)
```

```
ae_real_trained_l2.to(device)


# In[48]:


imrae_4 = IMRAE(4)
imrae_4.to(device)


# In[29]:


ae_real_trained_l1 = IMRAE(0)
ae_real_trained_l1.to(device)


# In[49]:


#alternative if there is malfunction in train

#regularization = None
#lmbda = 1e-10
optimizer = torch.optim.Adam(params=imrae_4.parameters(), lr=0.0001)
num_epochs = 100
x=[]

for epoch in range(num_epochs):
```

```
train_loss_avg = 0
num_batches = 0


for batch in train_dataloader:
    optimizer.zero_grad()
    l1_regularization = 0
    l2_regularization = 0
    batch = batch.to(device)
    reconstructed = imrae_4(batch)
    loss = F.mse_loss(reconstructed, batch)

    #if(regularization=='l1'):
    #    l1_regularization = torch.norm(imrae_4.encoder(batch),1)
    #loss+= lmbda*l1_regularization
    #if(regularization == 'l2'):
    #    l2_regularization = torch.norm(imrae_4.encoder(batch),2)**2
    #loss+= lmbda*l2_regularization

    loss.backward()
    optimizer.step()
    train_loss_avg+=(loss.item())
    num_batches += 1
    x.append(to_pil_image(reconstructed[0].detach().cpu().clone()))

train_loss_avg /= num_batches
print(f'Epoch [{epoch+1} / {num_epochs}] average reconstruction error: {train_loss_avg}')
```

```
# In[58]:


#torch.save(ae_real_trained_l2.state_dict(),"ae_real_trained_l2_cvcircles_smallrects_encoderReg_1e-10.pt")
#torch.save(ae_real_trained_l1.state_dict(),"ae_real_trained_l1_cvcircles_smallrects_encoderReg_1e-10_real.pt")
#torch.save(imrae_4.state_dict(), "imrae_4_cvCircles_cvRectangles_lastOne_real.pt")


# In[61]:


x[-5]


# In[ ]:


ae_real_trained_l2, image_array_ae_real_l2, train_loss_avg_ae_real_l2 = train(lr=0.0001 ,train_dataloader = train_twoShape_dataloader ,num_epochs = 5,regularization = "l2")


# In[ ]:


ae_real_trained, image_array_ae_real, train_loss_avg_ae_real = train(lr=0.0001 ,train_dataloader = train_dataloader ,num_epochs = 100)
```

In[ ]:

```
image_array_ae_real[-5]
```

In[ ]:

```
#torch.save(ae_real_trained.state_dict(),"ae_real_trained_colorsbtwen0and1_cv2circles.pt")
```

In[ ]:

```
#torch.save(ae_trained.state_dict(),"ae_trained_twoShapes_100Epochs.pt")
```

In[ ]:

```
imrae_2_trained_real, image_array_imrae_2_real, train_loss_avg_imrae_2_trained_real = train(lr =
0.0001, train_dataloader = train_dataloader, num_epochs = 100, l = 2)
```

In[ ]:

```
imrae_4_trained, image_array_imrae_4, train_loss_avg_imrae_4_trained = train(lr = 0.0001,
train_dataloader = train_dataloader, num_epochs = 10, l = 4)


# In[ ]:


#torch.save(imrae_4_trained.state_dict(),"imrae_4_trained_cvCircles.pt")


# In[33]:


def singular_values(irmae, test_dataloader,layers = 0):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    irmae.eval()
    z = []
    for batch in test_dataloader:

        num_matrices = 0

        with torch.no_grad():
            batch = batch.to(device)
            latent_vec = irmae.encoder(batch)
            if(layers > 0):
                z_portion = irmae.linear_between(latent_vec)
            else:
                z_portion = latent_vec
```

```
        z.append(z_portion)

    z = torch.cat(z,axis = 0).cpu().numpy()
    latent_covariance = np.cov(z,rowvar = False)




    _,diag,_ = np.linalg.svd(latent_covariance)
    return (diag/max(diag)),latent_covariance



# In[ ]:



#using dataset with cv2 curcles but not cv2 rects
model = IMRAE(0)
model = IMRAE(0)
model.load_state_dict(torch.load("ae_real_trained_colorsbtwen0and1_cv2circles.pt"))
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model.to(device)
model2 = IMRAE(2)
model2.load_state_dict(torch.load("imrae_2_CVCircles.pt"))
model2.to(device)
model8 = IMRAE(4)
model8.load_state_dict(torch.load("imrae_4_trained_cvCircles.pt"))
model8.to(device)
model9 = IMRAE(0)
model9.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_encoderReg_1e-10_real.pt"))
model9.to(device)
```

```
model10 = IMRAE(0)

model10.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_encoderReg_1e-10.pt"))

model10.to(device)

diag_ae,_ = singular_values(model,test_dataloader)

diag_2,_ = singular_values(model2,test_dataloader)

diag_4,_ = singular_values(model8, test_dataloader)

diag_ae_l1_trained,_ = singular_values(model9, test_dataloader)

diag_ae_l2_trained,_ = singular_values(model10,test_dataloader)

# In[76]:




#using dataset with cv2 circles abd cv2 squares (smaller)

#models are "ae_real_trained_CV2circles_smallerSquares.pt" and
"imrae_2_trained_real_CV2circles_smallerSquares.pt"

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model3 = IMRAE(0)

model3.load_state_dict(torch.load("ae_real_trained_CV2circles_smallerSquares.pt"))

model3.to(device)

model4 = IMRAE(2)

model4.load_state_dict(torch.load("imrae_2_trained_real_CV2circles_smallerSquares.pt"))

model4.to(device)

model5 = IMRAE(0)

model5.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_smallrects.pt"))

model5.to(device)

model6 = IMRAE(0)

model6.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_smallrects_encoderReg_1e-10.pt"))

model6.to(device)

model7 = IMRAE(0)
```

```
model7.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_smallrects_encoderReg_1e-10_real.pt"))

model7.to(device)

diag_ae_real_trained, latent_cov_ae_real_train = singular_values(model3, test_dataloader)

diag_2_real_trained, latent_cov_2_real_trained = singular_values(model4, test_dataloader)

#ae_real_trained_l1_diag, cov_l1 = singular_values(model5, test_dataloader)

ae_real_trained_l2_diag,cov_l2 = singular_values(model6, test_dataloader)

#test_diag,_ = singular_values(ae_real_trained_l2,test_dataloader)

ae_real_trained_l1_diag, cov_l1 = singular_values(model7,test_dataloader)

imrae_4_diag, cov_4 = singular_values(imrae_4, test_dataloader)



# In[77]:




plt.plot(diag_ae_real_trained,label = f'ae with matrix rank: {torch.matrix_rank(torch.tensor(latent_cov_ae_real_train))}')

plt.plot(diag_2_real_trained, label = f"imrae (l=2) (cv2 rects: {torch.matrix_rank(torch.tensor(latent_cov_2_real_trained))}")

plt.plot(ae_real_trained_l2_diag, label = f"ae (l2 reg)  cv2 rects: {torch.matrix_rank(torch.tensor(cov_l2))}")

plt.plot(ae_real_trained_l1_diag, label = f"ae (l1 reg) cv2 rects: {torch.matrix_rank(torch.tensor(cov_l1))}")

plt.plot(imrae_4_diag, label = f"imrae (l=4) cv2 rects: {torch.matrix_rank(torch.tensor(cov_4))}")



plt.vlines(7,-1,1,linestyles = "dashed")

plt.ylim(0,0.3)

plt.ylabel('singular values')
```

```
plt.xlabel('singular value rank')
plt.title("cv2 squares, cv2 rects")
plt.legend()


# In[ ]:



import torchvision.utils


def plt_images(image):
    to_pil_image= transforms.ToPILImage()
    plt.imshow(to_pil_image(image))

images,labels = iter(test_dataloader).next()

plt_images(torchvision.utils.make_grid(images[1:31],10,3))
plt.show()


# In[71]:



import cv2 as cv
def interpolate(models,x):
    to_pil_image= transforms.ToPILImage()
    fig,axs = plt.subplots(len(models),len(x), figsize= (20,12))
```

```
    index = np.random.randint(30)

    images = iter(test_dataloader).next()

    images = images.to(device)

    row = 0

    for (model,name) in models:

        model.eval()

        z = model.linear_between(model.encoder(images))

        z1 = z[index]

        z2 = z[index+1]

        for b,i in enumerate(x):

            interpolated_image = i*z1 + (1-i) * z2

            ans = torch.reshape(model.decoder(interpolated_image),(3,32,32))

            ans_np = np.transpose(ans.cpu().detach().numpy(), [2,1,0])

            axs[row, b].imshow(ans_np)

            #axs[row,
b].imshow(to_pil_image(torch.reshape(model.decoder(interpolated_image),(3,32,32)).cpu()))

            axs[row, b].set_title(f'{name}, x:{np.round(i,decimals = 1)}',fontdict = {'fontsize':8})

        row+=1
```

```
# In[73]:

model = IMRAE(0)

model.load_state_dict(torch.load("ae_real_trained_colorsbtwen0and1_cv2circles.pt"))

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model.to(device)

model2 = IMRAE(2)

model2.load_state_dict(torch.load("imrae_2_CVCircles.pt"))
```

```
model2.to(device)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

model8 = IMRAE(4)

model8.load_state_dict(torch.load("imrae_4_trained_cvCircles.pt"))

model8.to(device)

model9 = IMRAE(0)

model9.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_encoderReg_1e-10_real.pt"))

model9.to(device)

model10 = IMRAE(0)

model10.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_encoderReg_1e-10.pt"))

model10.to(device)




model3 = IMRAE(0)

model3.load_state_dict(torch.load("ae_real_trained_CV2circles_smallerSquares.pt"))

model3.to(device)

model4 = IMRAE(2)

model4.load_state_dict(torch.load("imrae_2_trained_real_CV2circles_smallerSquares.pt"))

model4.to(device)

model6 = IMRAE(0)

model6.load_state_dict(torch.load("ae_real_trained_l2_cvcircles_smallrects_encoderReg_1e-10.pt"))

model6.to(device)

model7 = IMRAE(0)

model7.load_state_dict(torch.load("ae_real_trained_l1_cvcircles_smallrects_encoderReg_1e-
10_real.pt"))

model7.to(device)

model8 = IMRAE(4)

model8.load_state_dict(torch.load("imrae_4_cvCircles_cvRectangles_lastOne_real.pt"))
```

```
model8.to(device)

interpolate([(model3,"ae"),(model7,"ae(reg = l1)"),(model6,"ae(reg = l2)"),(model4,"irmae (l=2)"),(model8,"irmae (l=4)")],np.round(np.linspace(0,1,10),decimals = 1))
```

Experimentation:

```
#!/usr/bin/env python

# coding: utf-8


# In[1]:




import torch.nn as nn

import torch.nn.functional as F

from torchvision import transforms

import torch

import argparse

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from torch.utils.data import DataLoader, Dataset

import cv2

import pylab as plt




# In[2]:




def rectangleOrCircle():
```

```
    if(np.random.uniform(0,1)) >= 0.5:
        return 'rectangle'
    else:
        return 'circle'


def randParamsNumpy():
    x = np.random.randint(low = 8,high = 25)
    y = np.random.randint(low = 8,high = 25)
    size = np.random.randint(low = 3,high = 9)
    return x,y,size

def pickColor(only_rand_blue = False):
    if(only_rand_blue == True):
        red = 0
        green = 0
        blue = np.random.uniform(0,1)
    else:
        red = np.random.uniform(0,1)
        green = np.random.uniform(0,1)
        blue = np.random.uniform(0,1)
    return red,green,blue

def randParamsPytorch():
    x = torch.randint(low = 8,high = 25,size = (1,1))[0][0]
    y = torch.randint(low = 8,high = 25,size = (1,1))[0][0]
    size = torch.randint(low = 3,high = 9,size = (1,1))[0][0]
    return x,y,size
```

```
def createSquare(only_rand_blue = False):
    x,y,size = randParamsPytorch()
    red,green,blue = pickColor(only_rand_blue)
    z = torch.zeros(3,32,32) #or torch.ones for white background
    z[0][:,x-size:x+size][y-size:y+size] = red
    z[1][:,x-size:x+size][y-size:y+size] = green
    z[2][:,x-size:x+size][y-size:y+size] = blue
    return z
#,[x,y,size]


def createCircle(only_rand_blue = False):


    x,y,size = randParamsPytorch()
    red,green,blue = pickColor(only_rand_blue)
    z = torch.zeros(3,32,32) #or torch.ones for white background


    X = np.random.multivariate_normal([0, 0], [[1, 0], [0,1]], 10000)
    Z = X / 10 + X / np.sqrt(np.square(X).sum(axis=1, keepdims=True))
    #plt.plot(np.floor(size*np.array(Z[:,0])),np.floor(size*np.array(Z[:,1])),'x')
    new_dict = {}
    grouped_x =
pd.DataFrame({'x':np.floor(size*np.array(Z[:,0])),'y':np.floor(size*np.array(Z[:,1]))}).groupby(by='x')
    for i in grouped_x:
        val_x1 = min([j if j>0 else 20 for j in i[1]['y']] )
        val_x2 = max([j if j<0 else -20 for j in i[1]['y']])
        val_x1 = val_x1+size+(x-size)
        val_x2= val_x2+size+(x-size)
```

```
        new_dict[(int(i[0]+size+(y-size)))] = [(int(val_x2)),(int(val_x1))]
#ask if size is radius or total size
    #print(f'new_dct is {new_dict}')
    for i in range(2*size):
        y_axis = int(i+(y-size))
        bounds = new_dict[y_axis]

        #print(f'the bounds are{bounds}, size is {size} and x and y are {x},{y}')
        z[0][:,bounds[0]:bounds[1]][32-y_axis-1] = red
        z[1][:,bounds[0]:bounds[1]][32-y_axis-1] = green
        z[2][:,bounds[0]:bounds[1]][32-y_axis-1] = blue

    return z


# In[34]:


import cv2 as cv
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
#maybe try standardscaler next time?
def createCircleNewestTry(only_rand_blue = False):
    to_pil_image= transforms.ToPILImage()
    x,y,size = randParamsNumpy()
    red,green,blue = pickColor(only_rand_blue)
    z = np.zeros((32,32,3))
    cv2.circle(z,(x,y),size,(red,green,blue),-1)
    z = np.transpose(z,[2,0,1])
```

```
    z = torch.FloatTensor(z)

#for j in range(32):
    #   for k in range(32):
     #     if(z[0][j][k] != 0 or z[1][j][k] != 0 or  z[2][j][k] != 0):
      #        if(np.random.randint(2) == 0):
       #           z[0][j][k] = z[0][j][k]/-1
        #          z[1][j][k] = z[1][j][k]/-1
         #         z[2][j][k] = z[2][j][k]/-1

return(torch.FloatTensor(np.reshape(scaler.fit_transform(torch.reshape(z,(3,32*32)).numpy()),(3,32,32)
)))

#,[x,y,size]

def createSquareNewestTry(only_rand_blue = False):

    x,y,size = randParamsNumpy()
    red,green,blue = pickColor(only_rand_blue)
    z = np.zeros((32,32,3))
    cv2.rectangle(z,(x,y),(x+size,y+size),(red,green,blue),-1)
    z = np.transpose(z,[2,0,1])
    z = torch.FloatTensor(z)

    #for j in range(32):
    #   for k in range(32):
     #     if(z[0][j][k] != 0 or z[1][j][k] != 0 or  z[2][j][k] != 0):
      #        if(np.random.randint(2) == 0):
```

```
#           z[0][j][k] = z[0][j][k]/-1
 #          z[1][j][k] = z[1][j][k]/-1
  #         z[2][j][k] = z[2][j][k]/-1

return(torch.FloatTensor(np.reshape(scaler.fit_transform(torch.reshape(z,(3,32*32)).numpy()),(3,32,32)
)))




# In[35]:




def train_set(size = 50000):
    dataset = []


    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
            #labels.append(label)
        else:
            data = createSquareNewestTry()
            dataset.append(data)
            #labels.append(label)
    return dataset
```

```
# In[83]:


to_pil_image= transforms.ToPILImage()
x = createSquareNewestTry()
to_pil_image(x)


# In[84]:


torch.mean(x)


# In[85]:


torch.min(x)


# In[37]:


train = train_set()


# In[38]:
```

```
def eval_set(size = 10000):
    dataset = []
    for i in range(size):
        if(i%500==0):
            print(f' this is the {i}th iteration')
        if(np.random.randint(0,2) == 0):
            data = createCircleNewestTry()
            dataset.append(data)
            #labels.append(label)
        else:
            data = createSquareNewestTry()
            dataset.append(data)
            #labels.append(label)
    return dataset


# In[39]:


validation = eval_set()


# In[40]:


train_tensor_data = (torch.tensor(np.reshape((np.concatenate(train)),[50000,3,32,32])))
eval_tensor_data = (torch.tensor(np.reshape((np.concatenate(validation)),[10000,3,32,32])))
```

```
# In[41]:


from sklearn.decomposition import PCA
pca = PCA()
new_train_tensor =
torch.FloatTensor(np.reshape(pca.fit_transform(torch.reshape(train_tensor_data,(50000,3*32*32)).numpy()),(50000,3,32,32)))


# In[44]:


#not possible to decorrelate
to_pil_image= transforms.ToPILImage()
to_pil_image(new_train_tensor[10])


# In[212]:


to_pil_image= transforms.ToPILImage()
to_pil_image(z)


# In[199]:


z.numpy()
```

In[ ]:

```
class encoder(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 3, out_channels = 32, kernel_size = 4, stride = 2, padding = 1)
        self.conv2 = nn.Conv2d(in_channels = 32, out_channels = 64,kernel_size = 4, stride = 2, padding = 1)
        self.conv3 = nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size = 4, stride = 2, padding = 1)
        self.conv4 = nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size = 4, stride = 2, padding = 1)
        self.conv5 = nn.Conv2d(in_channels = 256, out_channels = 32, kernel_size = 4, stride = 2, padding = 1)
    def forward(self,x):
        #print(x.shape)
        x = F.relu(self.conv1(x))
        #print(x.shape)
        x = F.relu(self.conv2(x))
        #print(x.shape)
        x = F.relu(self.conv3(x))
        #print(x.shape)
        x = F.relu(self.conv4(x))
        #print(x.shape)
        x = F.relu(self.conv5(x))
        #print(x.shape)
        x = x.view(x.size(0),-1)
```

```
        return x


# In[ ]:


class linear_between(nn.Module):
    def __init__ (self, linear_layers):
        super().__init__()
        self.layers = nn.ModuleList([nn.Linear(32,32) for i in range(linear_layers)])
    def forward(self,x):
         #maybe replace with (x.view(x.size(0),-1))
        for layer in self.layers:
            x = layer(x)
        #print(x.shape)
        #print(x.shape)
        return x


# In[ ]:


class decoder(nn.Module):
    def __init__ (self):
        super().__init__()

        self.convt1 = nn.ConvTranspose2d(in_channels = 32, out_channels = 256,kernel_size = 4, stride = 2,
padding = 1)
```

```
        self.convt2 = nn.ConvTranspose2d(in_channels = 256, out_channels = 128, kernel_size = 4, stride = 2, padding = 1)

        self.convt3 = nn.ConvTranspose2d(in_channels = 128, out_channels = 64, kernel_size = 4, stride = 2, padding =1)

        self.convt4 = nn.ConvTranspose2d(in_channels = 64, out_channels = 32, kernel_size = 4, stride = 2, padding = 1)

        self.convt5 = nn.ConvTranspose2d(in_channels = 32, out_channels = 3, kernel_size = 4, stride = 2, padding = 1)

    def forward(self,x):

        #print(x.shape)

        x = x.view(-1,32,1,1)

        x = F.relu(self.convt1(x))

        #print(x.shape)

        x = F.relu(self.convt2(x))

        #print(x.shape)

        x = F.relu(self.convt3(x))

        #print(x.shape)

        x = F.relu(self.convt4(x))

        #print(x.shape)

        x = torch.tanh(self.convt5(x))

        #print(x.shape)

        #change to sigmoid for experimentation

        return x


class IMRAE(nn.Module):

    def __init__(self,linear_layers):

        super().__init__()

        self.linear_layers = linear_layers

        self.encoder = encoder()

        self.linear_between = linear_between(linear_layers)
```

```
        self.decoder = decoder()

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


# In[ ]:


test_dataloader = DataLoader(eval_tensor_data, batch_size = args['batch_size'], shuffle = True)
train_dataloader = DataLoader(train_tensor_data, batch_size = args['batch_size'], shuffle = True)


# In[82]:


device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
imrae_4 = IMRAE(4)
imrae_4.to(device)


# In[ ]:


#alternative if there is malfunction in train
#do for imrae(l=2) and ae(baseline) the zero mean thing and the sigmoid thing
```

```
regularization = None
lmbda = 1e-10
optimizer = torch.optim.Adam(params=imrae_4.parameters(), lr=0.0001)
num_epochs = 20
x=[]

for epoch in range(num_epochs):
    train_loss_avg = 0
    num_batches = 0



    for batch in train_dataloader:
        optimizer.zero_grad()
        l1_regularization = 0
        l2_regularization = 0
        batch = batch.to(device)
        reconstructed = imrae_4(batch)
        loss = F.mse_loss(reconstructed, batch)

        if(regularization=='l1'):
            l1_regularization = torch.norm(imrae_4.encoder(batch),1)
        loss+= lmbda*l1_regularization
        if(regularization == 'l2'):
            l2_regularization = torch.norm(imrae_4.encoder(batch),2)**2
        loss+= lmbda*l2_regularization

        loss.backward()
        optimizer.step()
```

```python
    train_loss_avg+=(loss.item())
    num_batches += 1
    x.append(to_pil_image(reconstructed[0].detach().cpu().clone()))

train_loss_avg /= num_batches
print(f'Epoch [{epoch+1} / {num_epochs}] average reconstruction error: {train_loss_avg}')
```