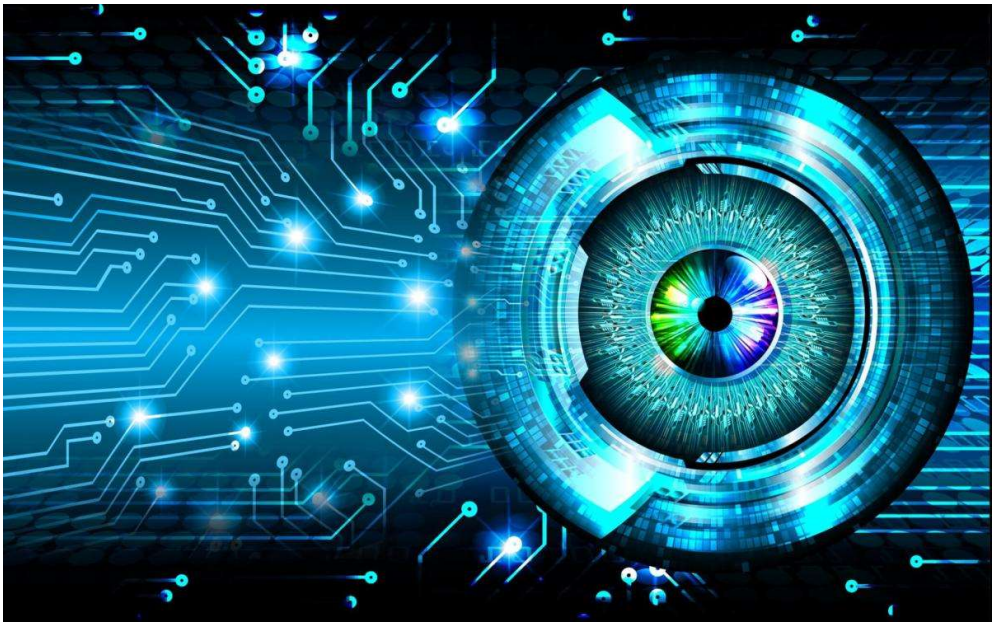


REMOVAL PREDICTION FILLER

Final Project Report

ABSTRACT

This report will describe in detail the most important finding from my Final Project. This will also include the process in which I achieve completing what I intended to do since the Project Proposals. We will also go through the attempts of creating the best Deep Learning model for our use case.

Student: Robles Herrera, Salvador

Course: Computer Vision (CS 4363)

Professor: Dr. Olac Fuentes

Fall 2022

Date: 12/07/2022

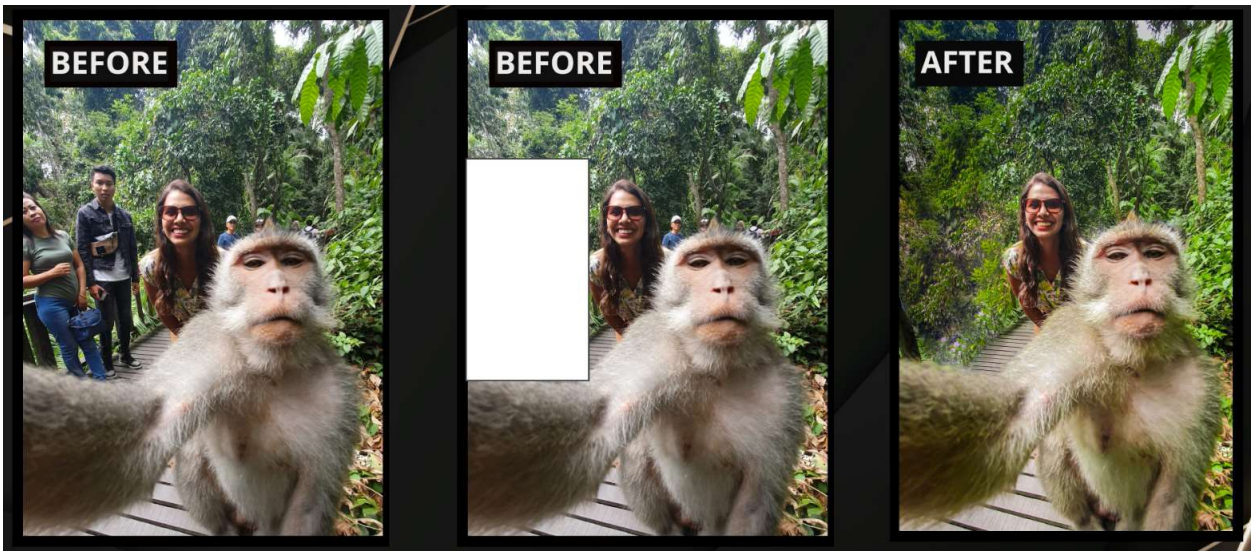
Introduction

Computer Vision enables the learning of knowledge from images, which will enable us to take actions with this information to solve a certain problem or task. In this semester we went over different topic over the wide field of Computer Vision. This final project will showcase some of the concepts we learnt in class and how it can be applied in real world application.

The idea is to create some sort of Removal Filter. This is known as Image Inpainting, which is a "task of reconstructing missing regions in an image", which is helpful for image restoration, image-based rendering. Another thing that Image Inpainting can do and the idea for this project is for object removal. For example, in an image you may have certain object you want to delete from the background, then you would select a box containing that undesired object and then that way you can create a model that predicts the missing part of the image.

An example of what I wanted to achieve is something like the following, where you have an image as an input and then you place a prediction box where you want to learn to predict missing parts of an image. What I want for my project is for the predicted image to contain a clean background, a predicted hole in the image that will look very similar to the original but without the objects that were not desired in the image.

An example of what I wanted to achieve is something like the following, where you have an image as an input and then you place a prediction filler of the removed image part.



Initial Approach Ideas

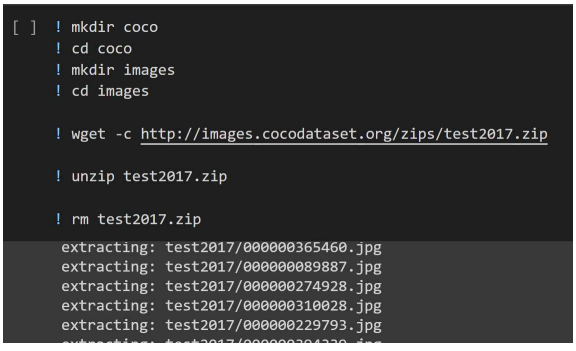
This is a very complex problem for traditional Machine Learning approaches like KNN or Support Vector Machines. Notice that the number of outputs will be the number of pixels in the area to be deleted times the number of color channels in the image. This is a big number of outputs required, therefore I opt for using a Deep Learning model that will learn from examples where an image is missing pixels and the actual original image.

Using a regular Neural Network can work and this is one of the initial approaches I did. There are also other more sophisticated ways to approach this problem, for example using **Convolutional** Neural Networks and **Autoencoders**. I used both of this approaches and checked what results and what images looked better after predicting the output images.

I used Colab as my environment for my code due to its simplicity and efficiency, as well as allowing us to use GPUs to make computations faster and run our Deep Leaning models.

Project Details

In this project I used the Common Objects in Context (COCO) which is a large-scale object detection dataset. Luckily for this project any type of digital image works as our input, this since we are not looking for anything for our dataset, just images. COCO allows us to only download the images, unlike other datasets where data segmentation data or other types of data is included. COCO has different collection of images; I used the testing dataset from 2017 and this is how you can download the 40 thousand images:



Some example of images in the COCO dataset are like the following:

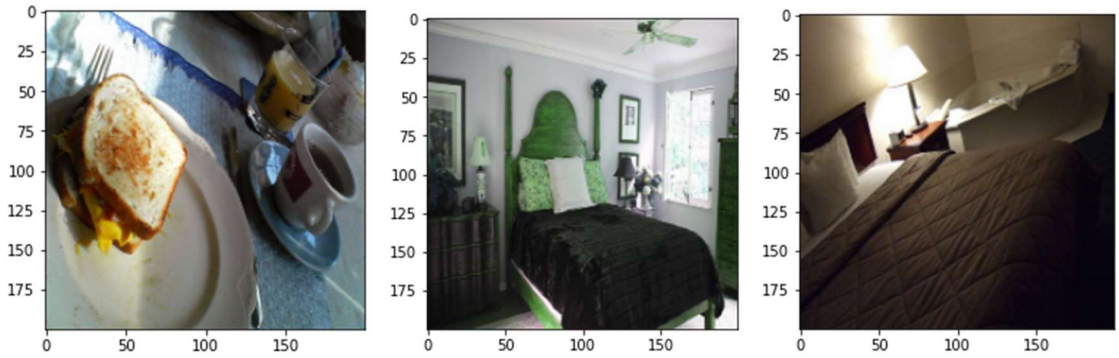
These images are very diverse, which is a good thing but at the same time there are multiple shapes and image sizes, so there will be some pre-processing required in order to get all the images to a specific size for Deep Learning to work.

Set up and Pre-Processing

Our dataset consists of 40k images, with sizes around 400x600 or 600x400, Neural Networks require the same number of values per input. Using the resize function from the "transform" module from skimage I resized the images to 200x200 in order to be able to store this data into Numpy arrays. There are difficulties using Colab since it doesn't allow for the desired RAM so the session would crash if I used all the 40k images. Initially I used a subset of 1000 images and then increased it to 2000 once Colab Pro was used.

```
[ ]  directory = '/content/test2017'

     num_images = 2000 # Number of images that are going to be used from datasset
     length = 200 # Dimension 1 for image
     width = 200 # Dimensino 2 for image
     Y = np.zeros((num_images, length, width, 3)) # Our Y will be the original images

     # iterate over files in the specified directory
     i = 0
     for filename in os.listdir(directory):
         if i == num_images:
           break
         f = os.path.join(directory, filename)
         if i % 200 == 0: # See progress creating numpy array
           print('Number of images so far: ', i)

         img = plt.imread(directory + "/" + filename)
         Y[i] = resize(img, output_shape=(length, width, 3)) # Resizing image size to the specified
         i += 1
```

Examples of 200x200 resized images:

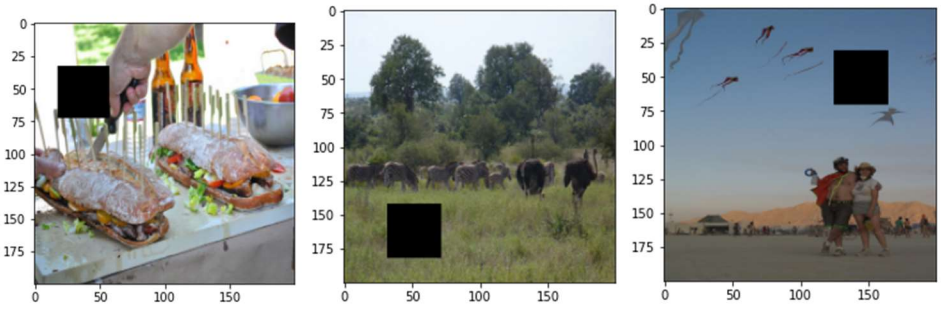


This images will now be our ground truth, which is our expected prediction, in other words our Y_Train and Y_test. Now we are required to create our X_Train and X_Test which will be our input where the images have a section that is missing. I would go through the images and create new ones where there is a dark square of sizes 40x40, the location of this square is randomized in the image. Also note that the square is the only 0.0 valued pixels, I changed any of them that were equal to 0.0 to something similar like 0.1, the pixels would still look black.

The following is the code I used to created random square holes in the images:

```
import random

def create_new_images_X(Y, square_size):
  X = np.zeros(Y.shape)
  for i in range(Y.shape[0]):
    img = np.copy(Y[i])
    img[img == 0.0] = 0.1 # Normalize data
    val_x = random.randint(0, length - square_size - 1)
    val_y = random.randint(0, width - square_size - 1)

    for k in range(val_x,val_x+square_size):
      for j in range(val_y, val_y+square_size):
        img[k][j] = 0.0
    X[i] = img

  return X

square_size = 40
X = create_new_images_X(Y, square_size)
```

Example of input images:



Until now we have two NumPy arrays with all the image information, now I split this set into training and testing sets. 90% of images will be used for training and 10% used for testing. On the training set our model would run and train parameters with the goal to reduces the loss of the error between images.

```
X_train, X_test, y_train, y_test = train_test_split(X, Y, random_state=0, train_size = .9)
```

Model Creation

Overall to create Deep Learning models I used this logic for all models. The first image is plotting the information for the training and the validation sets, representing how the values of the loss changing after each epoch. The second image contains an example of how a model is being compiled and fitted. To compile I use the same settings for all the models, which are the loss and metrics being set to Mean Squared Error, this makes sense to get difference between pixels and

see how far they are from each other, this loss takes the average of the difference between the actual value and the prediction. As for the optimizer, it is well known, at least for me, that Adam tends to perform great on the first epochs and then rapidly decline its performance. When the model is being fitted you need to specify the input and expected output which are X_Train and Y_Train.

One of the most important things that I learnt in this project was the **batch_size**, notice that it is set to 1 in the example below, in further models that the cost of training was higher the batch_size was incremented to 6. If you have a bigger number than this then you won't have good results, this since you don't want the images to be too correlated to each other. This size represents the number of examples/images that the network looks at in order to change a weight/parameter, in this case you don't want to generalize too much.

```
def plot_history(history):
  plt.ylabel('Loss')
  plt.xlabel('Epoch')
  plt.xticks(range(0, len(history['loss'] + 1)))
  plt.plot(history['loss'], label="training", marker='o')
  plt.plot(history['val_loss'], label="validation", marker='o')
  plt.legend()
  plt.show()
```

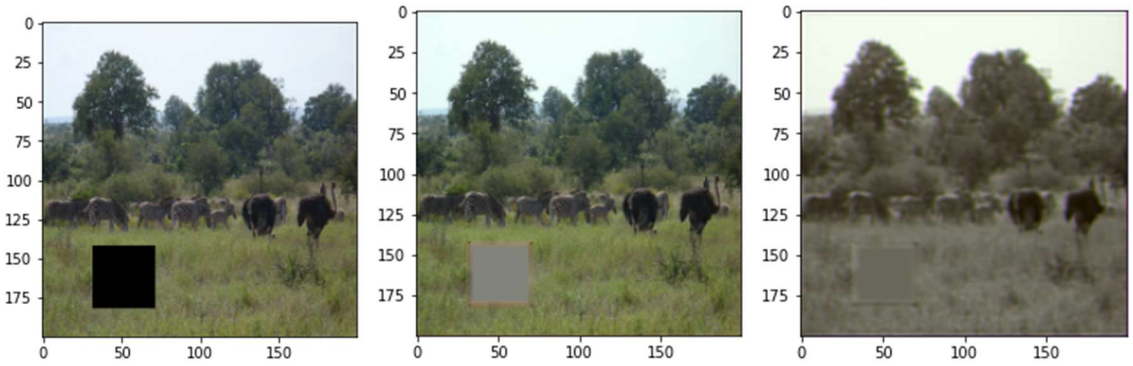
```
[ ] model =  cnn_model(input_shape = X_train.shape[1:])
    model.summary()
    model.compile(loss="mean_squared_error", metrics=["mean_squared_error"], optimizer = 'adam')

    history = model.fit(
      # X_train[:100], y_train[:100],
      X_train, y_train,
      epochs =  30, # Number of times the whole training set will be seen by the network
      batch_size = 1, # Number of training examples seen for every weight update
      verbose = 1,
      validation_data=(X_test, y_test),
    )

    history = pd.DataFrame(history.history)
    plot_history(history)
```

Model 1 (Neural Network)

My first Neural Network model wasn't a success. This first model was a very simple Neural Network with a few Relu layers. Unfortunately, this wasn't of my liking since it produced a predicted square which was of only one color. All images predicted the same square, that's why I

didn't keep the code and the running summary of the models I tried with a simple Neural Network approach. This are some of the results it created, on the left we have the input and on the right we have two images from two different variations of the model.

Results for Model 1:



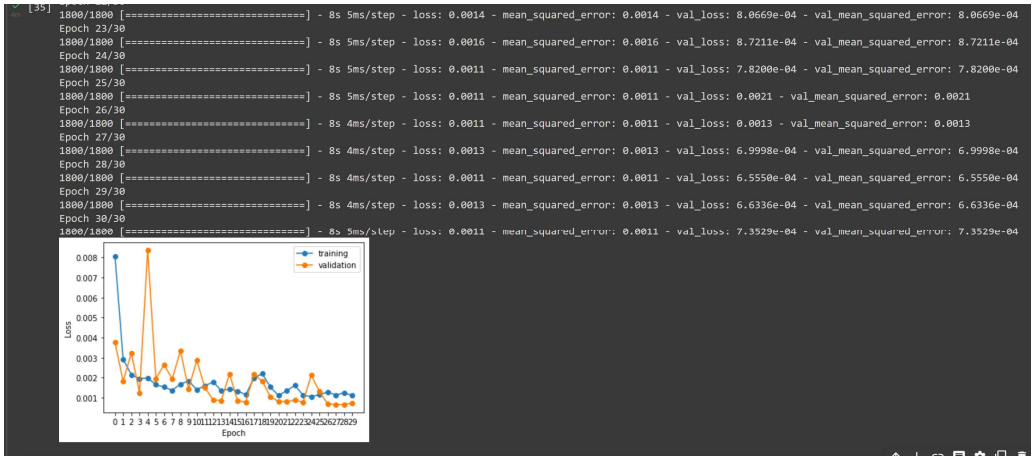
Model 2 (Convolutional Neural Network)

This model contains various Convolutional layers with the Relu activation function. In this model I played around with the parameters I could change, for example with the **kernel_size**, changing various layers from 3, 5 to 10. As the kernel increases I saw that the quality of the image seemed to go down, in terms of the color of the image and the blurriness overall. I also played around with the depth of the convolutional layer, going from 3, 4, 5, 10, and 30.

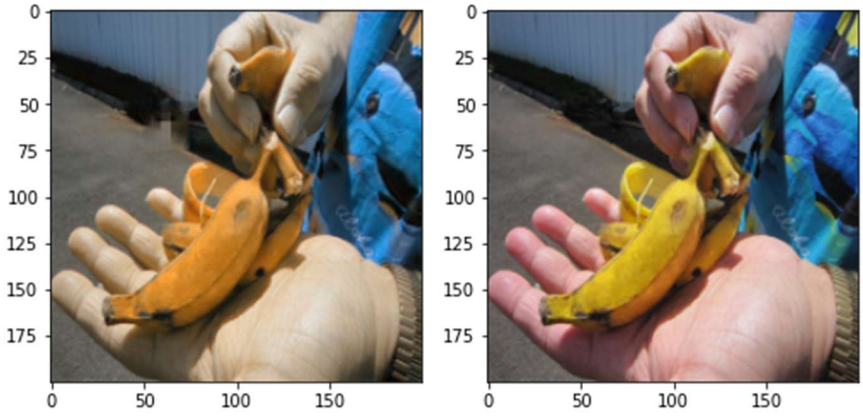
```
[33] def cnn_model(input_shape=(200,200,3),dropout=0.1):
       model = tf.keras.models.Sequential()
       model.add(Conv2D(30, kernel_size=(5, 5), input_shape=input_shape,  padding='same', activation="relu"))
       model.add(Conv2D(30, kernel_size=(5, 5), activation="relu", padding='same'))
       model.add(Conv2D(30, kernel_size=(5, 5), activation="relu", padding='same'))
       model.add(Conv2D(30, kernel_size=(5, 5), activation="relu", padding='same'))
       model.add(Conv2D(3, kernel_size=(5, 5), activation="relu", padding='same'))
       return model
```

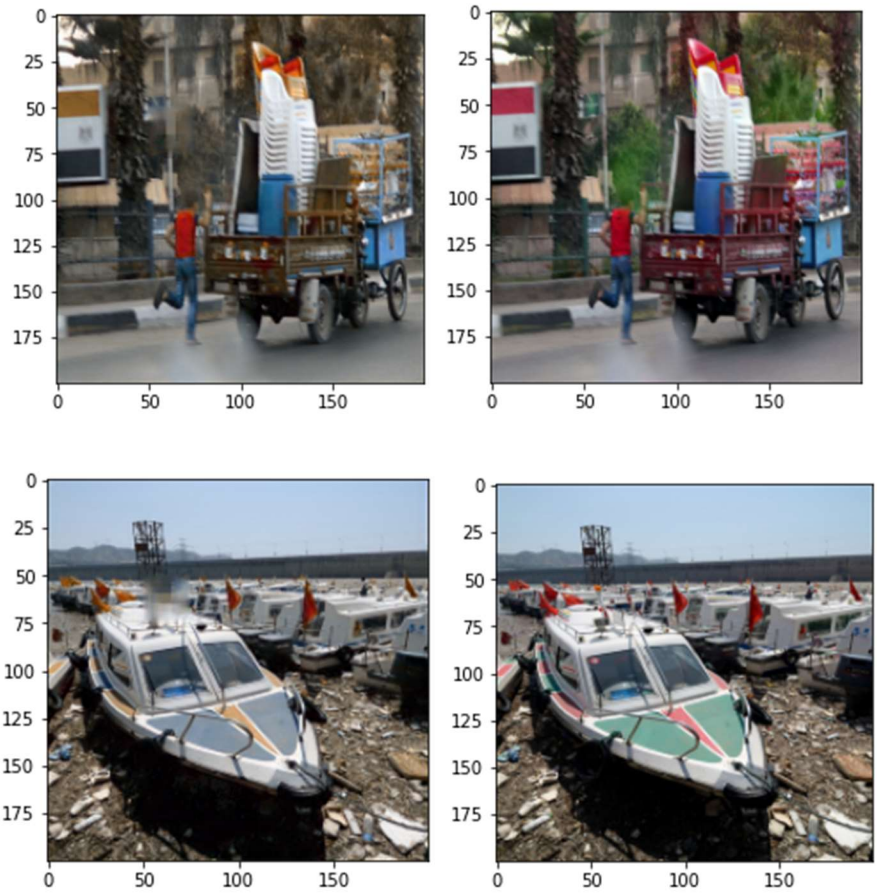
The following is the summary showing the number of parameters being used in the model, around 70k parameters. Also it shows a plot of how the training and validation loss changed through epoch, after 30 epochs it shows that the loss stops decreasing.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 200, 200, 30)      2280

 conv2d_1 (Conv2D)           (None, 200, 200, 30)      22530

 conv2d_2 (Conv2D)           (None, 200, 200, 30)      22530

 conv2d_3 (Conv2D)           (None, 200, 200, 30)      22530

 conv2d_4 (Conv2D)           (None, 200, 200, 3)       2253

=================================================================
Total params: 72,123
Trainable params: 72,123
Non-trainable params: 0
_________________________________________________________________
Epoch 1/30
```



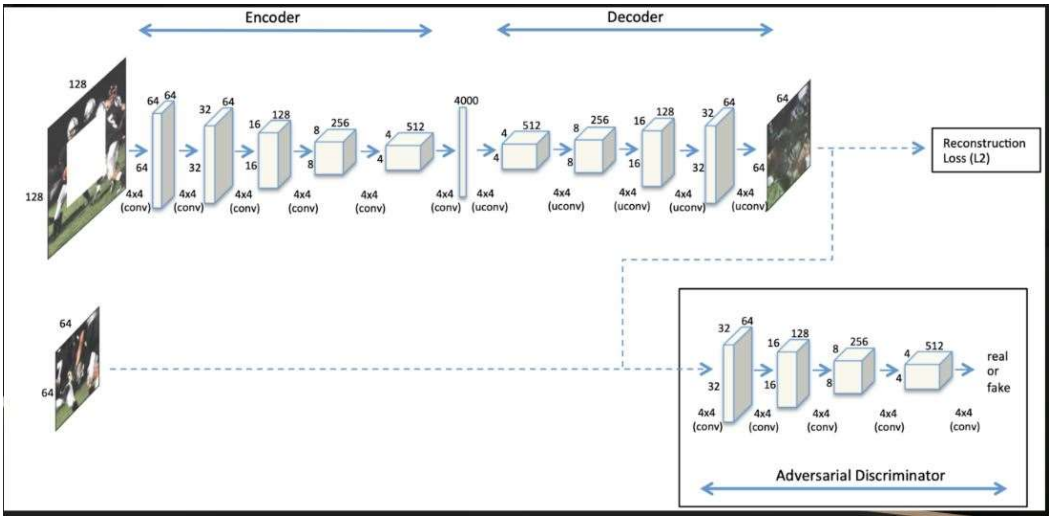
Results for Model 2:





Model 3 (Dense Autoencoder)

One of the things I tried is to use an Autoencoder approach. The idea behind this is to have a way to minimize the dimensions of the input like using several Convolutional layers to your initial image until you get a vector with a lot of values. Then, from this vector of values maximize the dimensions again into a desired shape, for example using De-convolution in order to finalize with a similar dimensional image from before.



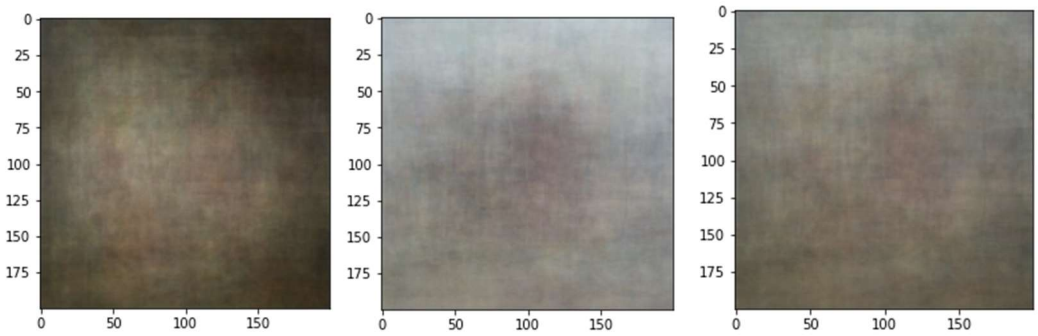
For this Model 3 I only used a dense layer which will produce a latent vector of 300 units. The problem with this approach is that for a dense layer to produce 300 units, by matrix multiplication it will need a matrix of 300 x (number of pixels to predict).

```
latent_dim = 300

class Autoencoder(Model):
  def __init__(self, latent_dim):
    super(Autoencoder, self).__init__()
    self.latent_dim = latent_dim
    self.encoder = tf.keras.Sequential([
      layers.Flatten(),
      layers.Dense(latent_dim, activation='relu'),
    ])
    self.decoder = tf.keras.Sequential([
      layers.Dense(length*width*3, activation='sigmoid'),
      layers.Reshape((length,width,3))
    ])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder = Autoencoder(latent_dim)
```

As you can see the number of parameters to train is extremely high, you would need millions of examples of images and a lot of epochs to train to have decent results. I couldn't produce good results due to this problems.

```
Model: "autoencoder"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_2 (Sequential)   (None, 600)               72000600

 sequential_3 (Sequential)   (None, 200, 200, 3)       72120000

=================================================================
Total params: 144,120,600
Trainable params: 144,120,600
Non-trainable params: 0
_________________________________________________________________
```

Results for Model 3: The images don't like quite like the originals.



Model 4 (Convolutional Autoencoder)

Now I will not a Dense Autoencoder but using the same idea this model will have convolutional layers on the encoder side and deconvolutional layers on the decoder side.

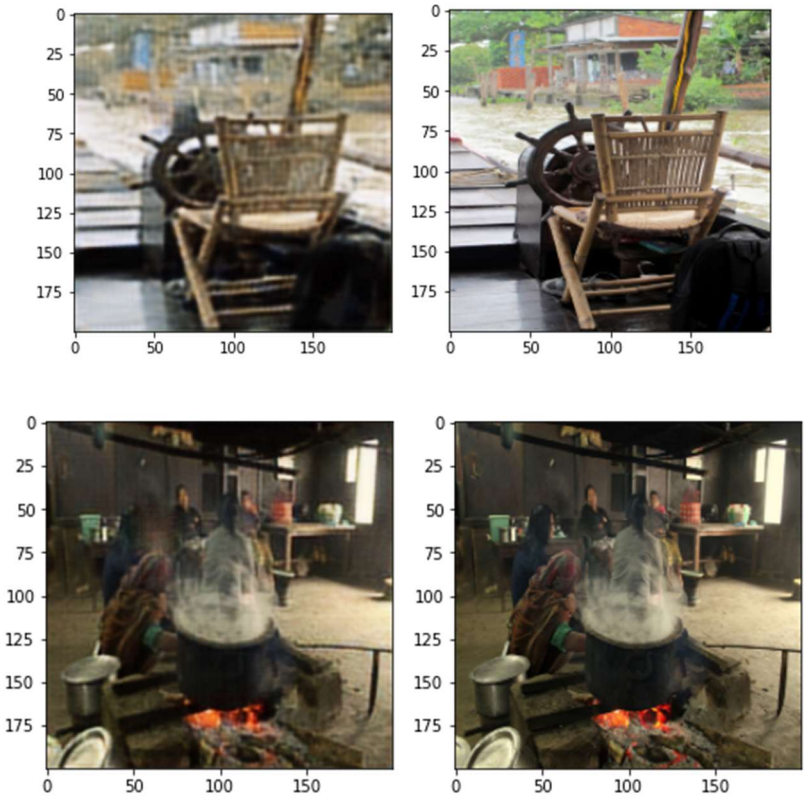
```
class Autoencoder_Conv(Model):
  def __init__(self):
    super(Autoencoder_Conv, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(200, 200, 3)),
      layers.Conv2D(3, kernel_size=(3, 3), activation='relu', padding='same', strides=2),
      layers.Conv2D(8, kernel_size=(3, 3), activation='relu', padding='same', strides=2),
      layers.Conv2D(32, kernel_size=(3, 3), activation="relu", padding='same')])

    self.decoder = tf.keras.Sequential([
      layers.Conv2DTranspose(32, kernel_size=(3, 3), activation="relu", padding='same'),
      layers.Conv2DTranspose(8, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2DTranspose(3, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2D(3, kernel_size=(5, 5), activation='sigmoid', padding='same')])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder_conv = Autoencoder_Conv()
```

The number of parameters is way lower than the previous model and it performs great taking into account the simplicity of the model.

```
Model: "autoencoder__conv"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_4 (Sequential)   (None, 50, 50, 32)        2644

 sequential_5 (Sequential)   (4, 200, 200, 3)          12007

=================================================================
Total params: 14,651
Trainable params: 14,651
Non-trainable params: 0
_________________________________________________________________
```

Results for model 4:

The color persistent is better but the image in itself looks blurrier than the normal convolutional model. In some of the images the color seems to be different the actual outputs.



Model 5 (2nd Convolutional Autoencoder)

This model is the same as Model 4 but adding multiple more Convolutional layers where the depth size is being increased by double. The depth of the layers go from 3 to 64, the kernel-size is the same with 3x3. Note that when adding convolutional an deconvolutional layers you can get errors that the output size is not the desired and it will throw an error. This is why there must be consistency between the Encoder and the Decoder.

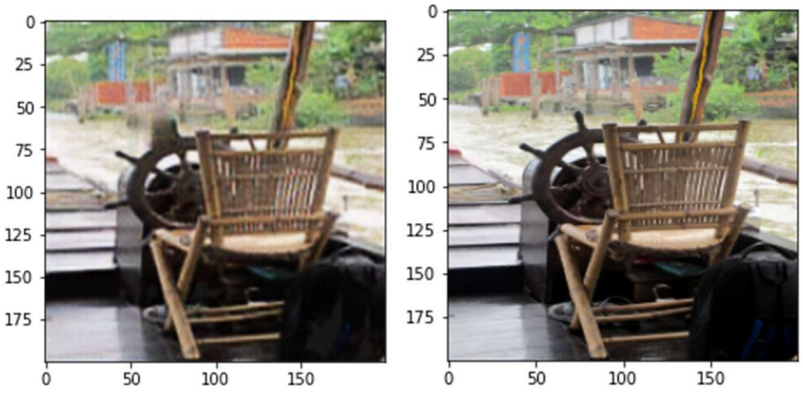
```
class Autoencoder_Conv_2(Model):
  def __init__(self):
    super(Autoencoder_Conv_2, self).__init__()
    self.encoder = tf.keras.Sequential([
      layers.Input(shape=(200, 200, 3)),
      layers.Conv2D(3, kernel_size=(3, 3), activation='relu', padding='same', strides = 2),
      layers.Conv2D(3, kernel_size=(3, 3), activation='relu', padding='same', strides = 2),
      layers.Conv2D(8, kernel_size=(3, 3), activation='relu', padding='same', strides = 2),
      layers.Conv2D(16, kernel_size=(3, 3), activation='relu', padding='same', strides = 2),
      layers.Conv2D(32, kernel_size=(3, 3), activation='relu', padding='same'),
      layers.Conv2D(64, kernel_size=(3, 3), activation='relu', padding='same')])

    self.decoder = tf.keras.Sequential([
      layers.Conv2DTranspose(64, kernel_size=(3, 3), activation="relu", padding='same'),
      layers.Conv2DTranspose(32, kernel_size=(3, 3), activation="relu", padding='same'),
      layers.Conv2DTranspose(16, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2DTranspose(8, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2DTranspose(3, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2DTranspose(3, kernel_size=(3,3), strides=2, activation='relu', padding='same'),
      layers.Conv2D(3, kernel_size=(3, 3), activation='sigmoid', padding='same')])

  def call(self, x):
    encoded = self.encoder(x)
    decoded = self.decoder(encoded)
    return decoded

autoencoder_conv_2 = Autoencoder_Conv_2()
```

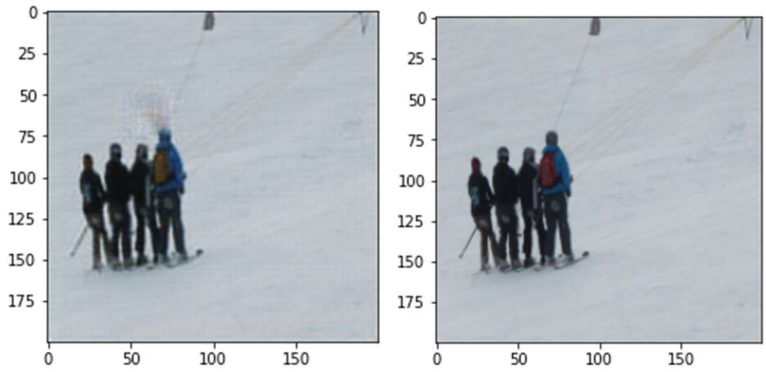
In comparison with the last Convolutional Autoencoder, this model has 50% more parameters to train. I also increased the number of epoch for this autoencoder approach to 150 epochs, this since each epoch doesn't take as much time as the other networks.

```
Model: "autoencoder__conv_2_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 sequential_66 (Sequential)  (None, 25, 25, 32)        6116

 sequential_67 (Sequential)  (None, 200, 200, 3)       15335

=================================================================
Total params: 21,451
Trainable params: 21,451
Non-trainable params: 0
_________________________________________________________________
```

Results for model 5

I think the color maintains more consistently than the other approaches, as well as that the blurriness is not that much in comparison with the last model.





Conclusion and Thoughts

The first model was not a success, but I saw that Convolution in general works very well and predicts and image that seems almost like if it was the original, dense layers are not successful in the environment I'm using and I'm guessing it will require a lot more computational power to be of better performance.

I think that for further implementation I am thinking on using a Generative Adversarial Networks approach. It seems like this kind of approach has been like by researchers since they are trying to implement something that would generate the predicted parts of an image and then let a discriminator model decide if it's one of the original images or not. Also, in the future there are a lot of good ideas that have been mentioned to me after my final presentation, for example the use of **Partial Convolution** as well as **Diffusion models**.

Overall, I'm happy with what I produced and I'm looking forward in the future to create more of this models using Computer Vision approaches to tackle interesting problems like this one.

References

Image Inpainting information:

https://paperswithcode.com/task/image-inpainting

COCO dataset:

https://cocodataset.org/#home

How to build Autoencoders:

https://blog.keras.io/building-autoencoders-in-keras.html

https://www.tensorflow.org/tutorials/generative/autoencoder

https://towardsdatascience.com/how-to-make-an-autoencoder-2f2d99cd5103

https://gaussian37.github.io/deep-learning-chollet-8-4/

Mean Squared Error as loss function:

https://keras.io/api/losses/regression_losses/#meansquarederror-class

Research papers about Image inpainting:

https://arxiv.org/pdf/1804.07723.pdf

https://arxiv.org/pdf/1811.11718.pdf