Capsule-Triple GAN with Wild horse optimization based Classification

```
# math libraries
import numpy as np
import tensorflow as tf
from keras              import layers                   as lyrs
from keras              import models                   as mdls
from keras import layers, models, optimizers
from keras import backend as K
from tensorflow.keras.utils import to_categorical
from keras.datasets import mnist, cifar10
from keras.layers import Input, Dense, Reshape, Flatten, Dropout, Lambda, Concatenate, Multiply
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from tensorflow.keras.optimizers import Adam
from keras.preprocessing.image import ImageDataGenerator
from keras import callbacks

# visualization
import skimage
from skimage import data, color, exposure
from skimage.transform import resize
import matplotlib.pyplot as plt
%matplotlib inline

# sys and helpers
import sys
import os
import glob
from tqdm import tqdm

print('Modules imported.')
# device check
from tensorflow.python.client import device_lib
print('Devices:', device_lib.list_local_devices())

# GPU check
if not tf.test.gpu_device_name():
    print('No GPU found.')
else:
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))
```

```
Modules imported.
Devices: [name: "/device:CPU:0"
device_type: "CPU"
memory_limit: 268435456
locality {
}
incarnation: 907583267152696063
xla_global_id: -1
]
No GPU found.
```

```
def load_dataset(dataset, width, height, channels):

    if dataset == 'mnist':
        # load MNIST data
        (X_train, y_train), (X_test, y_test) = mnist.load_data()

        # rescale -1 to 1
        X_train = (X_train.astype(np.float32) - 127.5) / 127.5
        X_train = np.expand_dims(X_train, axis=3)

    # defining input dims
    img_rows = width
    img_cols = height
    channels = channels
    img_shape = [img_rows, img_cols, channels]
    return X_train, img_shape

dataset, shape = load_dataset('mnist', 28, 28, 1)
print('Dataset shape: {0}, Image shape: {1}'.format(dataset.shape, shape))
```

```
Dataset shape: (60000, 28, 28, 1), Image shape: [28, 28, 1]
```

Capsule Concept

build_discriminator

```
def squash(vectors, axis=-1):
    """
    The non-linear activation used in Capsule. It drives the length of a large vector to near 1 and small vector to 0"""

    s_squared_norm = K.sum(K.square(vectors), axis, keepdims=True)
    scale = s_squared_norm / (1 + s_squared_norm) / K.sqrt(s_squared_norm + K.epsilon())
    return scale * vectors


def build_discriminator():

    """
    This is the part my 'Capsule Layer as a Discriminator in Generative Adversarial Networks'

    """

    # depending on dataset we define input shape for our network
    img = Input(shape=(shape[0], shape[1], shape[2]))

    # first typical convlayer outputs a 20x20x256 matrix
    x = Conv2D(filters=256, kernel_size=9, strides=1, padding='valid', name='conv1')(img)
    x = LeakyReLU()(x)

    # original 'Dynamic Routing Between Capsules' paper does not include the batch norm layer after the first conv group
    x = BatchNormalization(momentum=0.8)(x)

    print(x)
    """
    NOTE: Capsule architecture starts from here.
    """
    #
    # primarycaps coming first
    #

    # filters 256 (n_vectors=8 * channels=32)
    x = Conv2D(filters=8 * 32, kernel_size=9, strides=2, padding='valid', name='primarycap_conv2')(x)

    # reshape into the 8D vector for all 32 feature maps combined
    # (primary capsule has collections of activations which denote orientation of the digit
    # while intensity of the vector which denotes the presence of the digit)
    x = Reshape(target_shape=[-1, 8], name='primarycap_reshape')(x)

    # the purpose is to output a number between 0 and 1 for each capsule where the length of the input decides the amount
    x = Lambda(squash, name='primarycap_squash')(x)
    x = BatchNormalization(momentum=0.8)(x)
    x = Flatten()(x)
    # capsule (i) in a lower-level layer needs to decide how to send its output vector to higher-level capsules (j)
    # it makes this decision by changing scalar weight (c=coupling coefficient) that will multiply its output vector and then be treated
    #
    # uhat = prediction vector, w = weight matrix but will act as a dense layer, u = output from a previous layer
    # uhat = u * w
    # neurons 160 (num_capsules=10 * num_vectors=16)
    uhat = Dense(160, kernel_initializer='he_normal', bias_initializer='zeros', name='uhat_digitcaps')(x)

    # c = coupling coefficient (softmax over the bias weights, log prior) | "the coupling coefficients between capsule (i) and all the ca
    # we treat the coupling coefficiant as a softmax over bias weights from the previous dense layer
    c = Activation('softmax', name='softmax_digitcaps1')(uhat) # softmax will make sure that each weight c_ij is a non-negative number ar

    # s_j (output of the current capsule level) = uhat * c
    c = Dense(160)(c) # compute s_j
    x = Multiply()([uhat, c])
    """
    NOTE: Squashing the capsule outputs creates severe blurry artifacts, thus we replace it with Leaky ReLu.
    """
    s_j = LeakyReLU()(x)


    #
    # we will repeat the routing part 2 more times (num_routing=3) to unfold the loop
    #
    c = Activation('softmax', name='softmax_digitcaps2')(s_j) # softmax will make sure that each weight c_ij is a non-negative number anc
    c = Dense(160)(c) # compute s_j
    x = Multiply()([uhat, c])
```

```
    s_j = LeakyReLU()(x)

    c = Activation('softmax', name='softmax_digitcaps3')(s_j) # softmax will make sure that each weight c_ij is a non-negative number anc
    c = Dense(160)(c) # compute s_j
    x = Multiply()([uhat, c])
    s_j = LeakyReLU()(x)

    pred = Dense(100, activation='relu')(s_j)


    return Model(img, pred) ,pred

discriminator,pre = build_discriminator()
print('DISCRIMINATOR:')
discriminator.summary()
discriminator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
```

```
 conv1 (Conv2D)               (None, 20, 20, 256)  20992       ['input_6[0][0]']

 leaky_re_lu_4 (LeakyReLU)    (None, 20, 20, 256)  0           ['conv1[0][0]']

 batch_normalization_5 (BatchNo  (None, 20, 20, 256)  1024     ['leaky_re_lu_4[0][0]']
 rmalization)

 primarycap_conv2 (Conv2D)    (None, 6, 6, 256)    5308672     ['batch_normalization_5[0][0]']

 primarycap_reshape (Reshape)  (None, 1152, 8)     0           ['primarycap_conv2[0][0]']

 primarycap_squash (Lambda)   (None, 1152, 8)      0           ['primarycap_reshape[0][0]']

 batch_normalization_6 (BatchNo  (None, 1152, 8)   32          ['primarycap_squash[0][0]']
 rmalization)

 flatten_1 (Flatten)          (None, 9216)         0           ['batch_normalization_6[0][0]']

 uhat_digitcaps (Dense)       (None, 160)          1474720     ['flatten_1[0][0]']

 softmax_digitcaps1 (Activation  (None, 160)       0           ['uhat_digitcaps[0][0]']
 )

 dense_10 (Dense)             (None, 160)          25760       ['softmax_digitcaps1[0][0]']

 multiply_3 (Multiply)        (None, 160)          0           ['uhat_digitcaps[0][0]',
                                                                'dense_10[0][0]']

 leaky_re_lu_5 (LeakyReLU)    (None, 160)          0           ['multiply_3[0][0]']

 softmax_digitcaps2 (Activation  (None, 160)       0           ['leaky_re_lu_5[0][0]']
 )

 dense_11 (Dense)             (None, 160)          25760       ['softmax_digitcaps2[0][0]']

 multiply_4 (Multiply)        (None, 160)          0           ['uhat_digitcaps[0][0]',
                                                                'dense_11[0][0]']

 leaky_re_lu_6 (LeakyReLU)    (None, 160)          0           ['multiply_4[0][0]']

 softmax_digitcaps3 (Activation  (None, 160)       0           ['leaky_re_lu_6[0][0]']
 )

 dense_12 (Dense)             (None, 160)          25760       ['softmax_digitcaps3[0][0]']

 multiply_5 (Multiply)        (None, 160)          0           ['uhat_digitcaps[0][0]',
                                                                'dense_12[0][0]']

 leaky_re_lu_7 (LeakyReLU)    (None, 160)          0           ['multiply_5[0][0]']

 dense_13 (Dense)             (None, 100)          16100       ['leaky_re_lu_7[0][0]']

==================================================================================================
Total params: 6,898,820
Trainable params: 6,898,292
Non-trainable params: 528
__________________________________________________________________________________________________
```

build_generator

```
def build_generator():

    noise_shape = (100,)
    x_noise = Input(shape=noise_shape)

    # we apply different kernel sizes in order to match the original image size
```

```
    if (shape[0] == 28 and shape[1] == 28):
        x = Dense(128 * 7 * 7, activation="relu")(x_noise)
        x = Reshape((7, 7, 128))(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = UpSampling2D()(x)
        x = Conv2D(128, kernel_size=3, padding="same")(x)
        x = Activation("relu")(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = UpSampling2D()(x)
        x = Conv2D(64, kernel_size=3, padding="same")(x)
        x = Activation("relu")(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = Conv2D(1, kernel_size=3, padding="same")(x)
        gen_out = Activation("tanh")(x)

        return Model(x_noise, gen_out)

    if (shape[0] == 32 and shape[1] == 32):
        x = Dense(128 * 8 * 8, activation="relu")(x_noise)
        x = Reshape((8, 8, 128))(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = UpSampling2D()(x)
        x = Conv2D(128, kernel_size=3, padding="same")(x)
        x = Activation("relu")(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = UpSampling2D()(x)
        x = Conv2D(64, kernel_size=3, padding="same")(x)
        x = Activation("relu")(x)
        x = BatchNormalization(momentum=0.8)(x)
        x = Conv2D(3, kernel_size=3, padding="same")(x)
        gen_out = Activation("tanh")(x)


        return Model(x_noise, gen_out)


# build and compile the generator
generator = build_generator()
print('GENERATOR:')
generator.summary()
generator.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```

```
GENERATOR:
Model: "model_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_7 (InputLayer)        [(None, 100)]             0

 dense_14 (Dense)            (None, 6272)              633472

 reshape_1 (Reshape)         (None, 7, 7, 128)         0

 batch_normalization_7 (Batc  (None, 7, 7, 128)        512
 hNormalization)

 up_sampling2d_2 (UpSampling  (None, 14, 14, 128)      0
 2D)

 conv2d_3 (Conv2D)           (None, 14, 14, 128)       147584

 activation_3 (Activation)   (None, 14, 14, 128)       0

 batch_normalization_8 (Batc  (None, 14, 14, 128)      512
 hNormalization)

 up_sampling2d_3 (UpSampling  (None, 28, 28, 128)      0
 2D)

 conv2d_4 (Conv2D)           (None, 28, 28, 64)        73792

 activation_4 (Activation)   (None, 28, 28, 64)        0

 batch_normalization_9 (Batc  (None, 28, 28, 64)       256
 hNormalization)

 conv2d_5 (Conv2D)           (None, 28, 28, 1)         577

 activation_5 (Activation)   (None, 28, 28, 1)         0

=================================================================
Total params: 856,705
Trainable params: 856,065
Non-trainable params: 640
_________________________________________________________________
```

build_classifer

```
def build_classifer(pre):

    layer1 = Dense(100, activation='relu')(pre)
    layer2 = Dense(75, activation='relu')(layer1)
    layer3= Dense(50, activation='relu')(layer2)
    layer4= Dense(25, activation='sigmoid')(layer3)
    pred = Dense(1, activation='sigmoid')(layer4)


    return Model(pre, pred)
```

Double-click (or enter) to edit

```
classifer= build_classifer(pre)
print('classifer:')
classifer.summary()
classifer.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5), metrics=['accuracy'])
```

```
classifer:
Model: "model_7"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_8 (InputLayer)        [(None, 100)]             0         
                                                                 
 dense_15 (Dense)            (None, 100)               10100     
                                                                 
 dense_16 (Dense)            (None, 75)                7575      
                                                                 
 dense_17 (Dense)            (None, 50)                3800      
                                                                 
 dense_18 (Dense)            (None, 25)                1275      
                                                                 
 dense_19 (Dense)            (None, 1)                 26        
                                                                 
=================================================================
Total params: 22,776
Trainable params: 22,776
Non-trainable params: 0
_________________________________________________________________
```

Build Triple GAN

```
z = Input(shape=(100,))
img = generator(z)
discriminator.trainable = False
valid = discriminator(img)
cl=classifer(valid)
combined = Model(z,cl)
print('COMBINED:')
combined.summary()
combined.compile(loss='binary_crossentropy', optimizer=Adam(0.0002, 0.5))
```

```
COMBINED:
Model: "model_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_9 (InputLayer)        [(None, 100)]             0         
                                                                 
 model_6 (Functional)        (None, 28, 28, 1)         856705    
                                                                 
 model_5 (Functional)        (None, 100)               6898820   
                                                                 
 model_7 (Functional)        (None, 1)                 22776     
                                                                 
=================================================================
Total params: 7,778,301
Trainable params: 878,841
Non-trainable params: 6,899,460
_________________________________________________________________
```

```
D_L_REAL = []
D_L_FAKE = []
D_L = []
```

```
D_ACC = []
G_L = []
C_L = []

def save_imgs(dataset_title, epoch):
        r, c = 5, 5
        noise = np.random.normal(0, 1, (r * c, 100))
        gen_imgs = generator.predict(noise)

        # rescale images 0 - 1
        gen_imgs = 0.5 * gen_imgs + 0.5

        fig, axs = plt.subplots(r, c)
        cnt = 0

        # iterate in order to create a subplot
        for i in range(r):
            for j in range(c):
                if dataset_title == 'mnist':
                    axs[i,j].imshow(gen_imgs[cnt, :,:,0], cmap='gray')
                    axs[i,j].axis('off')
                    cnt += 1
                elif dataset_title == 'cifar10':
                    axs[i,j].imshow(gen_imgs[cnt, :,:,:])
                    axs[i,j].axis('off')
                    cnt += 1
                else:
                    print('Please indicate the image options.')

        if not os.path.exists('images_{0}'.format(dataset_title)):
            os.makedirs('images_{0}'.format(dataset_title))

        fig.savefig("images_{0}/{1}.png".format(dataset_title, epoch))
        plt.close()


history = train('mnist', epochs=100, batch_size=32, save_interval=50)

generator.save('mnist_model.h5')
```

```
42 [D loss: 1.488921] [G loss: 0.119559][C loss: 0.116804,acc.: 88.32%]
43 [D loss: 1.468638] [G loss: 0.113184][C loss: 0.110502,acc.: 88.95%]
44 [D loss: 1.483024] [G loss: 0.107114][C loss: 0.104804,acc.: 89.52%]
45 [D loss: 1.500237] [G loss: 0.101942][C loss: 0.099802,acc.: 90.02%]
46 [D loss: 1.426166] [G loss: 0.096938][C loss: 0.095187,acc.: 90.48%]
```

```
89 [D loss: 1.225300] [G loss: 0.051380][C loss: 0.051061,acc.: 94.89%]
90 [D loss: 1.246238] [G loss: 0.050794][C loss: 0.050537,acc.: 94.95%]
91 [D loss: 1.265237] [G loss: 0.050161][C loss: 0.049936,acc.: 95.01%]
92 [D loss: 1.246063] [G loss: 0.049452][C loss: 0.049282,acc.: 95.07%]
93 [D loss: 1.236745] [G loss: 0.048958][C loss: 0.048793,acc.: 95.12%]
94 [D loss: 1.201630] [G loss: 0.048458][C loss: 0.048322,acc.: 95.17%]
95 [D loss: 1.217942] [G loss: 0.048032][C loss: 0.047916,acc.: 95.21%]
96 [D loss: 1.184759] [G loss: 0.047729][C loss: 0.047621,acc.: 95.24%]
97 [D loss: 1.179074] [G loss: 0.047527][C loss: 0.047419,acc.: 95.26%]
98 [D loss: 1.187752] [G loss: 0.047364][C loss: 0.047262,acc.: 95.27%]
99 [D loss: 1.185374] [G loss: 0.047111][C loss: 0.047010,acc.: 95.30%]
```

Wild horse optimizer

```
import random
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')

lb=0
ub=1
d=2
nPop=100
pos=np.zeros((nPop,d))
for i in range(nPop):
 for j in range(d):
   pos[i][j]=(ub-lb)*random.random()+lb
```

```
from hyperopt import fmin, tpe, hp,Trials
trials = Trials()
fit=[]
def fitness(x):
  fitval=x[0]+x[1]
  return fitval
best = fmin(fn=lambda x: x ** 2,
            space= hp.uniform('x', -10, 10),
            algo=tpe.suggest,
            max_evals=50,
            trials = trials)
dataset, shape = load_dataset('mnist', 28, 28, 1)
print(best)
for i in range(nPop):
  fit.append(fitness(pos[i,:]))

fit=np.array(fit)
```

```
100%|██████████| 50/50 [00:00<00:00, 419.16it/s, best loss: 0.0051779342065860975]
{'x': 0.07195786410522548}
```

```
# Check whether the problem is minimization problem or maximization problem
problem='max'
if problem=='max':
  best_fit=np.max(fit)
  idx=np.unravel_index(np.argmax(fit, axis=None), fit.shape)
  best_pos=pos[idx,:]
elif problem=='min':
  best_fit=np.min(fit)
  idx=np.unravel_index(np.argmax(fit, axis=None), fit.shape)
  best_pos=pos[idx,:]
print('Best fitness :',best_fit)
print('Best position :',best_pos)
```

```
Best fitness : 1.8094475927560583
Best position : [[0.92607832 0.88336927]]
```

```
def train(dataset_title, epochs, batch_size=32, save_interval=50):

        half_batch = int(batch_size / 2)

        for epoch in range(epochs):

            # ---------------------
            #  Train Discriminator
            # ---------------------

            # select a random half batch of images
            idx = np.random.randint(0, dataset.shape[0], half_batch)
            imgs = dataset[idx]
```

```
            noise = np.random.normal(0, 1, (half_batch, 100))

            # generate a half batch of new images
            gen_imgs = generator.predict(noise)

            # train the discriminator by feeding both real and fake (generated) images one by one
            d_loss_real = discriminator.train_on_batch(imgs, np.ones((half_batch, 1))*0.9) # 0.9 for label smoothing
            d_loss_fake = discriminator.train_on_batch(gen_imgs, np.zeros((half_batch, 1)))
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)


            # ---------------------
            #  Train Generator AND Clssifier
            # ---------------------

            noise = np.random.normal(0, 1, (batch_size, 100))

            # the generator wants the discriminator to label the generated samples
            # as valid (ones)
            valid_y = np.array([1] * 32)

            # train the generator
            g_loss = combined.train_on_batch(noise, np.ones((batch_size, 1)))
            C_loss= combined.train_on_batch(noise, np.ones((batch_size, 1)))

            # Plot the progress
            print ("%d [D loss: %f] [G loss: %f][C loss: %f,acc.: %.2f%%]" % (epoch,d_loss[0],g_loss,C_loss,100*(1-C_loss)))
            D_L_REAL.append(d_loss_real)
            D_L_FAKE.append(d_loss_fake)
            D_L.append(d_loss)
            D_ACC.append(d_loss[1])
            G_L.append(g_loss)
            C_L.append(C_loss)

            # if at save interval => save generated image samples
            if epoch % save_interval == 0:
                save_imgs(dataset_title, epoch)
```

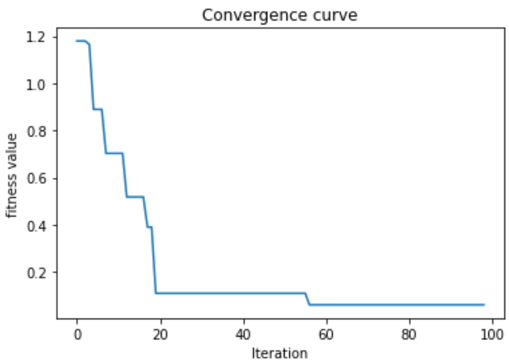
```
import math
import numpy as np

x=0
max_iter=100
fitt=[]

PS=0.2                # Stallions Percentage
PC=0.13               # Crossover Percentage
NStallion=PS*nPop     # number Stallion
Nfoal=nPop-NStallion  # number foal

#create Group
# pos=[]
cost=[]
group=Nfoal
k=1
group_pos=np.zeros((nPop,d))
groupcost=np.zeros((nPop,d))

def create_group():
 for k in range(int(10)):
   group_pos[k]=lb+(1-d)*random.random()
   group_pos=group_pos[k]*(ub-lb)
   groupcost[k]=ub+(1-d)*random.random()
Stallion=pos

#Select Stallion
def N_Stallion():
 for k in range (NStallion):
    Stallion(k).pos=lb+(1,d)*random.random()*(ub-lb)
    Stallion(k).cost=ub+(Stallion(k)*pos)
    ngroup=len(group)
    a=random.random(ngroup)
    group=group(a)


while x < max_iter-1:
  TDR=1-x*((1)/max_iter)  #Calculate TDR
  i=1;j=1;P=0
  R1=random.random()
  R2=(1-d)*random.random()
```

```
    idx=(P==0)
    Z=(1-d)*random.random()<TDR
    R3=R1*idx+R2*~idx
    for j in range(nPop):
      if random.random()>PC:
        rr=-2+4*R3
        Stallion[i,:]= 2*R3*math.cos(2*math.pi*rr)*Stallion[i,:]*(Stallion[i,:]*pos[i,:]-Stallion[i,:]*group*pos[i,:])+(Stallion[i,:]*pos[i
      else:
        A=(NStallion)*random.random()
        a=1
        c=2
        x1=Stallion[c,:]*group*pos
        x2=Stallion[a,:]*group*pos
        y1=(x1+x2)/2   # Crossover

      R=(-2-2)*random.random()+1  # R is random number[-2,2]
      WH=1
      if (j<Nfoal):
          j=j+1
      else:
          WH=1

      if (R3>0.5):
          Stallion=2*Z*math.cos(2*math.pi*R*Z)*(WH-Stallion)+WH
      else:
          Stallion=2*Z*math.cos(2*math.pi*R*Z)*(WH-Stallion)-WH
      if (i<NStallion):
          i=i+1
          j=1
      else:
          new_fit=fitness(pos[x,:])
    if (new_fit<best_fit):  # if minimization problem use '<'
          best_fit=new_fit
          best_pos=pos[x,:]
    print('Iteration - ',str(x+1),': Best Position',str(best_pos),': Best Fitness',str("%.6f"%best_fit))
    fitt.append(best_fit)
    x=x+1

  fitt=np.array(fitt)
  print("\nBest solution found:\n")
  print('Best fitness :',best_fit)
  print('Best position :',best_pos)
```

```
Iteration -  1 : Best Position [0.86383782 0.31548556] : Best Fitness 1.179323
Iteration -  2 : Best Position [0.86383782 0.31548556] : Best Fitness 1.179323
Iteration -  3 : Best Position [0.86383782 0.31548556] : Best Fitness 1.179323
Iteration -  4 : Best Position [0.48471247 0.68013139] : Best Fitness 1.164844
Iteration -  5 : Best Position [0.34259824 0.54695609] : Best Fitness 0.889554
Iteration -  6 : Best Position [0.34259824 0.54695609] : Best Fitness 0.889554
Iteration -  7 : Best Position [0.34259824 0.54695609] : Best Fitness 0.889554
Iteration -  8 : Best Position [0.36106257 0.34194185] : Best Fitness 0.703004
Iteration -  9 : Best Position [0.36106257 0.34194185] : Best Fitness 0.703004
Iteration -  10 : Best Position [0.36106257 0.34194185] : Best Fitness 0.703004
Iteration -  11 : Best Position [0.36106257 0.34194185] : Best Fitness 0.703004
Iteration -  12 : Best Position [0.36106257 0.34194185] : Best Fitness 0.703004
Iteration -  13 : Best Position [0.43119403 0.08678578] : Best Fitness 0.517980
Iteration -  14 : Best Position [0.43119403 0.08678578] : Best Fitness 0.517980
Iteration -  15 : Best Position [0.43119403 0.08678578] : Best Fitness 0.517980
Iteration -  16 : Best Position [0.43119403 0.08678578] : Best Fitness 0.517980
Iteration -  17 : Best Position [0.43119403 0.08678578] : Best Fitness 0.517980
Iteration -  18 : Best Position [0.3364429 0.0538168] : Best Fitness 0.390260
Iteration -  19 : Best Position [0.3364429 0.0538168] : Best Fitness 0.390260
Iteration -  20 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  21 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  22 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  23 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  24 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  25 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  26 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  27 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  28 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  29 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  30 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  31 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  32 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  33 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  34 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  35 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  36 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  37 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  38 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  39 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  40 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  41 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  42 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  43 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
```

```
Iteration -  44 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  45 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  46 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  47 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  48 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  49 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  50 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  51 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  52 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  53 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  54 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  55 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  56 : Best Position [0.03392045 0.07540011] : Best Fitness 0.109321
Iteration -  57 : Best Position [0.04071689 0.0198735 ] : Best Fitness 0.060590
Iteration -  58 : Best Position [0.04071689 0.0198735 ] : Best Fitness 0.060590
```

```
plt.plot(fitt)
plt.xlabel('Iteration')
plt.ylabel('fitness value')
plt.title('Convergence curve')
plt.show()
```



Updation of weights on cap_triple gan using optimization

```
z = Input(shape=(100,))
img = generator(z)
discriminator.trainable = False
valid = discriminator(img)
cl=classifer(valid)
opt_combined = Model(z,cl)
print('COMBINED:')
opt_combined.summary()
opt_combined.compile(loss='binary_crossentropy', optimizer=Adam(0.145031, 0.5))
```

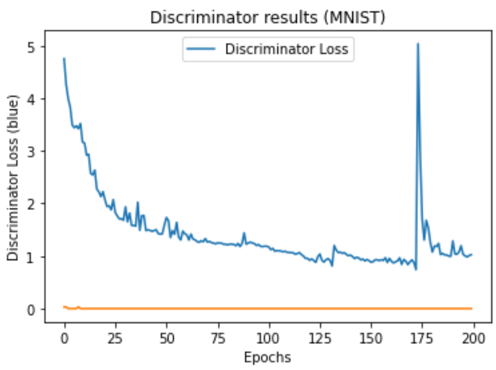
```
COMBINED:
Model: "model_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_10 (InputLayer)       [(None, 100)]             0

 model_6 (Functional)        (None, 28, 28, 1)         856705

 model_5 (Functional)        (None, 100)               6898820

 model_7 (Functional)        (None, 1)                 22776

=================================================================
Total params: 7,778,301
Trainable params: 878,841
Non-trainable params: 6,899,460
_________________________________________________________________
```

```
opt_history = train('mnist', epochs=100, batch_size=32, save_interval=50)
```

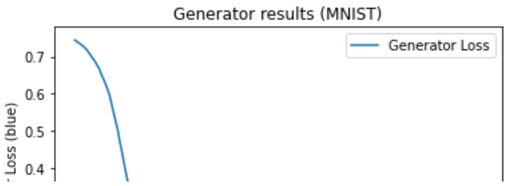
```
0 [D loss: 1.173042] [G loss: 0.046131][C loss: 0.046060,acc.: 95.39%]
1 [D loss: 1.120710] [G loss: 0.046096][C loss: 0.046018,acc.: 95.40%]
2 [D loss: 1.144038] [G loss: 0.046336][C loss: 0.046248,acc.: 95.38%]
3 [D loss: 1.093052] [G loss: 0.046444][C loss: 0.046343,acc.: 95.37%]
4 [D loss: 1.103481] [G loss: 0.046429][C loss: 0.046317,acc.: 95.37%]
5 [D loss: 1.100568] [G loss: 0.046396][C loss: 0.046268,acc.: 95.37%]
6 [D loss: 1.095693] [G loss: 0.046329][C loss: 0.046208,acc.: 95.38%]
7 [D loss: 1.078386] [G loss: 0.046218][C loss: 0.046092,acc.: 95.39%]
8 [D loss: 1.090196] [G loss: 0.046199][C loss: 0.046062,acc.: 95.39%]
9 [D loss: 1.070992] [G loss: 0.046176][C loss: 0.046025,acc.: 95.40%]
10 [D loss: 1.067618] [G loss: 0.045963][C loss: 0.045821,acc.: 95.42%]
```

```
11 [D loss: 1.067017] [G loss: 0.045979][C loss: 0.045812,acc.: 95.42%]
12 [D loss: 1.060284] [G loss: 0.045903][C loss: 0.045720,acc.: 95.43%]
13 [D loss: 1.034272] [G loss: 0.045790][C loss: 0.045620,acc.: 95.44%]
14 [D loss: 1.046444] [G loss: 0.045832][C loss: 0.045621,acc.: 95.44%]
15 [D loss: 1.065213] [G loss: 0.045774][C loss: 0.045528,acc.: 95.45%]
16 [D loss: 1.026112] [G loss: 0.045873][C loss: 0.045585,acc.: 95.44%]
17 [D loss: 0.998381] [G loss: 0.045818][C loss: 0.045503,acc.: 95.45%]
18 [D loss: 0.954988] [G loss: 0.045732][C loss: 0.045434,acc.: 95.46%]
19 [D loss: 0.955223] [G loss: 0.045325][C loss: 0.045014,acc.: 95.50%]
20 [D loss: 0.917663] [G loss: 0.045322][C loss: 0.044952,acc.: 95.50%]
21 [D loss: 0.944984] [G loss: 0.045475][C loss: 0.045065,acc.: 95.49%]
22 [D loss: 0.914441] [G loss: 0.046010][C loss: 0.045487,acc.: 95.45%]
23 [D loss: 0.878798] [G loss: 0.049837][C loss: 0.047753,acc.: 95.22%]
24 [D loss: 0.984111] [G loss: 0.049296][C loss: 0.046942,acc.: 95.31%]
25 [D loss: 1.037951] [G loss: 0.047127][C loss: 0.045548,acc.: 95.45%]
26 [D loss: 0.919117] [G loss: 0.047935][C loss: 0.046100,acc.: 95.39%]
27 [D loss: 0.885861] [G loss: 0.046237][C loss: 0.044862,acc.: 95.51%]
28 [D loss: 0.928973] [G loss: 0.046368][C loss: 0.044666,acc.: 95.53%]
29 [D loss: 0.950459] [G loss: 0.044516][C loss: 0.043595,acc.: 95.64%]
30 [D loss: 0.915460] [G loss: 0.044333][C loss: 0.043354,acc.: 95.66%]
31 [D loss: 0.811855] [G loss: 0.064612][C loss: 0.046256,acc.: 95.37%]
32 [D loss: 1.199010] [G loss: 0.055209][C loss: 0.046832,acc.: 95.32%]
33 [D loss: 1.112162] [G loss: 0.046863][C loss: 0.044664,acc.: 95.53%]
34 [D loss: 1.065502] [G loss: 0.044494][C loss: 0.043629,acc.: 95.64%]
35 [D loss: 1.072695] [G loss: 0.043475][C loss: 0.042833,acc.: 95.72%]
36 [D loss: 1.051716] [G loss: 0.042779][C loss: 0.042441,acc.: 95.76%]
37 [D loss: 1.063133] [G loss: 0.043483][C loss: 0.042413,acc.: 95.76%]
38 [D loss: 1.028140] [G loss: 0.042526][C loss: 0.042367,acc.: 95.76%]
39 [D loss: 1.005933] [G loss: 0.042312][C loss: 0.042126,acc.: 95.79%]
40 [D loss: 1.017147] [G loss: 0.042109][C loss: 0.041964,acc.: 95.80%]
41 [D loss: 0.987532] [G loss: 0.041787][C loss: 0.041681,acc.: 95.83%]
42 [D loss: 0.950978] [G loss: 0.041757][C loss: 0.041619,acc.: 95.84%]
43 [D loss: 0.975854] [G loss: 0.041820][C loss: 0.041629,acc.: 95.84%]
44 [D loss: 0.963667] [G loss: 0.041777][C loss: 0.041578,acc.: 95.84%]
45 [D loss: 0.925505] [G loss: 0.041555][C loss: 0.041401,acc.: 95.86%]
46 [D loss: 0.941696] [G loss: 0.041618][C loss: 0.041445,acc.: 95.86%]
47 [D loss: 0.906585] [G loss: 0.041515][C loss: 0.041332,acc.: 95.87%]
48 [D loss: 0.934160] [G loss: 0.041458][C loss: 0.041217,acc.: 95.88%]
49 [D loss: 0.912132] [G loss: 0.041395][C loss: 0.041082,acc.: 95.89%]
50 [D loss: 0.881726] [G loss: 0.041775][C loss: 0.041293,acc.: 95.87%]
51 [D loss: 0.890788] [G loss: 0.041511][C loss: 0.041122,acc.: 95.89%]
52 [D loss: 0.927253] [G loss: 0.040956][C loss: 0.040750,acc.: 95.92%]
53 [D loss: 0.928749] [G loss: 0.041281][C loss: 0.040853,acc.: 95.91%]
54 [D loss: 0.912042] [G loss: 0.040846][C loss: 0.040596,acc.: 95.94%]
55 [D loss: 0.929390] [G loss: 0.041212][C loss: 0.040744,acc.: 95.93%]
56 [D loss: 0.915143] [G loss: 0.041093][C loss: 0.040762,acc.: 95.92%]
57 [D loss: 0.967413] [G loss: 0.041055][C loss: 0.040741,acc.: 95.93%]
```

```
plt.plot(D_L)
plt.title('Discriminator results (MNIST)')
plt.xlabel('Epochs')
plt.ylabel('Discriminator Loss (blue)')
plt.legend(['Discriminator Loss'])
plt.show()
```




```
plt.plot(G_L)
plt.title('Generator results (MNIST)')
plt.xlabel('Epochs')
plt.ylabel('Generator Loss (blue)')
plt.legend(['Generator Loss'])
plt.show()
```




```
plt.plot(C_L)
plt.title('classifer results (MNIST)')
plt.xlabel('Epochs')
plt.ylabel('classifer Loss (blue)')
plt.legend(['classifer  Loss'])
plt.show()
```