BANGLADESH UNIVERSITY OF BUSINESS AND TECHNOLOGY

BUBT

Committed to Academic Excellence

Theory Assignment

Course Code: CSE 477

Course Title: Neural Network and Fuzzy Systems

<table>
<tr>
<td>

<u>Submitted to:</u>

Name: Mr.T.M. Amir - Ul - Haque Bhuiyan

Assistant Professor

Department of Computer Science & Engineering

at Bangladesh University of Business and Technology.

</td>
<td>

<u>Submitted by:</u>

Name: Syeda Nowshin Ibnat

ID: 17183103020

Intake: 39

Section: 02

Program: B.Sc. in CSE

Semester: Fall 2021-2022

</td>
</tr>
</table>

Date of Submission: 10-04-2022

GAN

Generative Adversarial Networks, or GANs, are an architecture for training generative models, such as deep convolutional neural networks for generating images.

Developing a GAN for generating images requires both a discriminator convolutional neural network model for classifying whether a given image is real or generated and a generator model that uses inverse convolutional layers to transform an input to a full two-dimensional image of pixel values.

Code:

```
# example of loading the mnist dataset

from keras.datasets.mnist import load_data

# load the images into memory

(trainX, trainy), (testX, testy) = load_data()

# summarize the shape of the dataset

print('Train', trainX.shape, trainy.shape)

print('Test', testX.shape, testy.shape)

import matplotlib

import matplotlib.pyplot as plt

# plot raw pixel data

pyplot.imshow(trainX[i], cmap='gray_r')

# example of loading the mnist dataset

from keras.datasets.mnist import load_data

from matplotlib import pyplot

# load the images into memory

(trainX, trainy), (testX, testy) = load_data()
```

```
# plot images from the training dataset

for i in range(25):

# define subplot

pyplot.subplot(5, 5, 1 + i)

# turn off axis

pyplot.axis('off')

# plot raw pixel data

pyplot.imshow(trainX[i], cmap='gray_r')

pyplot.show()

# define the standalone discriminator model

def define_discriminator(in_shape=(28,28,1)):

model = Sequential()

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

```
return model
# example of defining the discriminator model
import keras
import os
from tensorflow import keras
from keras.models import Sequential
#from keras.optimizers import Adam
from tensorflow.keras.optimizers import Adam
from keras.layers import Dense
from keras.layers import Conv2D
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LeakyReLU
from keras.utils.vis_utils import plot_model
# define the standalone discriminator model
def define_discriminator(in_shape=(28,28,1)):
model = Sequential()
model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.4))
model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.4))
```

```
model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

return model

# define model

model = define_discriminator()

# summarize the model

model.summary()

# plot the model

plot_model(model, to_file='discriminator_plot.png', show_shapes=True,
show_layer_names=True)

# load mnist dataset

(trainX, _), (_, _) = load_data()

# expand to 3d, e.g. add channels dimension

def expand_dims(x, axis=-1):

X = expand_dims(trainX, axis=-1)

# convert from unsigned ints to floats

def f(x):

X = X.astype('float32')

# scale from [0,255] to [0,1]

X = X / 255.0
```

```
# load and prepare mnist training images

def load_real_samples():

# load mnist dataset

(trainX, _), (_, _) = load_data()

# expand to 3d, e.g. add channels dimension

X = expand_dims(trainX, axis=-1)

# convert from unsigned ints to floats

X = X.astype('float32')

# scale from [0,255] to [0,1]

X = X / 255.0

return X

# select real samples

def generate_real_samples(dataset, n_samples):

# choose random instances

ix = randint(0, dataset.shape[0], n_samples)

# retrieve selected images

X = dataset[ix]

# generate 'real' class labels (1)

y = ones((n_samples, 1))

return X, y

# generate n fake samples with class labels

def generate_fake_samples(n_samples):

# generate uniform random numbers in [0,1]
```

```
X = rand(28 * 28 * n_samples)

# reshape into a batch of grayscale images

X = X.reshape((n_samples, 28, 28, 1))

# generate 'fake' class labels (0)

y = zeros((n_samples, 1))

return X, y

# train the discriminator model

def train_discriminator(model, dataset, n_iter=100, n_batch=256):

half_batch = int(n_batch / 2)

# manually enumerate epochs

for i in range(n_iter):

# get randomly selected 'real' samples

X_real, y_real = generate_real_samples(dataset, half_batch)

# update discriminator on real samples

_, real_acc = model.train_on_batch(X_real, y_real)

# generate 'fake' examples

X_fake, y_fake = generate_fake_samples(half_batch)

# update discriminator on fake samples

_, fake_acc = model.train_on_batch(X_fake, y_fake)

# summarize performance

print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# example of training the discriminator model on real and random mnist images

from numpy import expand_dims
```

```
from numpy import ones

from numpy import zeros

from numpy.random import rand

from numpy.random import randint

from keras.datasets.mnist import load_data

#from keras.optimizers import Adam

from tensorflow.keras.optimizers import Adam

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Conv2D

from keras.layers import Flatten

from keras.layers import Dropout

from keras.layers import LeakyReLU

# define the standalone discriminator model

def define_discriminator(in_shape=(28,28,1)):

model = Sequential()

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Flatten())
```

```
model.add(Dense(1, activation='sigmoid'))

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

return model

# load and prepare mnist training images

def load_real_samples():

# load mnist dataset

(trainX, _), (_, _) = load_data()

# expand to 3d, e.g. add channels dimension

X = expand_dims(trainX, axis=-1)

# convert from unsigned ints to floats

X = X.astype('float32')

# scale from [0,255] to [0,1]

X = X / 255.0

return X

# select real samples

def generate_real_samples(dataset, n_samples):

# choose random instances

ix = randint(0, dataset.shape[0], n_samples)

# retrieve selected images

X = dataset[ix]

# generate 'real' class labels (1)
```

```
y = ones((n_samples, 1))

return X, y

# generate n fake samples with class labels

def generate_fake_samples(n_samples):

# generate uniform random numbers in [0,1]

X = rand(28 * 28 * n_samples)

# reshape into a batch of grayscale images

X = X.reshape((n_samples, 28, 28, 1))

# generate 'fake' class labels (0)

y = zeros((n_samples, 1))

return X, y

# train the discriminator model

def train_discriminator(model, dataset, n_iter=100, n_batch=256):

half_batch = int(n_batch / 2)

# manually enumerate epochs

for i in range(n_iter):

# get randomly selected 'real' samples

X_real, y_real = generate_real_samples(dataset, half_batch)

# update discriminator on real samples

_, real_acc = model.train_on_batch(X_real, y_real)

# generate 'fake' examples

X_fake, y_fake = generate_fake_samples(half_batch)

# update discriminator on fake samples
```

```
_, fake_acc = model.train_on_batch(X_fake, y_fake)

# summarize performance

print('>%d real=%.0f%% fake=%.0f%%' % (i+1, real_acc*100, fake_acc*100))

# define the discriminator model

model = define_discriminator()

# load image data

dataset = load_real_samples()

# fit the model

train_discriminator(model, dataset)

# foundation for 7x7 image

model.add(Dense(128 * 7 * 7, input_dim=100))

def reshape():

model.add(Reshape((7, 7, 128)))

# upsample to 14x14

def Conv2DTranspose():

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

# define the standalone generator model

def define_generator(latent_dim):

model = Sequential()

# foundation for 7x7 image

n_nodes = 128 * 7 * 7

model.add(Dense(n_nodes, input_dim=latent_dim))

model.add(LeakyReLU(alpha=0.2))
```

```
model.add(Reshape((7, 7, 128)))

# upsample to 14x14

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# upsample to 28x28

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))

return model

# example of defining the generator model

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Reshape

from keras.layers import Conv2D

from keras.layers import Conv2DTranspose

from keras.layers import LeakyReLU

from keras.utils.vis_utils import plot_model

# define the standalone generator model

def define_generator(latent_dim):

model = Sequential()

# foundation for 7x7 image

n_nodes = 128 * 7 * 7

model.add(Dense(n_nodes, input_dim=latent_dim))
```

```
model.add(LeakyReLU(alpha=0.2))

model.add(Reshape((7, 7, 128)))

# upsample to 14x14

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# upsample to 28x28

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))

return model

# define the size of the latent space

latent_dim = 100

# define the generator model

model = define_generator(latent_dim)

# summarize the model

model.summary()

# plot the model

plot_model(model, to_file='generator_plot.png', show_shapes=True, show_layer_names=True)

# generate points in latent space as input for the generator

def generate_latent_points(latent_dim, n_samples):

# generate points in the latent space

x_input = randn(latent_dim * n_samples)

# reshape into a batch of inputs for the network
```

```
x_input = x_input.reshape(n_samples, latent_dim)
return x_input
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
# generate points in latent space
x_input = generate_latent_points(latent_dim, n_samples)
# predict outputs
X = g_model.predict(x_input)
# create 'fake' class labels (0)
y = zeros((n_samples, 1))
return X, y
# example of defining and using the generator model
from numpy import zeros
from numpy.random import randn
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose
from keras.layers import LeakyReLU
from matplotlib import pyplot
# define the standalone generator model
def define_generator(latent_dim):
```

```
model = Sequential()

# foundation for 7x7 image

n_nodes = 128 * 7 * 7

model.add(Dense(n_nodes, input_dim=latent_dim))

model.add(LeakyReLU(alpha=0.2))

model.add(Reshape((7, 7, 128)))

# upsample to 14x14

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

# upsample to 28x28

model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))

return model

# generate points in latent space as input for the generator

def generate_latent_points(latent_dim, n_samples):

# generate points in the latent space

x_input = randn(latent_dim * n_samples)

# reshape into a batch of inputs for the network

x_input = x_input.reshape(n_samples, latent_dim)

return x_input

# use the generator to generate n fake examples, with class labels

def generate_fake_samples(g_model, latent_dim, n_samples):
```

```
# generate points in latent space

x_input = generate_latent_points(latent_dim, n_samples)

# predict outputs

X = g_model.predict(x_input)

# create 'fake' class labels (0)

y = zeros((n_samples, 1))

return X, y

# size of the latent space

latent_dim = 100

# define the discriminator model

model = define_generator(latent_dim)

# generate samples

n_samples = 25

X, _ = generate_fake_samples(model, latent_dim, n_samples)

# plot the generated samples

for i in range(n_samples):

# define subplot

pyplot.subplot(5, 5, 1 + i)

# turn off axis labels

pyplot.axis('off')

# plot single image

pyplot.imshow(X[i, :, :, 0], cmap='gray_r')

# show the figure
```

```
pyplot.show()

# define the combined generator and discriminator model, for updating the generator

def define_gan(g_model, d_model):

# make weights in the discriminator not trainable

d_model.trainable = False

# connect them

model = Sequential()

# add generator

model.add(g_model)

# add the discriminator

model.add(d_model)

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt)

return model

# demonstrate creating the three models in the gan

#from keras.optimizers import Adam

from tensorflow.keras.optimizers import Adam

from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Reshape

from keras.layers import Flatten

from keras.layers import Conv2D
```

```
from keras.layers import Conv2DTranspose

from keras.layers import LeakyReLU

from keras.layers import Dropout

from keras.utils.vis_utils import plot_model

# define the standalone discriminator model

def define_discriminator(in_shape=(28,28,1)):

model = Sequential()

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])

return model

# define the standalone generator model

def define_generator(latent_dim):

model = Sequential()

# foundation for 7x7 image
```

```
n_nodes = 128 * 7 * 7
model.add(Dense(n_nodes, input_dim=latent_dim))
model.add(LeakyReLU(alpha=0.2))
model.add(Reshape((7, 7, 128)))
# upsample to 14x14
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
# upsample to 28x28
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
return model
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
# make weights in the discriminator not trainable
d_model.trainable = False
# connect them
model = Sequential()
# add generator
model.add(g_model)
# add the discriminator
model.add(d_model)
# compile model
```

```
opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt)

return model

# size of the latent space

latent_dim = 100

# create the discriminator

d_model = define_discriminator()

# create the generator

g_model = define_generator(latent_dim)

# create the gan

gan_model = define_gan(g_model, d_model)

# summarize gan model

gan_model.summary()

# plot gan model

plot_model(gan_model, to_file='gan_plot.png', show_shapes=True, show_layer_names=True)

# train the composite model

def train_gan(gan_model, latent_dim, n_epochs=100, n_batch=256):

# manually enumerate epochs

for i in range(n_epochs):

# prepare points in latent space as input for the generator

x_gan = generate_latent_points(latent_dim, n_batch)

# create inverted labels for the fake samples

y_gan = ones((n_batch, 1))
```

```
# update the generator via the discriminator's error
gan_model.train_on_batch(x_gan, y_gan)
# train the generator and discriminator
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):
bat_per_epo = int(dataset.shape[0] / n_batch)
half_batch = int(n_batch / 2)
# manually enumerate epochs
for i in range(n_epochs):
# enumerate batches over the training set
for j in range(bat_per_epo):
# get randomly selected 'real' samples
X_real, y_real = generate_real_samples(dataset, half_batch)
# generate 'fake' examples
X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)
# create training set for the discriminator
X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))
# update discriminator model weights
d_loss, _ = d_model.train_on_batch(X, y)
# prepare points in latent space as input for the generator
X_gan = generate_latent_points(latent_dim, n_batch)
# create inverted labels for the fake samples
y_gan = ones((n_batch, 1))
# update the generator via the discriminator's error
```

```
g_loss = gan_model.train_on_batch(X_gan, y_gan)

# summarize loss on this batch

print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))

# evaluate the discriminator, plot generated images, save generator model

def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):

# prepare real samples

X_real, y_real = generate_real_samples(dataset, n_samples)

# evaluate discriminator on real examples

_, acc_real = d_model.evaluate(X_real, y_real, verbose=0)

# prepare fake examples

x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)

# evaluate discriminator on fake examples

_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)

# summarize discriminator performance

print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))

# train the generator and discriminator

def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):

bat_per_epo = int(dataset.shape[0] / n_batch)

half_batch = int(n_batch / 2)

# manually enumerate epochs

for i in range(n_epochs):

# evaluate the model performance, sometimes

if (i+1) % 10 == 0:
```

```
summarize_performance(i, g_model, d_model, dataset, latent_dim)

# save the generator model tile file

def epoch():

filename = 'generator_model_%03d.h5' % (epoch + 1)

g_model.save(filename)

# create and save a plot of generated images (reversed grayscale)

def save_plot(examples, epoch, n=10):

# plot images

for i in range(n * n):

# define subplot

pyplot.subplot(n, n, 1 + i)

# turn off axis

pyplot.axis('off')

# plot raw pixel data

pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')

# save plot to file

filename = 'generated_plot_e%03d.png' % (epoch+1)

pyplot.savefig(filename)

pyplot.close()

# evaluate the discriminator, plot generated images, save generator model

def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):

# prepare real samples

X_real, y_real = generate_real_samples(dataset, n_samples)
```

```
# evaluate discriminator on real examples

_, acc_real = d_model.evaluate(X_real, y_real, verbose=0)

# prepare fake examples

x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)

# evaluate discriminator on fake examples

_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)

# summarize discriminator performance

print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))

# save plot

save_plot(x_fake, epoch)

# save the generator model tile file

filename = 'generator_model_%03d.h5' % (epoch + 1)

g_model.save(filename)

# Complete Example of GAN for MNIST

# example of training a gan on mnist

from numpy import expand_dims

from numpy import zeros

from numpy import ones

from numpy import vstack

from numpy.random import randn

from numpy.random import randint

from keras.datasets.mnist import load_data

from keras.optimizers import Adam
```

```
from keras.models import Sequential

from keras.layers import Dense

from keras.layers import Reshape

from keras.layers import Flatten

from keras.layers import Conv2D

from keras.layers import Conv2DTranspose

from keras.layers import LeakyReLU

from keras.layers import Dropout

from matplotlib import pyplot

# define the standalone discriminator model

def define_discriminator(in_shape=(28,28,1)):

model = Sequential()

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same', input_shape=in_shape))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))

model.add(LeakyReLU(alpha=0.2))

model.add(Dropout(0.4))

model.add(Flatten())

model.add(Dense(1, activation='sigmoid'))

# compile model

opt = Adam(lr=0.0002, beta_1=0.5)

model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
```

```
return model
# define the standalone generator model
def define_generator(latent_dim):
model = Sequential()
# foundation for 7x7 image
n_nodes = 128 * 7 * 7
model.add(Dense(n_nodes, input_dim=latent_dim))
model.add(LeakyReLU(alpha=0.2))
model.add(Reshape((7, 7, 128)))
# upsample to 14x14
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
# upsample to 28x28
model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
model.add(LeakyReLU(alpha=0.2))
model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
return model
# define the combined generator and discriminator model, for updating the generator
def define_gan(g_model, d_model):
# make weights in the discriminator not trainable
d_model.trainable = False
# connect them
model = Sequential()
```

```
# add generator
model.add(g_model)
# add the discriminator
model.add(d_model)
# compile model
opt = Adam(lr=0.0002, beta_1=0.5)
model.compile(loss='binary_crossentropy', optimizer=opt)
return model
# load and prepare mnist training images
def load_real_samples():
# load mnist dataset
(trainX, _), (_, _) = load_data()
# expand to 3d, e.g. add channels dimension
X = expand_dims(trainX, axis=-1)
# convert from unsigned ints to floats
X = X.astype('float32')
# scale from [0,255] to [0,1]
X = X / 255.0
return X
# select real samples
def generate_real_samples(dataset, n_samples):
# choose random instances
ix = randint(0, dataset.shape[0], n_samples)
```

```
# retrieve selected images
X = dataset[ix]
# generate 'real' class labels (1)
y = ones((n_samples, 1))
return X, y
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
# generate points in the latent space
x_input = randn(latent_dim * n_samples)
# reshape into a batch of inputs for the network
x_input = x_input.reshape(n_samples, latent_dim)
return x_input
# use the generator to generate n fake examples, with class labels
def generate_fake_samples(g_model, latent_dim, n_samples):
# generate points in latent space
x_input = generate_latent_points(latent_dim, n_samples)
# predict outputs
X = g_model.predict(x_input)
# create 'fake' class labels (0)
y = zeros((n_samples, 1))
return X, y
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, epoch, n=10):
```

```
# plot images
for i in range(n * n):
# define subplot
pyplot.subplot(n, n, 1 + i)
# turn off axis
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
# save plot to file
filename = 'generated_plot_e%03d.png' % (epoch+1)
pyplot.savefig(filename)
pyplot.close()
# evaluate the discriminator, plot generated images, save generator model
def summarize_performance(epoch, g_model, d_model, dataset, latent_dim, n_samples=100):
# prepare real samples
X_real, y_real = generate_real_samples(dataset, n_samples)
# evaluate discriminator on real examples
_, acc_real = d_model.evaluate(X_real, y_real, verbose=0)
# prepare fake examples
x_fake, y_fake = generate_fake_samples(g_model, latent_dim, n_samples)
# evaluate discriminator on fake examples
_, acc_fake = d_model.evaluate(x_fake, y_fake, verbose=0)
# summarize discriminator performance
```

```
print('>Accuracy real: %.0f%%, fake: %.0f%%' % (acc_real*100, acc_fake*100))

# save plot

save_plot(x_fake, epoch)

# save the generator model tile file

filename = 'generator_model_%03d.h5' % (epoch + 1)

g_model.save(filename)

# train the generator and discriminator

def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=256):

bat_per_epo = int(dataset.shape[0] / n_batch)

half_batch = int(n_batch / 2)

# manually enumerate epochs

for i in range(n_epochs):

# enumerate batches over the training set

for j in range(bat_per_epo):

# get randomly selected 'real' samples

X_real, y_real = generate_real_samples(dataset, half_batch)

# generate 'fake' examples

X_fake, y_fake = generate_fake_samples(g_model, latent_dim, half_batch)

# create training set for the discriminator

X, y = vstack((X_real, X_fake)), vstack((y_real, y_fake))

# update discriminator model weights

d_loss, _ = d_model.train_on_batch(X, y)

# prepare points in latent space as input for the generator
```

```
X_gan = generate_latent_points(latent_dim, n_batch)

# create inverted labels for the fake samples

y_gan = ones((n_batch, 1))

# update the generator via the discriminator's error

g_loss = gan_model.train_on_batch(X_gan, y_gan)

# summarize loss on this batch

print('>%d, %d/%d, d=%.3f, g=%.3f' % (i+1, j+1, bat_per_epo, d_loss, g_loss))

# evaluate the model performance, sometimes

if (i+1) % 10 == 0:

summarize_performance(i, g_model, d_model, dataset, latent_dim)

# size of the latent space

latent_dim = 100

# create the discriminator

d_model = define_discriminator()

# create the generator

g_model = define_generator(latent_dim)

# create the gan

gan_model = define_gan(g_model, d_model)

# load image data

dataset = load_real_samples()

# train model

train(g_model, d_model, gan_model, dataset, latent_dim)

# How to Use the Final Generator Model to Generate Images
```

```
# example of loading the generator model and generating images
from keras.models import load_model
from numpy.random import randn
from matplotlib import pyplot
# generate points in latent space as input for the generator
def generate_latent_points(latent_dim, n_samples):
# generate points in the latent space
x_input = randn(latent_dim * n_samples)
# reshape into a batch of inputs for the network
x_input = x_input.reshape(n_samples, latent_dim)
return x_input
# create and save a plot of generated images (reversed grayscale)
def save_plot(examples, n):
# plot images
for i in range(n * n):
# define subplot
pyplot.subplot(n, n, 1 + i)
# turn off axis
pyplot.axis('off')
# plot raw pixel data
pyplot.imshow(examples[i, :, :, 0], cmap='gray_r')
pyplot.show()
# load model
```

```
model = load_model('generator_model_100.h5')

# generate images

latent_points = generate_latent_points(100, 25)

# generate images

X = model.predict(latent_points)

# plot the result

save_plot(X, 5)

# example of generating an image for a specific point in the latent space

from keras.models import load_model

from numpy import asarray

from matplotlib import pyplot

# load model

model = load_model('generator_model_100.h5')

# all 0s

vector = asarray([[0.0 for _ in range(100)]])

# generate image

X = model.predict(vector)

# plot the result

pyplot.imshow(X[0, :, :, 0], cmap='gray_r')

pyplot.show()
```

Input Snapshot:

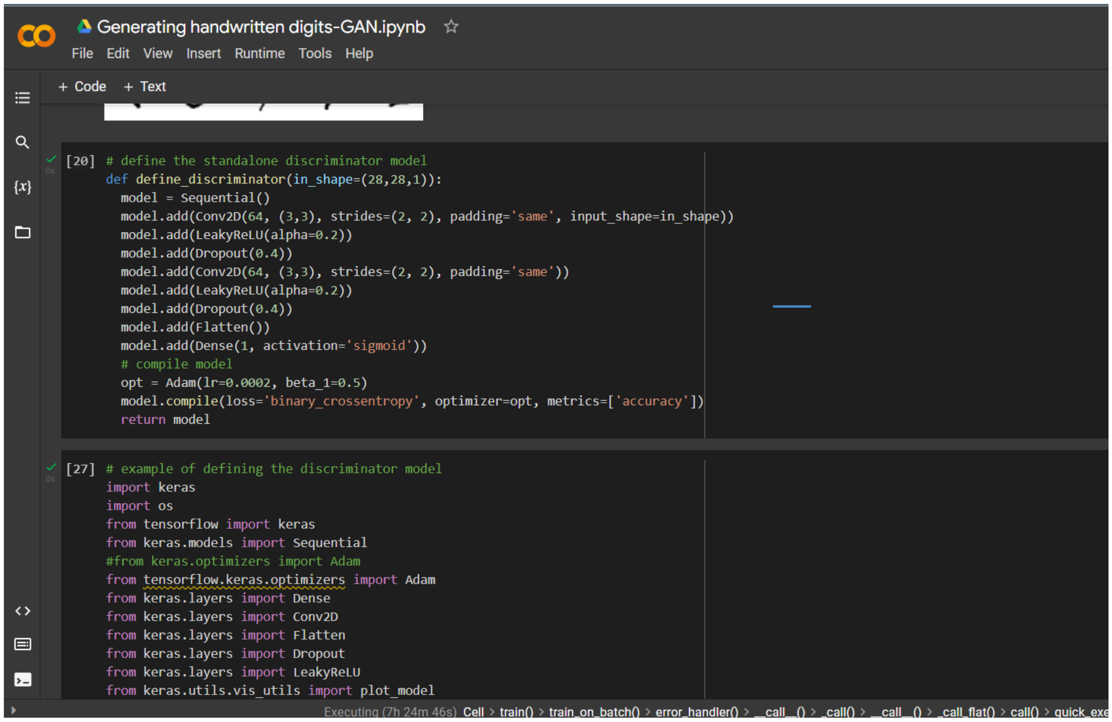


Figure1: Implementation of Generating handwritten digits using GAN in Colab.

Note: This portion of code did not run. It was loading all day long.

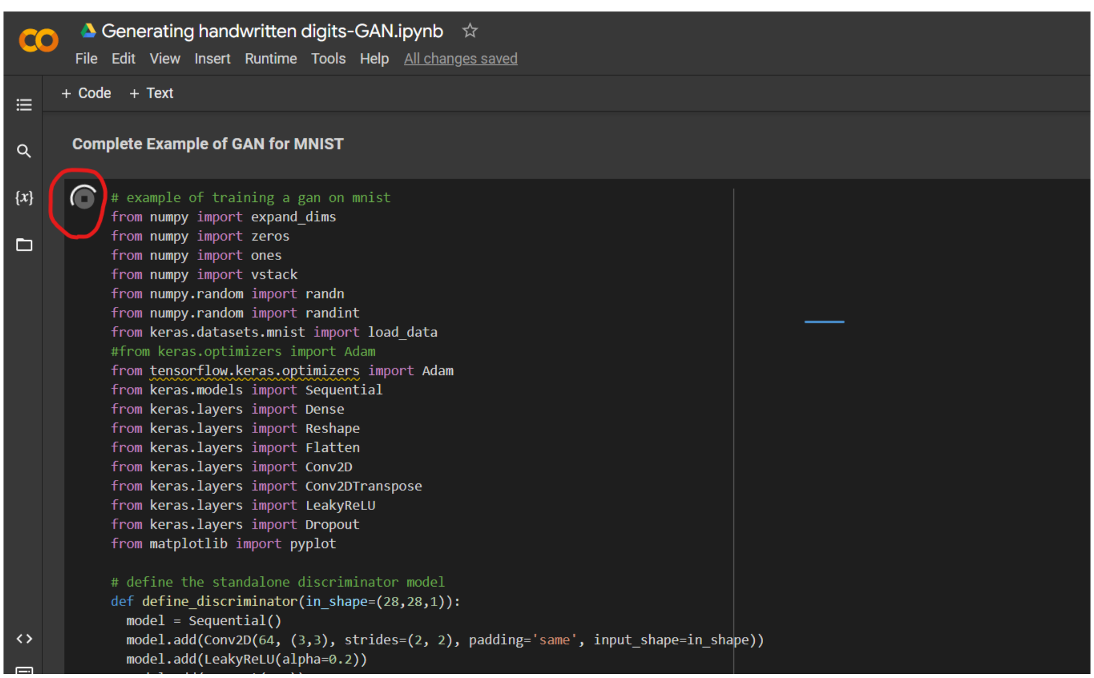
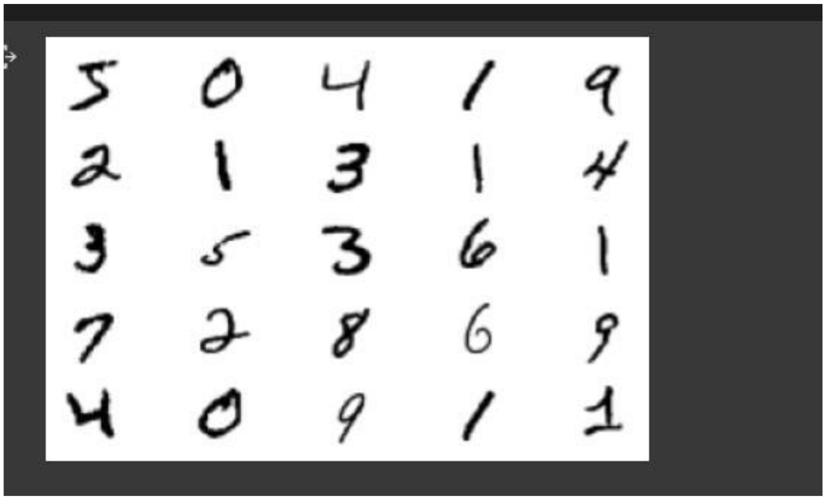
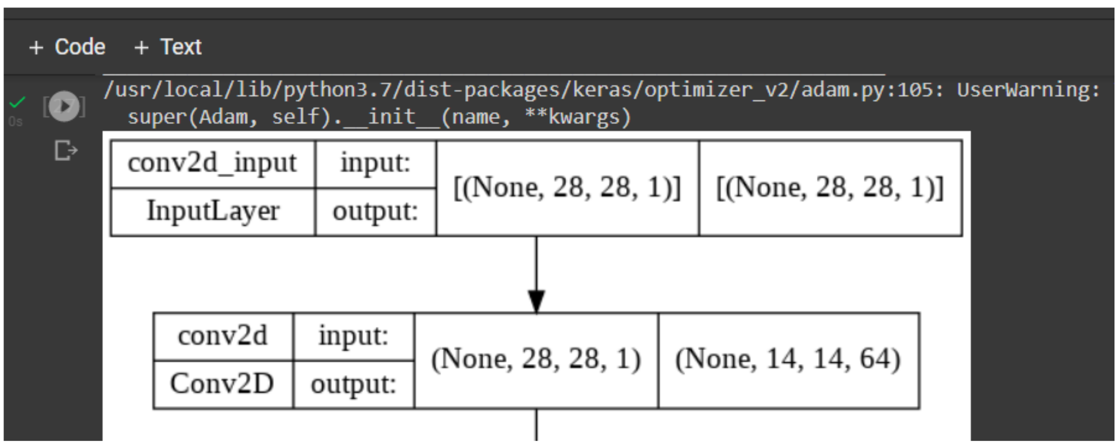


Figure2: Implementation of Generating handwritten digits using GAN in Colab.

Output Snapshot:

Some of the snaps of output.





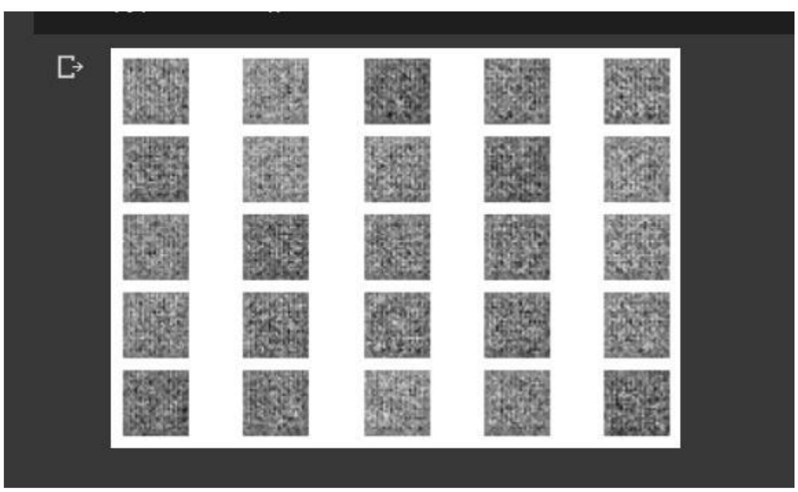


Figure3: Output Snaps

RNN

RNN have become extremely popular in the deep learning space which makes learning them even more imperative. We can do sequence prediction problem using RNN. One of the simplest tasks for this is sine wave prediction. The sequence contains a visible trend and is easy to solve using heuristics. This is what a sine wave looks like:

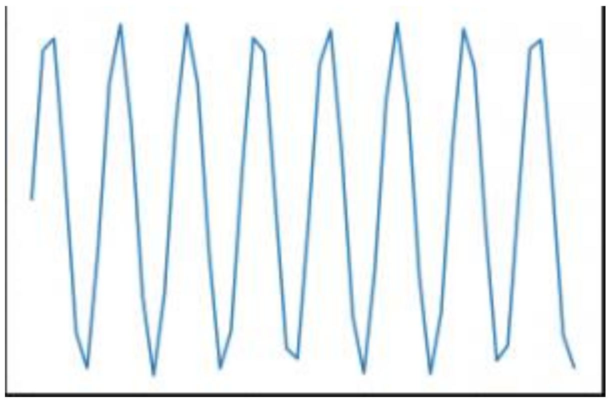


Figure4: Sine Wave

Code:

```
%pylab inline

import math

sin_wave = np.array([math.sin(x) for x in np.arange(200)])

plt.plot(sin_wave[:50])

X = []

Y = []

seq_len = 50

num_records = len(sin_wave) - seq_len
```

```
for i in range(num_records - 50):

X.append(sin_wave[i:i+seq_len])

Y.append(sin_wave[i+seq_len])

X = np.array(X)

X = np.expand_dims(X, axis=2)

Y = np.array(Y)

Y = np.expand_dims(Y, axis=1)

X.shape, Y.shape

X_val = []

Y_val = []

for i in range(num_records - 50, num_records):

X_val.append(sin_wave[i:i+seq_len])

Y_val.append(sin_wave[i+seq_len])

X_val = np.array(X_val)

X_val = np.expand_dims(X_val, axis=2)

Y_val = np.array(Y_val)

Y_val = np.expand_dims(Y_val, axis=1)

learning_rate = 0.0001

nepoch = 25
```

```
T = 50    # length of sequence

hidden_dim = 100

output_dim = 1

bptt_truncate = 5

min_clip_value = -10

max_clip_value = 10

U = np.random.uniform(0, 1, (hidden_dim, T))

W = np.random.uniform(0, 1, (hidden_dim, hidden_dim))

V = np.random.uniform(0, 1, (output_dim, hidden_dim))

def sigmoid(x):

return 1 / (1 + np.exp(-x))

for epoch in range(nepoch):

# check loss on train

loss = 0.0

# do a forward pass to get prediction

for i in range(Y.shape[0]):

x, y = X[i], Y[i]                    # get input, output values of each record

prev_s = np.zeros((hidden_dim, 1))   # here, prev-s is the value of the previous activation of
hidden layer; which is initialized as all zeroes
```

```
for t in range(T):

new_input = np.zeros(x.shape)    # we then do a forward pass for every timestep in the sequence

new_input[t] = x[t]              # for this, we define a single input for that timestep

mulu = np.dot(U, new_input)

mulw = np.dot(W, prev_s)

add = mulw + mulu

s = sigmoid(add)

mulv = np.dot(V, s)

prev_s = s

# calculate error

loss_per_record = (y - mulv)**2 / 2

loss += loss_per_record

loss = loss / float(y.shape[0])

# check loss on val

val_loss = 0.0

for i in range(Y_val.shape[0]):

x, y = X_val[i], Y_val[i]

prev_s = np.zeros((hidden_dim, 1))

for t in range(T):
```

```
new_input = np.zeros(x.shape)

new_input[t] = x[t]

mulu = np.dot(U, new_input)

mulw = np.dot(W, prev_s)

add = mulw + mulu

s = sigmoid(add)

mulv = np.dot(V, s)

prev_s = s

loss_per_record = (y - mulv)**2 / 2

val_loss += loss_per_record

val_loss = val_loss / float(y.shape[0])

print('Epoch: ', epoch + 1, ', Loss: ', loss, ', Val Loss: ', val_loss)

# train model

for i in range(Y.shape[0]):

x, y = X[i], Y[i]

layers = []

prev_s = np.zeros((hidden_dim, 1))

dU = np.zeros(U.shape)

dV = np.zeros(V.shape)
```

```
dW = np.zeros(W.shape)

dU_t = np.zeros(U.shape)

dV_t = np.zeros(V.shape)

dW_t = np.zeros(W.shape)

dU_i = np.zeros(U.shape)

dW_i = np.zeros(W.shape)

# forward pass

for t in range(T):

new_input = np.zeros(x.shape)

new_input[t] = x[t]

mulu = np.dot(U, new_input)

mulw = np.dot(W, prev_s)

add = mulw + mulu

s = sigmoid(add)

mulv = np.dot(V, s)

layers.append({'s':s, 'prev_s':prev_s})

prev_s = s

# derivative of pred

dmulv = (mulv - y)
```

```
# backward pass

for t in range(T):

dV_t = np.dot(dmulv, np.transpose(layers[t]['s']))

dsv = np.dot(np.transpose(V), dmulv)

ds = dsv

dadd = add * (1 - add) * ds

dmulw = dadd * np.ones_like(mulw)

dprev_s = np.dot(np.transpose(W), dmulw)

for i in range(t-1, max(-1, t-bptt_truncate-1), -1):

ds = dsv + dprev_s

dadd = add * (1 - add) * ds

dmulw = dadd * np.ones_like(mulw)

dmulu = dadd * np.ones_like(mulu)

dW_i = np.dot(W, layers[t]['prev_s'])

dprev_s = np.dot(np.transpose(W), dmulw)

new_input = np.zeros(x.shape)

new_input[t] = x[t]

dU_i = np.dot(U, new_input)

dx = np.dot(np.transpose(U), dmulu)
```

```
dU_t += dU_i

dW_t += dW_i

dV += dV_t

dU += dU_t

dW += dW_t

if dU.max() > max_clip_value:

dU[dU > max_clip_value] = max_clip_value

if dV.max() > max_clip_value:

dV[dV > max_clip_value] = max_clip_value

if  dW.max() > max_clip_value:

dW[dW > max_clip_value] = max_clip_value

if dU.min() < min_clip_value:

dU[dU < min_clip_value] = min_clip_value

if dV.min() < min_clip_value:

dV[dV < min_clip_value] = min_clip_value

if dW.min() < min_clip_value:

dW[dW < min_clip_value] = min_clip_value

# update

U -= learning_rate * dU
```

```
V -= learning_rate * dV

W -= learning_rate * dW

preds = []

for i in range(Y.shape[0]):

x, y = X[i], Y[i]

prev_s = np.zeros((hidden_dim, 1))

# Forward pass

for t in range(T):

mulu = np.dot(U, x)

mulw = np.dot(W, prev_s)

add = mulw + mulu

s = sigmoid(add)

mulv = np.dot(V, s)

prev_s = s

preds.append(mulv)

preds = np.array(preds)

plt.plot(preds[:, 0, 0], 'g')

plt.plot(Y[:, 0], 'r')

plt.show()
```

```
preds = []

for i in range(Y_val.shape[0]):

x, y = X_val[i], Y_val[i]

prev_s = np.zeros((hidden_dim, 1))

# For each time step...

for t in range(T):

mulu = np.dot(U, x)

mulw = np.dot(W, prev_s)

add = mulw + mulu

s = sigmoid(add)

mulv = np.dot(V, s)

prev_s = s

preds.append(mulv)

preds = np.array(preds)

plt.plot(preds[:, 0, 0], 'g')

plt.plot(Y_val[:, 0], 'r')

plt.show()
```

Input Snapshot:

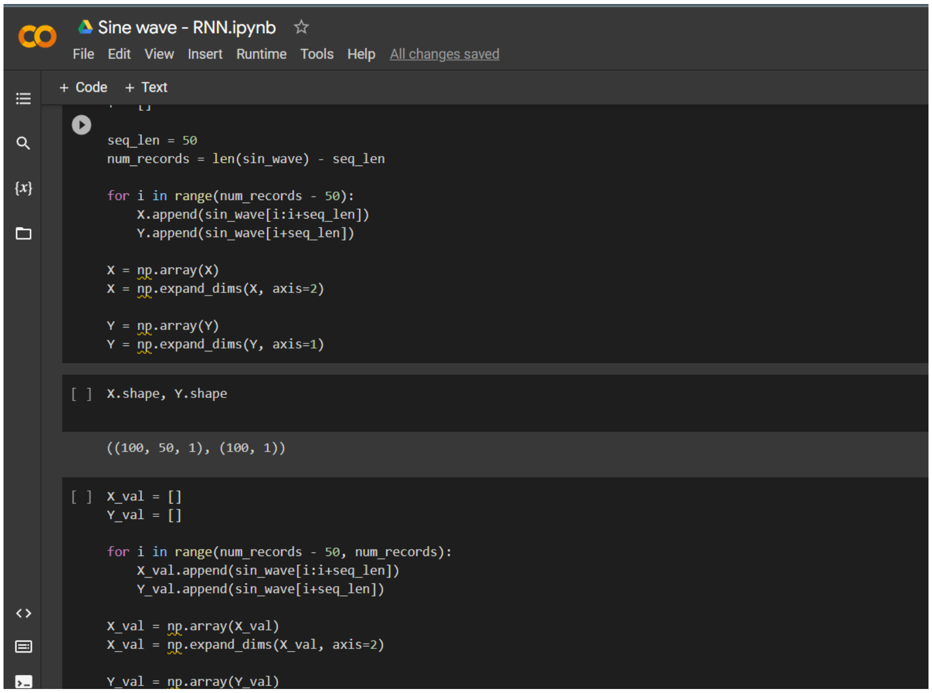
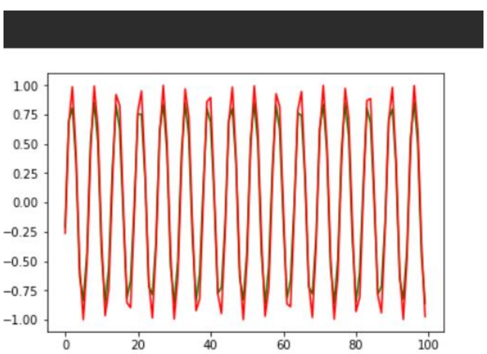


Figure5: Implementation of Sine wave sequence prediction using RNN in Colab.

Output Snapshot:

Some of the snaps of output.



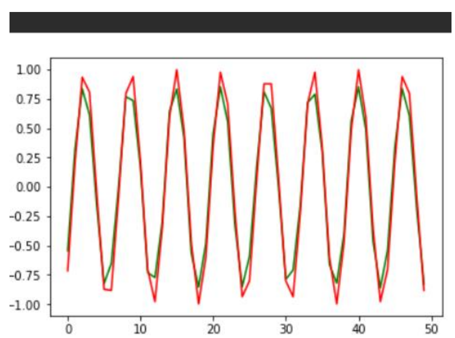


Figure6: Output Snaps.

SOM

The Self-Organizing Map is one of the most popular neural models. It belongs to the category of the competitive learning network. The SOM is based on unsupervised learning, which means that no human intervention is needed during the training and those little needs to be known about characterized by the input data. Here, giving an example of real world application of SOM: Credit Card Fraud Detection.

Code:

```
# import the Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from google.colab import files

files.upload()

# import the dataset

dataset = pd.read_csv('Credit_Card_Applications.csv')

X = dataset.iloc [: ,:-1].values    # independent variables

y = dataset.iloc [: , -1].values    # dependent variables

# feature Scaling

from sklearn.preprocessing import MinMaxScaler

sc = MinMaxScaler(feature_range = (0,1))

X = sc.fit_transform(X)

# feature Scaling

from sklearn.preprocessing import MinMaxScaler
```

```
sc = MinMaxScaler(feature_range = (0,1))

X = sc.fit_transform(X)

# import the SOM model

from minisom import MiniSom

# init the model

som = MiniSom( x = 10, y = 10, input_len = 15, sigma = 1.0, learning_rate = 0.5)

# init the weight

som.random_weights_init(X)

# traing the model

som.train_random(data = X, num_iteration = 100)

# making a self organization map

from pylab import bone, pcolor, colorbar, plot, show

bone()

pcolor(som.distance_map().T)

colorbar()

markers = ['o' , 's']

colors = ['r', 'g']

for i, x in enumerate(X):

w = som.winner(x)

plot(w[0] + 0.5,

w[1] + 0.5,

markers[y[i]],

markeredgecolor = colors[y[i]],
```

markerfacecolor = 'None',

markersize = 10,

markeredgewidth = 2)

show()

mapping the winning node

mappings = som.win_map(X)

#catch the cheater

frauds = np.concatenate((mappings[(7,8)], mappings[(3,1)], mappings[(5,1)]) , axis=0)

rescale the value using inverse function

frauds = sc.inverse_transform(frauds)

frauds

Input Snapshot:

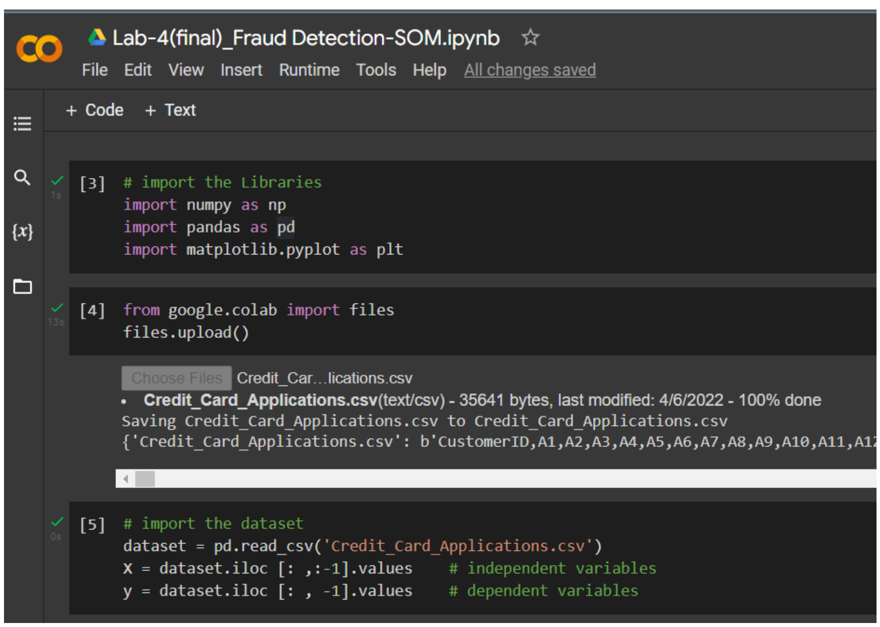
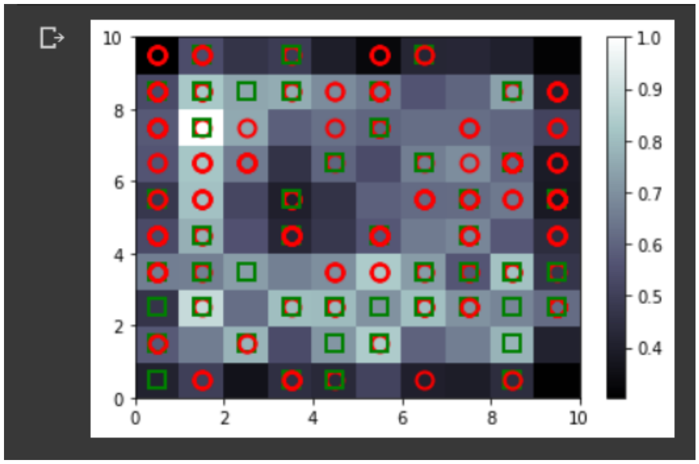


Figure7: Implementation of Credit Card Detection using SOM in Colab

Output Snapshot:




```
array([[1.5748499e+07, 1.0000000e+00, 4.4330000e+01, 5.0000000e-01,
        2.0000000e+00, 3.0000000e+00, 8.0000000e+00, 5.0000000e+00,
        1.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 3.2000000e+02, 1.0000000e+00],
       [1.5781975e+07, 1.0000000e+00, 5.6000000e+01, 1.2500000e+01,
        2.0000000e+00, 4.0000000e+00, 8.0000000e+00, 8.0000000e+00,
        1.0000000e+00, 0.0000000e+00, 0.0000000e+00, 1.0000000e+00,
        2.0000000e+00, 2.4000000e+01, 2.0290000e+03],
       [1.5605791e+07, 1.0000000e+00, 1.9500000e+01, 9.5850000e+00,
        2.0000000e+00, 6.0000000e+00, 4.0000000e+00, 7.9000000e-01,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
        2.0000000e+00, 8.0000000e+01, 3.5100000e+02],
       [1.5571415e+07, 1.0000000e+00, 3.7580000e+01, 0.0000000e+00,
        2.0000000e+00, 8.0000000e+00, 4.0000000e+00, 0.0000000e+00,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
        3.0000000e+00, 1.8400000e+02, 1.0000000e+00],
       [1.5565714e+07, 1.0000000e+00, 4.2750000e+01, 4.0850000e+00,
        2.0000000e+00, 6.0000000e+00, 4.0000000e+00, 4.0000000e-02,
        0.0000000e+00, 0.0000000e+00, 0.0000000e+00, 0.0000000e+00,
        2.0000000e+00, 1.0800000e+02, 1.0100000e+02],
```

Figure8: Output Snaps.

CNN

Convolutional Neural Network or CNN is a type of artificial neural network, which is widely used for image/object recognition and classification. Deep Learning thus recognizes objects in an image by using a CNN.In neural networks, Convolutional neural network (ConvNets or CNNs) is one of the main categories to do images recognition, images classifications. Objects detections, recognition faces etc., are some of the areas where CNNs are widely used. CNN image classifications takes an input image, process it and classify it under certain categories (Eg., Dog, Cat, Tiger, Lion). A convolution neural network has multiple hidden layers that help in extracting information from an image. The four important layers in CNN are:

1. Convolution layer
2. ReLU layer
3. Pooling layer
4. Fully connected layer

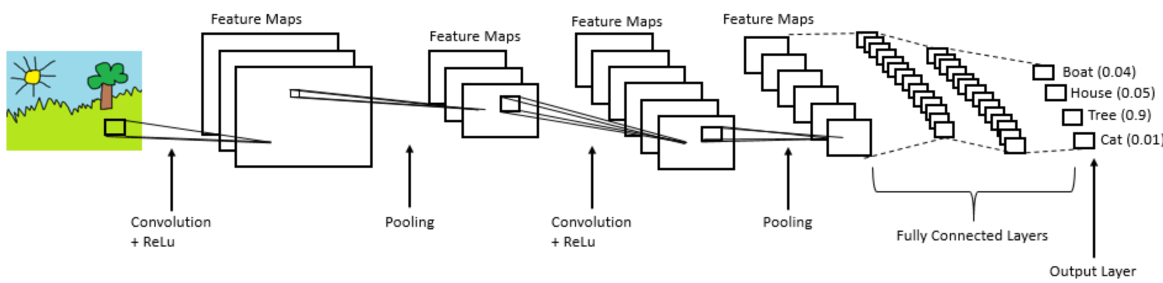


Figure9: Complete CNN architecture

How it works:

- Provide input image into convolution layer.
- Choose parameters, apply filters with strides, padding if requires. Perform convolution on the image and apply ReLU activation to the matrix.
- Perform pooling to reduce dimensionality size.
- Add as many convolutional layers until satisfied.
- Flatten the output and feed into a fully connected layer (FC Layer).
- Output the class using an activation function (Logistic Regression with cost functions) and classifies images.