Shirin Mohebbi 9935597

Neural Networks and Deep Learning

Homework #4: Deep ConvNet and GAN

Part A:

In this part, we will learn a deep ConvNet to classify COVID and Non-COVID images. in order to do that we use the pre trained network named VGG16 from keras library. VGG-16 is a convolutional neural network that is 16 layers deep. The input to cov1 layer is of fixed size 224 x 224 RGB image. The image is passed through a stack of convolutional (conv.) layers, where the filters were used with a very small receptive field: 3×3. The convolution stride is fixed to 1 pixel; the spatial padding of conv. layer input is such that the spatial resolution is preserved after convolution. the padding is 1-pixel for 3×3 conv. layers. Spatial pooling is carried out by five max-pooling layers, which follow some of the conv. layers. Max-pooling is performed over a 2×2 pixel window, with stride 2. The architecture depicted below is VGG16

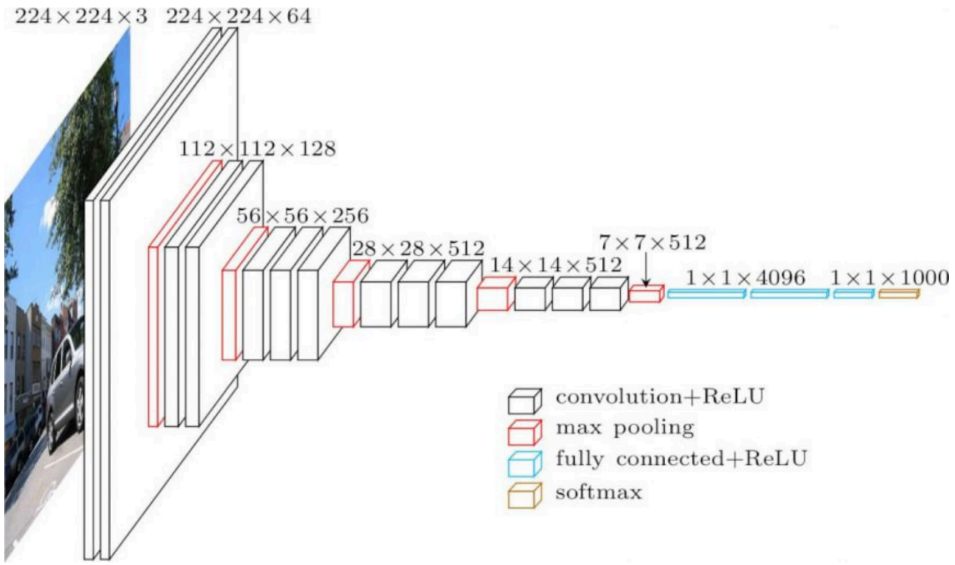


Figure 1. VGG16 Architecture

first we load images based on our target size (=256), convert Covid train and non-Covid train images to array via Keras pre processing *image.img_to_array* function. and do the same for test images.

```
img = image.load_img(+path_img, target_size=(size_img, size_img), color_mode='grayscale')

x = image.img_to_array(img)
```

then we define VGG model but we set its include top to False so we could customize this model for our use.

```
model_vgg16_conv = VGG16(weights='imagenet', include_top=False)
```

we define our input based on image size (=256) give this input to defined VGG model and get the output, now we should add our layer to the end of this model with functional API. and then we define our model.

```
input = Input(shape=(size_img, size_img, 3),name = 'image_input')
output_vgg16_conv = model_vgg16_conv(input)
x = Flatten(name='flatten')(output_vgg16_conv)
x = Dense(4096, activation='relu', name='fc1')(x)
x = Dense(4096, activation='relu', name='fc2')(x)
x = Dense(2, activation='softmax', name='predictions')(x)
mlp = Model(inputs = input, outputs = x)
```

we can see our final model with this instruction

```
mlp.summary()
```

then we compile our network and fit train data on it and evaluate test data on these metrics.

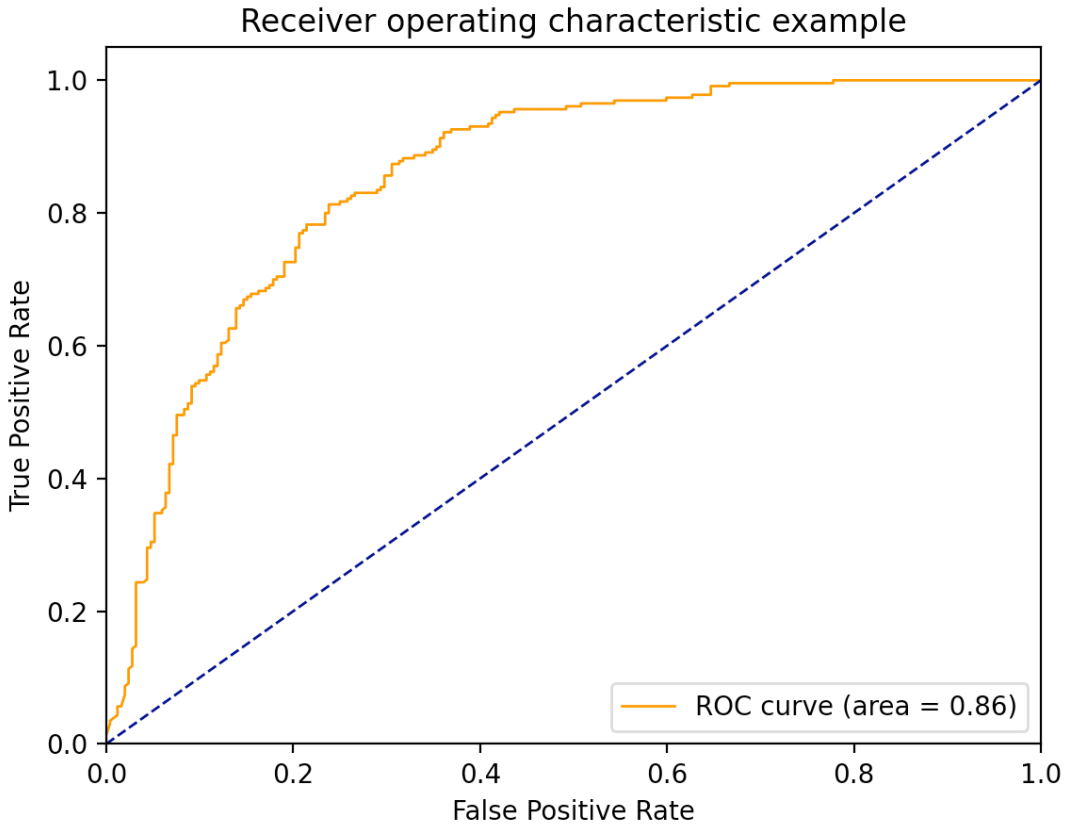
```
mlp.compile(loss="categorical_crossentropy", optimizer="adam", metrics=['acc',f1_score_m,precision_m, recall_m])
```

```
mlp.fit(xTrain, yTrain, batch_size=batch_size, epochs=epochs,
validation_split=0.1)
loss, accuracy, f1_score, precision, recall = mlp.evaluate(xTest,
yTest, verbose=0)
```

Results:

```
2021-07-29 16:05:52.300511: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:176] None of the MLIR Optimization Passes are enabled (registered 2)
Epoch 1/5
1/9 [==>.....] - ETA: 14:46 - loss: 7.2239 - acc: 0.4500 - f1_score_m: 0.4500 - precision_m: 0.4500 - recall_m
2/9 [=====>.....] - ETA: 11:22 - loss: 699.5535 - acc: 0.4750 - f1_score_m: 0.4750 - precision_m: 0.4750 - recall
3/9 [=========>.....] - ETA: 9:09 - loss: 819.1530 - acc: 0.4922 - f1_score_m: 0.4922 - precision_m: 0.4922 - recall_
4/9 [============>.....] - ETA: 7:10 - loss: 813.9709 - acc: 0.5045 - f1_score_m: 0.5045 - precision_m: 0.5045 - recall_
5/9 [================>.....] - ETA: 5:32 - loss: 779.3333 - acc: 0.5104 - f1_score_m: 0.5104 - precision_m: 0.5104 - recall_
6/9 [====================>.....] - ETA: 4:05 - loss: 738.6177 - acc: 0.5124 - f1_score_m: 0.5124 - precision_m: 0.5124 - recall_
7/9 [======================>.....] - ETA: 2:40 - loss: 698.6469 - acc: 0.5143 - f1_score_m: 0.5143 - precision_m: 0.5143 - recall_
8/9 [========================>....] - ETA: 1:19 - loss: 661.5172 - acc: 0.5162 - f1_score_m: 0.5162 - precision_m: 0.5162 - recall_
9/9 [==============================] - 835s 91s/step - loss: 600.6848 - acc: 0.5193 - f1_score_m: 0.5193 - precision_m: 0.5193 - recall_m: 0.5193 - val_loss
: 0.2913 - val_acc: 0.9000 - val_f1_score_m: 0.9000 - val_precision_m: 0.9000 - val_recall_m: 0.9000
Epoch 2/5
9/9 [==============================] - 765s 88s/step - loss: 0.5486 - acc: 0.7936 - f1_score_m: 0.7936 - precision_m: 0.7936 - recall_m: 0.7936 - val_loss:
0.2671 - val_acc: 0.8850 - val_f1_score_m: 0.8850 - val_precision_m: 0.8850 - val_recall_m: 0.8850
Epoch 3/5
9/9 [==============================] - 976s 113s/step - loss: 0.1255 - acc: 0.9519 - f1_score_m: 0.9519 - precision_m: 0.9519 - recall_m: 0.9519 - val_loss:
 0.3167 - val_acc: 0.8900 - val_f1_score_m: 0.8900 - val_precision_m: 0.8900 - val_recall_m: 0.8900
Epoch 4/5
9/9 [==============================] - 817s 91s/step - loss: 0.0353 - acc: 0.9939 - f1_score_m: 0.9939 - precision_m: 0.9939 - recall_m: 0.9939 - val_loss:
0.0604 - val_acc: 0.9850 - val_f1_score_m: 0.9850 - val_precision_m: 0.9850 - val_recall_m: 0.9850
Epoch 5/5
9/9 [==============================] - 996s 109s/step - loss: 0.0113 - acc: 0.9981 - f1_score_m: 0.9981 - precision_m: 0.9981 - recall_m: 0.9981 - val_loss:
 0.1426 - val_acc: 0.9600 - val_f1_score_m: 0.9600 - val_precision_m: 0.9600 - val_recall_m: 0.9600
```

```
test results:
loss: 1.4424784183502197 accuracy: 0.7759336233139038 f1 score: 0.7890624403953552 precision: 0.7890
625 recall: 0.7890625
```



Part 2:

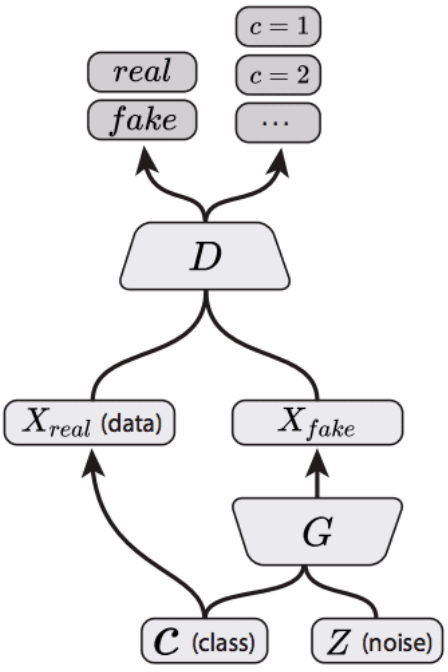
In this part is we generate images and predicting their classes. for generating image we use AC-GAN network. ac-gan is an architecture for training generative models. this type of GAN involves the conditional generation of images by a generator model. Image generation can be conditional on a class label, if available, allowing the targeted generated of images of a given type. ac gan consist of two parts Generator Model and Discriminator Model.

Generator Model:

- **Input**: Random point from the latent space, and the class label.
- **Output**: Generated image.

Discriminator Model:

- **Input**: Image.
- **Output**: Probability that the provided image is real, probability of the image belonging to each known class.



AC-GAN
(Present Work)

first we read samples and convert them to array as we did in part1. then we should define our models.

Build_discriminator

The discriminator model take as input an image and predict both the probability of being real/fake of the image and the probability of the image belonging to each of the classes. we define Sequential model. we add our layers

```
model = Sequential()
model.add(Conv2D(16, kernel_size=3, strides=2,
input_shape=self.img_shape, padding="same"))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.25))
model.add(Conv2D(32, kernel_size=3, strides=2, padding="same"))
model.add(ZeroPadding2D(padding=((0, 1), (0, 1))))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.25))
model.add(BatchNormalization(momentum=0.8))
model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.25))
model.add(BatchNormalization(momentum=0.8))
model.add(Conv2D(128, kernel_size=3, strides=1, padding="same"))
model.add(LeakyReLU(alpha=0.2))
model.add(Dropout(0.25))
model.add(Flatten())
model.summary()
```

Now we should set inputs and outputs of our model, input is our image, and we have two outputs. real/fake and class label. the first is a single node with the sigmoid activation for predicting the realness of the image. the second is a single nodes for class label, using the

softmax activation function to predict the class label of the given image.

```
img = Input(shape=self.img_shape)
features = model(img)
validity = Dense(1, activation="sigmoid")(features)
label = Dense(self.num_classes, activation="softmax")(features)
return Model(img, [validity, label])
```

Build_generator

for generator we set our layer

```
model = Sequential()
model.add(Dense(128 * x * x, activation="relu",
input_dim=self.latent_dim))
model.add(Reshape((x, x, 128)))
model.add(BatchNormalization(momentum=0.8))
model.add(UpSampling2D())
model.add(Conv2D(128, kernel_size=3, padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(momentum=0.8))
model.add(UpSampling2D())
model.add(Conv2D(64, kernel_size=3, padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(momentum=0.8))
model.add(Conv2D(self.channels, kernel_size=3, padding='same'))
model.add(Activation("tanh"))
model.summary()
```

generator has two input, noise from latent space and label. and model builds an image for output,

```
noise = Input(shape=(self.latent_dim,))
label = Input(shape=(1,), dtype='int32')
label_embedding = Flatten()(Embedding(self.num_classes,
self.latent_dim)(label))
model_input = multiply([noise, label_embedding])
img = model(model_input)
return Model([noise, label], img)
```

ACGAN

for AC-GAN we should combine these two models. for losses of discriminator model we use sparse_categorical_crossentropy for label prediction and binary_crossentropy for real/fake prediction.

```
losses = ['binary_crossentropy', 'sparse_categorical_crossentropy']
self.discriminator = self.build_discriminator()
self.discriminator.compile(loss=losses, optimizer='adam',
metrics=['accuracy'])
```

The generator takes noise and the target label as input and generates the corresponding label

```
self.generator = self.build_generator()
noise = Input(shape=(self.latent_dim,))
label = Input(shape=(1,))
img = self.generator([noise, label])
```

For the combined model we will only train the generator

```
self.discriminator.trainable = False
```

The discriminator takes generated image as input and determines validity and the label of that image

```
valid, target_label = self.discriminator(img)
```

The combined(generator and discriminator) model Trains the generator to fool the discriminator. The model is fit using the Adam version of stochastic gradient descent with a small learning rate and modest momentum.

```
self.combined = Model([noise, label], [valid, target_label])
self.combined.compile(loss=losses, optimizer='adam')
```

Train

For training first we read samples as we did in part1, then we convert it between 0 and 1, in each epoch we select n (number of batch size) random data from our training samples, we generate noises of latent size and random labels for generator inputs and call generator to generate images from these noises.
for training discriminator, we call discriminator on real selected datas and fake images generated by generator. loss of discriminator is average of these two.

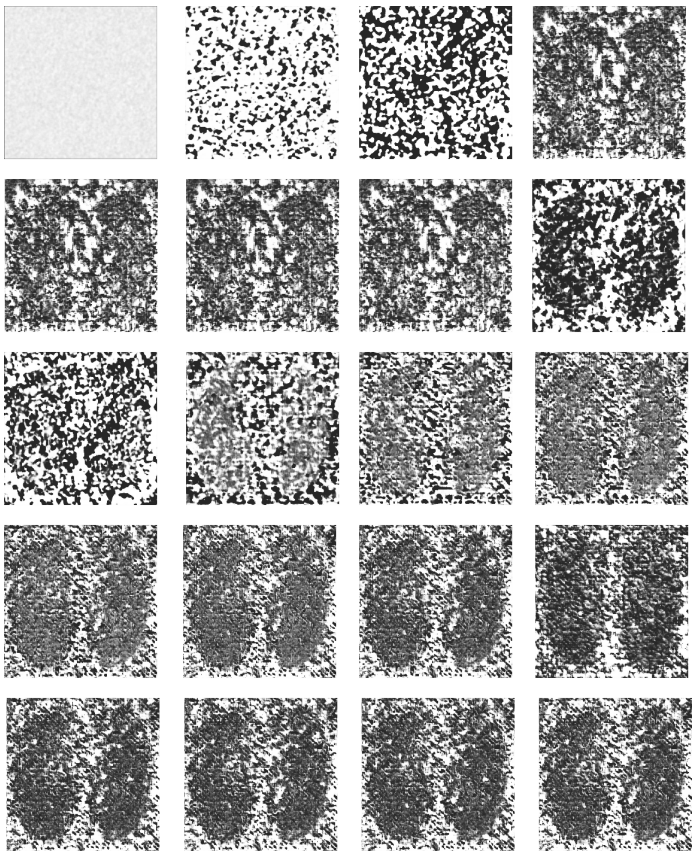
```
d_loss_real = self.discriminator.train_on_batch(
    imgs, [valid, img_labels])
d_loss_fake = self.discriminator.train_on_batch(
    gen_imgs, [fake, sampled_labels])
d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)
```

Then we train the generator

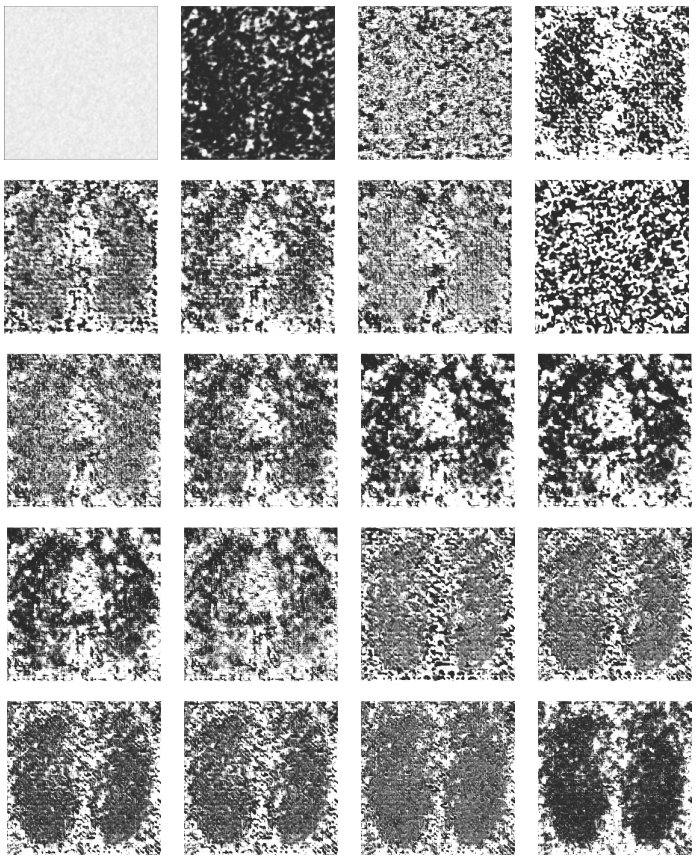
```
g_loss = self.combined.train_on_batch([noise, sampled_labels],
[valid, sampled_labels])
```

Results:

process of generating covid images:



process of generating non-covid images:



gan training errors

```
1497 [D loss: 0.033402, acc.: 100.00%, op_acc: 99.00%] [G loss: 0.008170]
1498 [D loss: 0.017852, acc.: 100.00%, op_acc: 99.50%] [G loss: 0.016086]
1499 [D loss: 0.057296, acc.: 100.00%, op_acc: 97.50%] [G loss: 0.012182]
```

generated images with gan classification metrics:

```
test gan results:
loss: 3.154000759124756 accuracy: 0.6731234788894653 f1 score: 0.6711790561676025
precision: 0.6703414916992188 recall: 0.6720467209815979
```

gan ROC and AUC