CMPE258-Lab 2 step 1 report

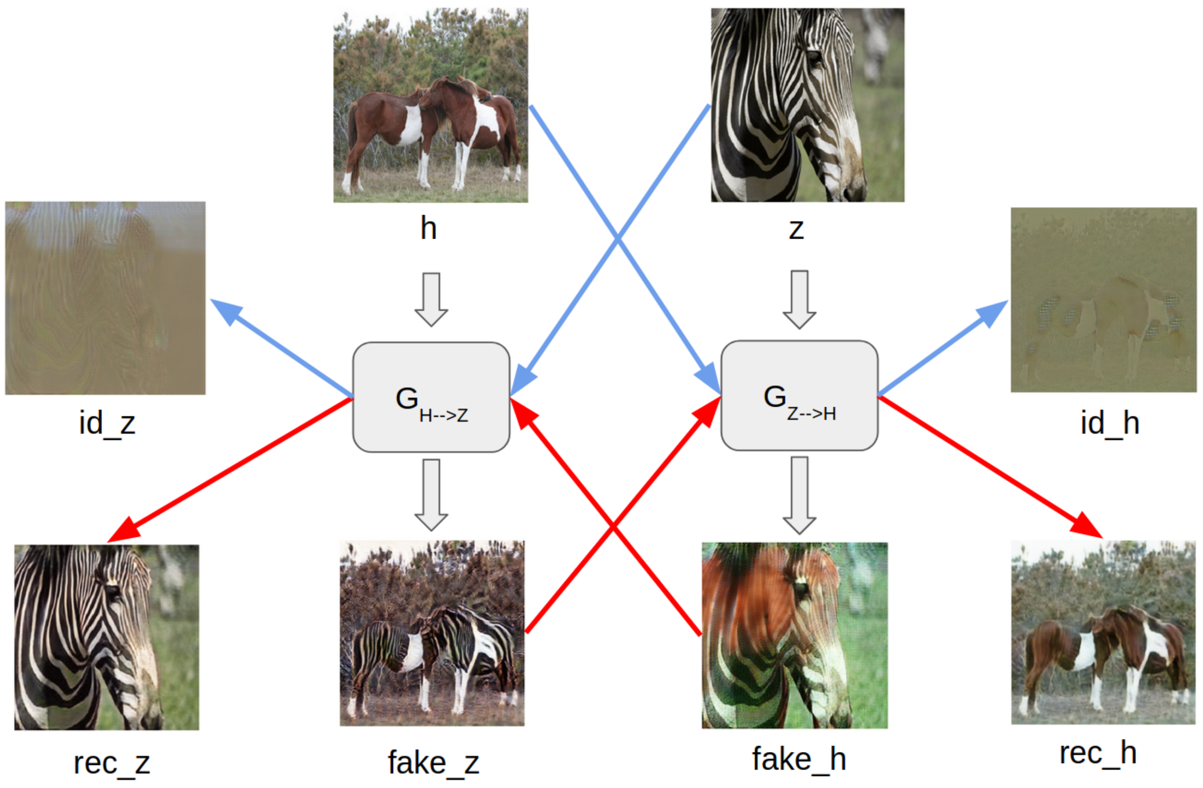
Introduction:

For step 1 of Lab 2, I have implemented CycleGAN to translate characteristics of Apple images into Orange images using generator and discriminator. CycleGAN can generate images of different domains without a one-to-one relationship with the other domain.

Step 1 and 2:

CycleGan Architecture:

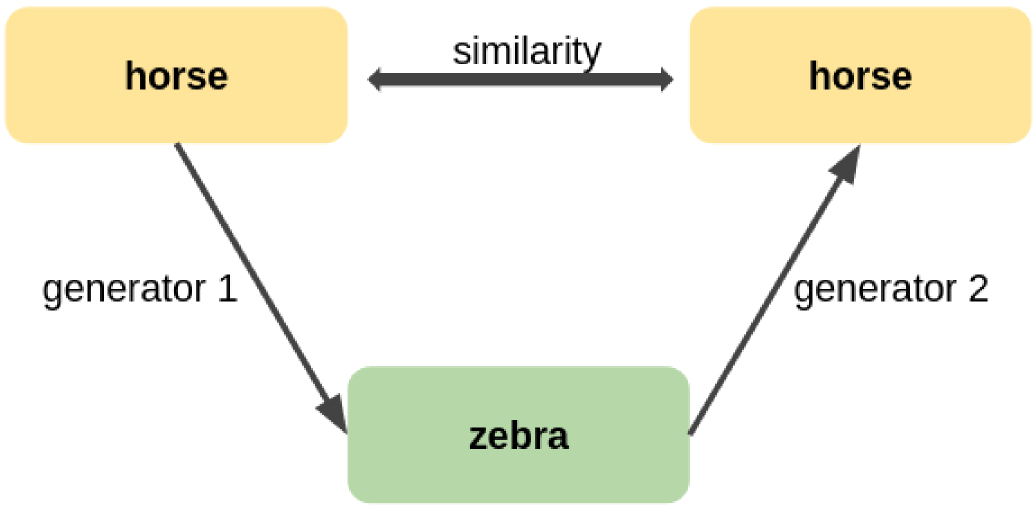
I've used **Apple2orange** dataset but let's try to understand the concept using an architectural example for horse and zebra.



GANs (Generative Adversarial Networks) are made up of two neural networks: a generator and a discriminator. A CycleGAN is made up of two GANs, totaling two generators and two discriminators.

One generator transforms horses into zebras while the other transforms zebras into horses when given two sets of different images, such as horses and zebras. The discriminators are present throughout the training phase to determine if images generated by generators appear authentic or false. With the feedback of their respective discriminators, generators can improve through this process. A generator in CycleGAN receives additional feedback from the other generator. This feedback ensures that an image formed by a generator is cycle consistent, which means that using both generators on the same image should result in a similar image.

Design Process and Loss/object function:



In the above figure, we can see that if we generate a zebra image using generator 1 and use that generated image to generate a horse, we should receive the same image.

For Apple2Orange example:
G{A->O} is the generator that transforms apple images into orange images and G{O->A} is the generator that transforms orange images into apple images.

Let’s try to understand the process for a given (a,o) example:

1. From a, G{A->O), generates fake_o a fake orange image. From o, G{A->O} generates ident_o which should be identical to o.

2. From o, G{O->A), generates fake_a a fake orange image. From a, G{O->A} generates id_a which should be identical to a.

3. From fake_o, G{O->A), generates rec_a which should be similar to that of a.

4. From fake_a, G{A->O), generates rec_o which should be similar to that of o.

All these 4 processes make sure that the generated image that we get is of the other domain.

Since we need to update the weights after every iteration, we use the following losses:

$$MSE(D_z(fake_z), 1) = \frac{1}{30}\sum_{i,j}^{30}(D_z(fake_z)(i,j) - 1)^2$$

Mean Squared Error (MSE) for the first generated image fake_o (in the equation fake_z). D(fake_o)) is 1 if the image looks similar to fake_o looks similar to real orange.

Similarly, we have MSE(rec_h,h) which is the loss function of G{A->O} and G{O->A} for a real apple image and recreated apple image. This loss is very important as we get a sense of how accurate both generators are.

We update the weights of discriminators by minimizing the above losses.

GAN utilization:

In order to get the best of GAN utilizations, I did data augmentation to avoid overfitting. The following techniques were performed:

1. Random mirroring: horizontal flipping of images
2. Random Jitter: Since both the generators take images of size 256x256, I had to randomly crop images from the dataset.

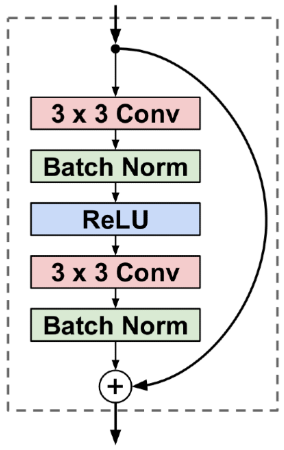
We import and reuse the Pix2Pix models which have the generator and discriminator models.

Some of the differences are observed in CycleGAN compared to GANs:

1. Instance Normalization instead of BatchNormalization:
   The batch size in every iteration is 1 hence we can't use BatchNormalization

2. Unet based generator instead of resnet based generator:
   Used Unet based generator for simplicity

Generator Architecture:



Layer	Activation Size
Input	3x256x256
64x7x7 conv, stride 1, pad 3	64x256x256
128x3x3 conv, stride 2, pad 1	128x128x128
256x3x3 conv, stride 2, pad 1	256x64x64
9 consecutive Residual Blocks	256x64x64
128x3x3 convTranspose, stride 2, pad 1, out_pad 1	128x128x128
64x3x3 convTranspose, stride 2, pad 1, out_pad 1	64x256x256
3x7x7 conv, stride 1, pad 3	3x256x256

Generators take a 256x256 image as input, down sample it, and then up sample it back to 256x256 to produce the generated image.

Discriminator Architecture:

Layer	Activation Size
Input	3x256x256
64x4x4 conv, stride 2, pad 1	64x128x128
128x4x4 conv, stride 2, pad 1	128x64x64
256x4x4 conv, stride 2, pad 1	256x32x32
512x4x4 conv, stride 1, pad 1	512x31x31
1x4x4 conv, stride 1, pad 1	1x30x30

Discriminator takes as input an image of size 256x256 and outputs a tensor of size 30x30.

Issues or challenges you faced during training and/or data selection:

Following issues were faced during training the model:

1. **Training Time:** Since each training epoch processes images in a single batch and each training step goes through multiple generators, discriminators and loss functions, it took around 12-13 minutes to train each epoch on GPU. To overcome this, I had to train the model on google colab pro.

2. **Saving Images**: I tried plt.savefig method to save images earlier. It was giving me a padding error as well as pixel noise while saving the images. I used keras inbuilt save_img method to save images.

3. **Checkpoint Save:** The checkpoint was initially being saved in the current runtime instance. As soon as the runtime expired, the checkpoints got erased. I had to change the instance path to a google drive location.

4. **Less Testing Images:** All the tensorflow dataset had very less amount of testing data. For step 3, it was very important to get at least 1500 photos for the CNN model to train efficiently. I had to predict some training images using generator and save them to the drive.

<u>Step 3:</u>

In step 3, I used the images saved in step 2 and developed a CNN model to differentiate between the original images and the predicted images.

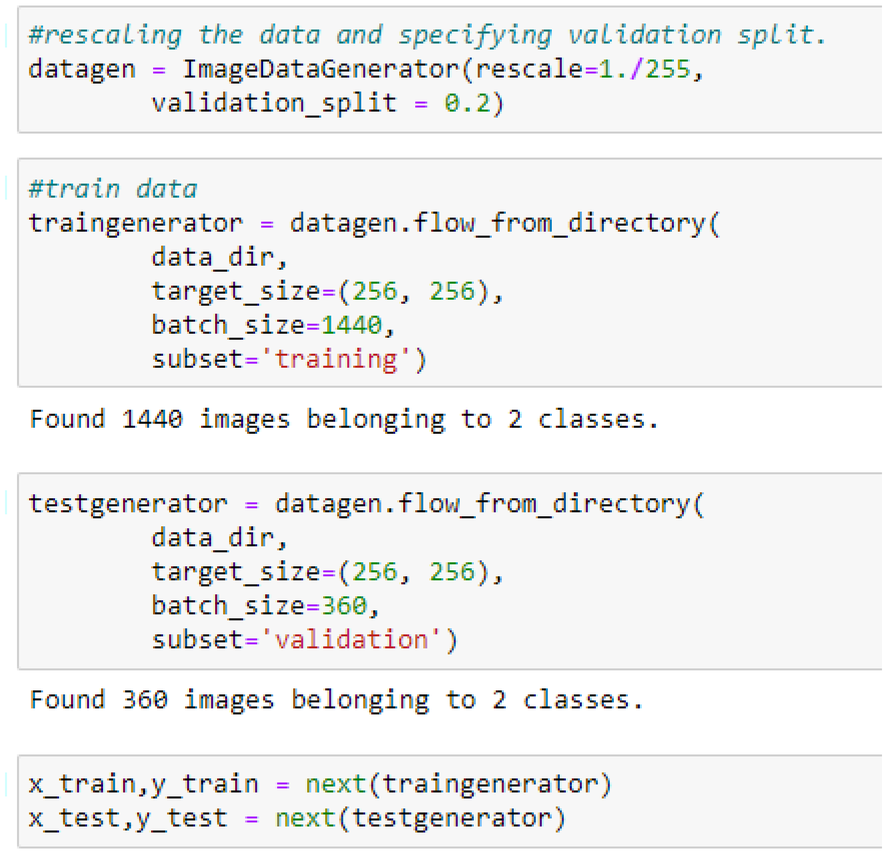
Process:

1. **Load Data:**

I loaded the data using ImageDataGenerator from keras. The images generated from step 2 were saved in a folder called ‘predicted’ and the original images were saved in a folder called ‘origins’. Using ImageDataGen’s flow_from_directory, the images were loaded and converted into tensors.

2. **Splitting the Tensors:**

While initializing an ImageDataGen object, we can specify a validation split and then define two objects. One being the training data and the other being the validation data. x_train, y_train and x_test,y_test were generated using train data and validation data.



```
#rescaling the data and specifying validation split.
datagen = ImageDataGenerator(rescale=1./255,
        validation_split = 0.2)
```

```
#train data
traingenerator = datagen.flow_from_directory(
        data_dir,
        target_size=(256, 256),
        batch_size=1440,
        subset='training')
```

```
Found 1440 images belonging to 2 classes.
```

```
testgenerator = datagen.flow_from_directory(
        data_dir,
        target_size=(256, 256),
        batch_size=360,
        subset='validation')
```

```
Found 360 images belonging to 2 classes.
```

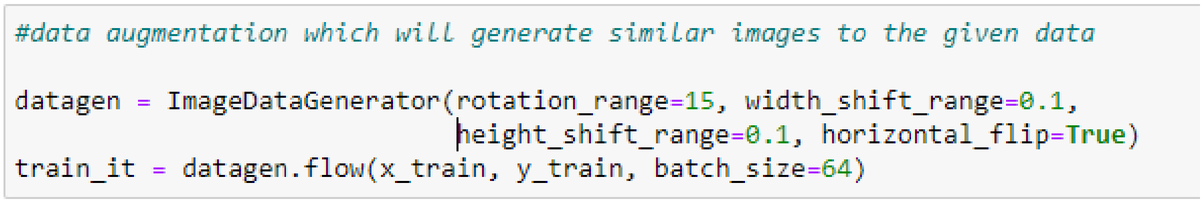
```
x_train,y_train = next(traingenerator)
x_test,y_test = next(testgenerator)
```

3. Preprocessing Data:

Data was rescaled between [0,1]

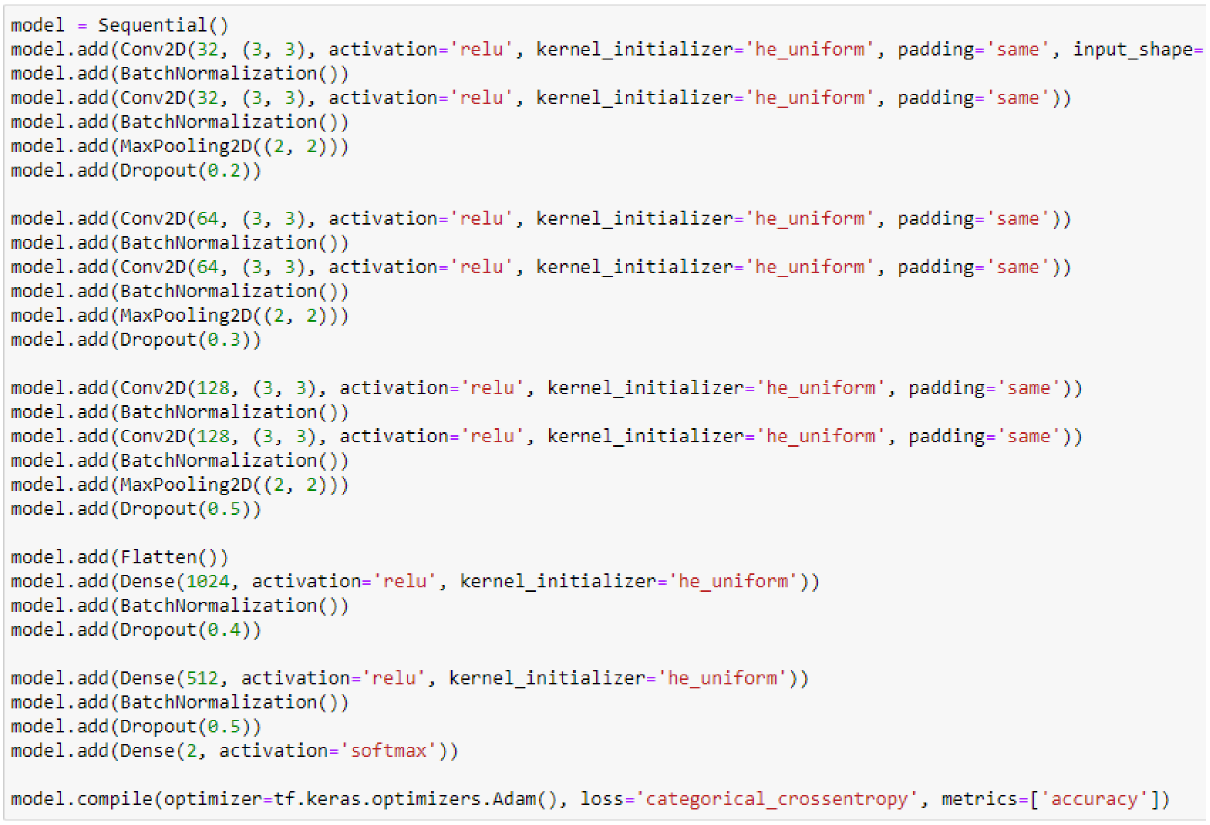
4. Data Augmentation:

Since we have very less data, data augmentation techniques would generate more similar data. The following data augmentation techniques were applied.



```
#data augmentation which will generate similar images to the given data

datagen = ImageDataGenerator(rotation_range=15, width_shift_range=0.1,
                             height_shift_range=0.1, horizontal_flip=True)
train_it = datagen.flow(x_train, y_train, batch_size=64)
```

5. Model Architecture:
A CNN based model with 6 Convolution layers and 1 dense layer were used. Multiple Dropout and batch Normalization layers were used to avoid overfitting and normalize output after every layer.



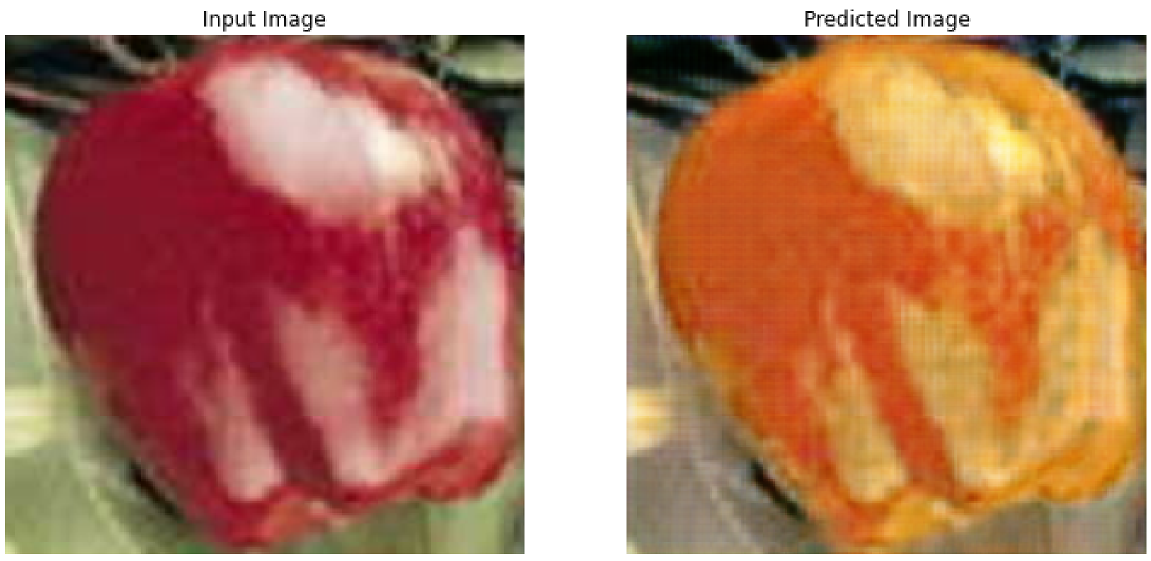
```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=
model.add(BatchNormalization())
model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.2))

model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.3))

model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
model.add(BatchNormalization())
model.add(MaxPooling2D((2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(1024, activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dropout(0.4))

model.add(Dense(512, activation='relu', kernel_initializer='he_uniform'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(2, activation='softmax'))

model.compile(optimizer=tf.keras.optimizers.Adam(), loss='categorical_crossentropy', metrics=['accuracy'])
```

6. Model Optimizations:

In order for a model to work efficiently, it is very important that we reach the global minimum. Current combination of optimizers ensures that. I used the Adam optimizer and categorical cross entropy. Early stopping was used to avoid overfitting and restore best weights which gave me the lease validation loss.

7. Accuracy:

The model achieved an accuracy score of 91.67% which means that the images generated using cycleGAN were not close to the real images. This differentiation is visible to the human eye as well.



It is clearly visible that the predicted image has just changed the color of the original apple image. Oranges do not look like the predicted image.

8. Challenges:

- Using OpenCV for loading images and saving into arrays was crashing the Ram again and again because the images have a shape of 256x256. I used the inbuilt ImageDataGenerator by tensorflow for loading images to arrays. This was done efficiently as google colab is tensorflow optimized.
- Getting x_train and x_test from traingenerator and testgenerator took a lot of time because converting around 1440 images with 256x256 shape into the memory would naturally take a lot of time.
- Optimizing the model architecture was another task. Getting the right amount of layers, dropouts, BatchNormalization layers was a challenging task.

Conclusion:

CycleGan's is a great technique to generate images of different domains without a one to one relation with that domain. However, it is not that efficient currently as the CNN model was clearly able to differentiate between original and fake orange images. With better optimization techniques in the future maybe CycleGANs can become more efficient.