Fake or real:

machines can fool humans

Project for the Machine Learning Course

Alessio Ferrone 1751859

Jessica Frabotta 1758527

Summary

1. Introduction

Could you believe this person does not exist?

Deep learning made impressive improvements during those last years. In particular in the image processing field deep learning models, with their multi-level structures, are very helpful in extracting complicated information from input images. Models like Convolutional Neural Networks or Generative Models totally changed the way in which different kind of tasks are performed, from the classical image classification to Text-to-Image Translation, achieving results that were unimaginable before.

The creation of these types of fake images only became possible in recent years thanks to generative adversarial networks (GANs). GANs achieve this level of realism by pairing a generator, which learns to produce the target output, with a discriminator, which learns to distinguish true data from the output of the generator. The generator tries to fool the discriminator, and the discriminator tries to keep from being fooled.

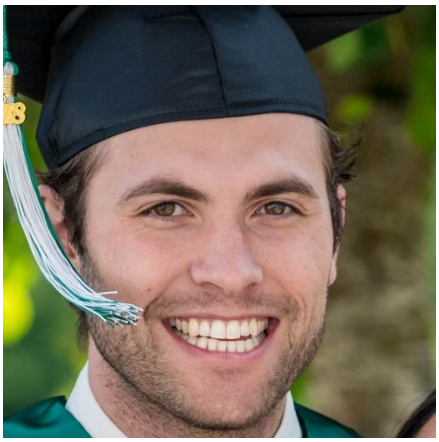
Images produced with deep learning are that realistic that there are now businesses that sell fake people. On the website Generated.Photos, you can buy a “unique, worry-free” fake person for $2.99, or 1,000 people for $1,000. If you just need a couple of fake people – for characters in a video game, or to make your company website appear more diverse – you can get their photos for free on ThisPersonDoesNotExist.com. Adjust their likeness as needed; make them old or young or the ethnicity of your choosing. If you want your fake person animated, a company called Rosebud.AI can do that and can even make them talk.

But can it become a problem for humans that machines generate face images that look so realistic?

Artificially generated media are already on the radar of the US government. The Pentagon has invested in research to detect deepfakes. Making a person appear to say or do something they did not has the potential to take the war of disinformation to a whole new level. Over the last few years, AI has armed malicious actors and bots with an invaluable weapon: the ability to appear alarmingly authentic. Unlike before, when trolls simply ripped real faces off the internet and anyone could unmask them by reverse-imaging their profile picture, it is practically impossible for someone to do the same for AI-generated photos because they are fresh and unique. And even upon closer inspection, most people cannot tell the difference. There are even reports of a spy that used an AI-generated face to connect with targets on LinkedIn.

So, it will be increasingly important to correctly identify fake faces. Machines, which are the creator of these very realistic fake faces that fool humans, are also the strongest weapon that humans have to correctly identify them.

The objective of this project is the implementation of a classifier which correctly distinguishes real and fake faces. A real time system was also developed. It takes in input images of real or artificially generated faces and detects and classifies them. We also implemented a Deep Convolutional Generative Adversarial Network (DCGAN) with which we generated some images of fake faces that were used to furtherly test the classifier.



REAL

FAKE

2. Ideas behind our project

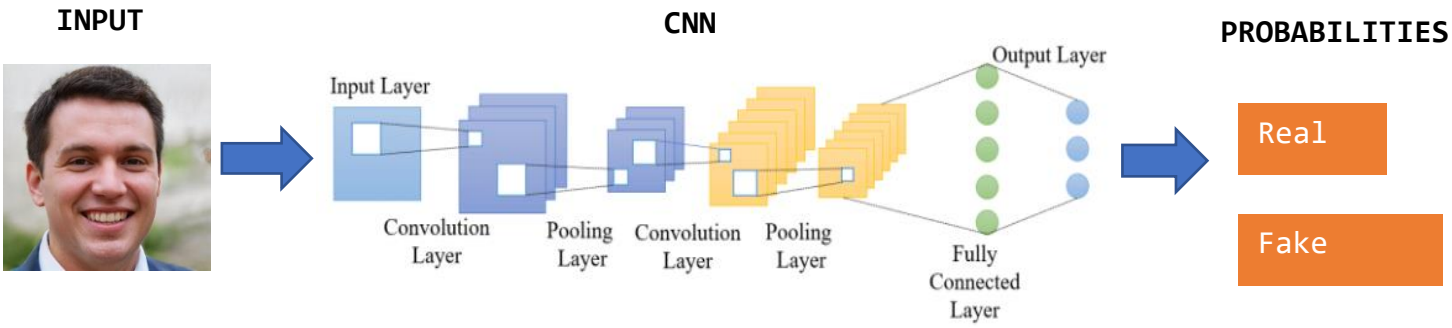
Our project is articulated in three main components:

1. A Convolutional Neural Network, used to solve the binary classification task;
2. A Deep Convolutional Generative Adversarial Network, used to produce some examples to validate the CNN model on additional unseen data;
3. A real-time system, which detects face images from a video and for each frame uses the CNN model to classify the face image identified.

A convolutional neural network, or CNN, is a deep learning neural network designed for processing structured arrays of data such as images. Convolutional neural networks are widely used in computer vision and have become the state of the art for many visual applications such as image classification.

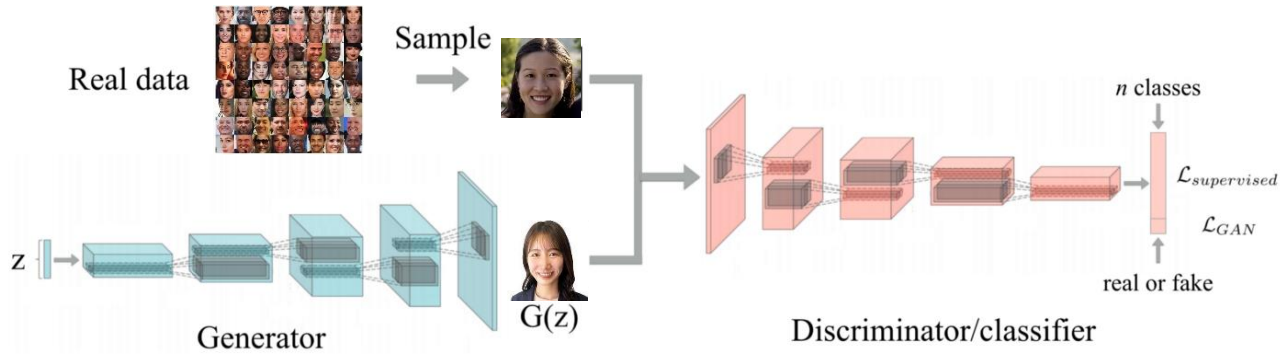
CNNs are very good at picking up on patterns in the input image, such as lines, gradients, circles, or even eyes and faces. It is this property that makes convolutional neural networks so powerful for computer vision. Other important properties of CNNs are:

- Translation invariance: in essence, it is the ability to ignore positional shifts of the target in the image. A cat is still a cat regardless of whether it appears in the top half or the bottom half of the image.
- Translation equivariance: for a convolutional neural network the position of the object in the image should not be fixed in order for it to be detected by the CNN. This means that a cat is still recognized as a cat even though, for example we change its color and then change its position. The same happens if we first shift its position and then its color. Summarizing, it simply means that if the input changes, the output also changes.



Deep Convolutional Generative Adversarial Network is basically a GAN (Generative Adversarial Network) architecture that uses convolutions. GANs are composed of 2 main networks called discriminator and generator which try to improve each other.

In the case of DCGANs instead of simple two-layer feed-forward networks, both generator and discriminator are implemented as convolutional neural networks. The generator takes the random noise in the latent vector and maps it to the data space. If we are using RGB images, our data-space means creating an RGB image. The generator starts with a dense or fully connected layer. After that, it is followed by transpose convolution and the leaky relu activation function. The discriminator is a simple binary classification network that takes both the real and the fake image and outputs a probability of whether the given image is real or fake.



Creating a real time system is a way to further asses the classification model robustness: the real and fake images are not always presented in a constrained environment but should be extracted from frames. The first operation that the system does is the extraction of the face image from the video frames, then the portion of frame in which the face was located is cropped and resized to have the same dimension of the images used to train the model. Finally, the image is ready to be classified by the model.

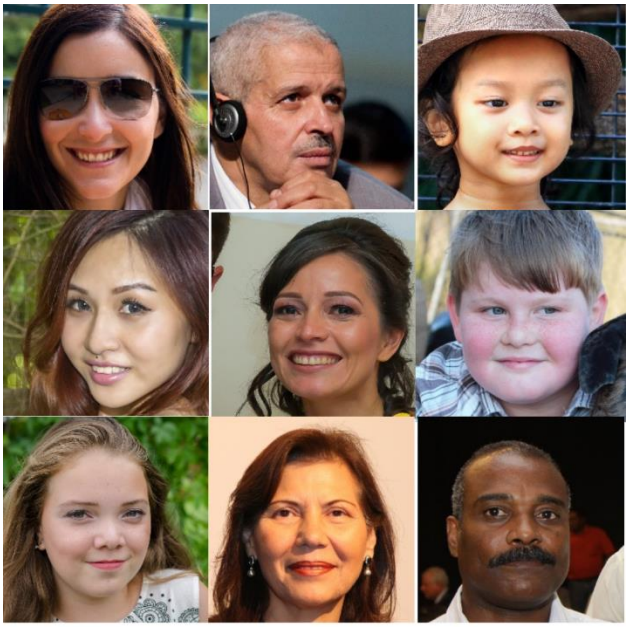
3. 140k Real and Fake Faces dataset

We decided to use the Kaggle dataset called "140k Real and Fake Faces". The link to the dataset is the following:

https://www.kaggle.com/datasets/xhlulu/140k-real-and-fake-faces

This dataset consists of all 70k REAL faces from the Flickr dataset collected by Nvidia, as well as 70k fake faces sampled from the 1 million FAKE faces (generated by StyleGAN) that was provided by Bojan. The data were split into three groups: train, test and validation set. In the training set we have 100,000 images while in the test and validation set there are 20,000 images respectively. In this dataset the images were initially resized to 256x256 pixels.

The dataset needs at least 4 GB of storage space so if you want to run the code you have to download the entire dataset at the Kaggle link indicated.



Examples of real images from the "140k Real and Fake Faces dataset"

Examples of fake images from the "140k Real and Fake Faces dataset"

4. Setup

To work on this project, we used several libraries:

- Python 3.8
- TensorFlow 2.2.0 and Keras (used to train and test our models: the first model recognizes whether an image is a fake or real face while the second is a DCGAN (Deep Convolutional Generative Adversarial Network) built to try to emulate the situation in which fake faces are generated and our original model has to classify them)
- NumPy (used to manipulate our data and transform the images captured in real time into NumPy arrays which we would then pass to our model)
- Pandas (used to insert our data into a DataFrame to better visualize them)
- OpenCV (used to run our model for the recognition of fake or real images in real time)
- Matplotlib (used to display graphs)

5. Implementation

5.1 Data analysis and transformation

When we are facing a deep learning problem, the first thing to do is to visualize the data.

The dataset we used did not have the classical structure of a csv file, which is probably the simplest way to manage data, but rather it was organized in files and directories where the label of each image is obtained from the name of the folder where the image is located.

For this reason, given the difficulty in visualizing the data, we initially decided to insert our data into a DataFrame, a data structure that allows us to manipulate data more easily and efficiently.

The DataFrame built is the following:

In [325]:
```
dataset = {"image_path":[],"label":[],"folder":[]}
for folder in os.listdir(path):
    for label in os.listdir(path+"/"+folder):
        for image in glob.glob(path+folder+"/"+label+"/"+"*.jpg"):
            dataset["image_path"].append(image)
            dataset["label"].append(label)
            dataset["folder"].append(folder)
df = pd.DataFrame(dataset)
df
```

Out[325]:

	image_path	label	folder
0	archive/real_vs_fake/real-vs-fake/valid/real/0...	real	valid
1	archive/real_vs_fake/real-vs-fake/valid/real/1...	real	valid
2	archive/real_vs_fake/real-vs-fake/valid/real/1...	real	valid
3	archive/real_vs_fake/real-vs-fake/valid/real/0...	real	valid
4	archive/real_vs_fake/real-vs-fake/valid/real/5...	real	valid
...	...	...	...
139995	archive/real_vs_fake/real-vs-fake/train/fake/4...	fake	train
139996	archive/real_vs_fake/real-vs-fake/train/fake/4...	fake	train
139997	archive/real_vs_fake/real-vs-fake/train/fake/4...	fake	train
139998	archive/real_vs_fake/real-vs-fake/train/fake/C...	fake	train
139999	archive/real_vs_fake/real-vs-fake/train/fake/S...	fake	train

140000 rows × 3 columns

As can be seen in the previous figure, the DataFrame contains 140,000 instances. For each instance is indicated the path where each face image is located, the label for that image (which can be "fake" or "real") and the type of set in which that instance is inserted: "train"," test" or "valid".

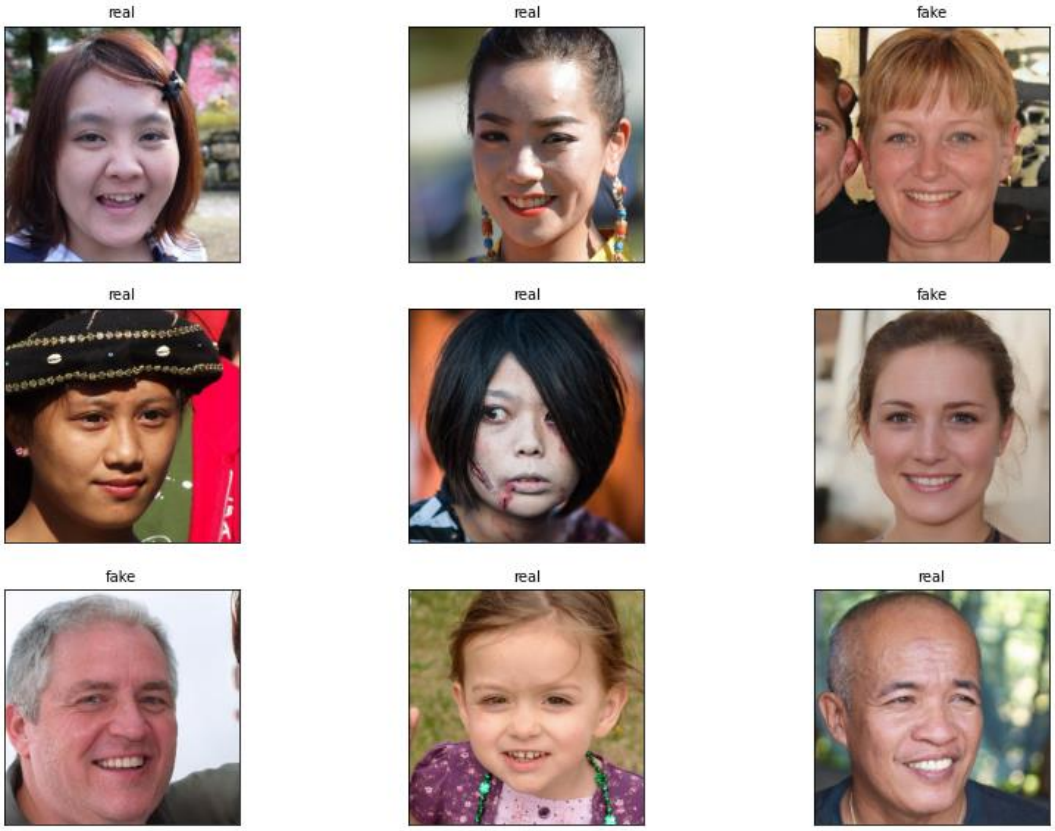
Looking at the figure below we can see how our dataset does not suffer from class unbalancement: in fact, 70000 instances are labeled with "fake", and, to the same extent, another 70000 instances are labeled with "real".

```
In [326]:  1 df[df['label']=='fake'].count()

Out[326]: image_path    70000
          label         70000
          folder        70000
          dtype: int64

In [327]:  1 df[df['label']=='real'].count()

Out[327]: image_path    70000
          label         70000
          folder        70000
          dtype: int64
```

As we have already said, visualizing the data is essential. The 9 random images shown below are taken from the DataFrame.

In [328]:

```
plt.figure(figsize = (14,10))
for i in range(9):
    random = np.random.randint(1,len(df))
    plt.subplot(3,3,i+1)
    plt.imshow(plt.imread(df.loc[random,"image_path"]))
    plt.title(df.loc[random, "label"], size = 10)
    plt.xticks([])
    plt.yticks([])
plt.show()
```



In the figure above we can see that the images are in RGB (3 channels), the faces are centered and frontal and that the fake faces have been generated with great accuracy by a powerful GAN because visually they are perfectly similar to images of real faces.

At this point, after verifying that our dataset has the right characteristics, we split the dataset into three parts: the training set, with which we trained a convolutional neural network (i.e. our binary classification model that establishes whether the image of a face is fake or real), the test set, to validate the robustness of our model and see its ability to generalize on unseen instances and the validation set, which instead we used to train a DCGAN with which we try to generate some fake instances starting from images of real faces which we then used to more robustly test our binary classification model.

After splitting the dataset, we applied preprocessing transformations on each image of the three sets to improve the performance of the models that will be fitted later.

We have scaled the pixels of each image by a scale of 1.0/255. Rescaling 1.0/255 helps us to transform every pixel value from range [0,255] to [0,1]. In this way we treat all images in the same manner: some images are high pixel range; some are low pixel range. The images share the same model, weights and learning rate. Since the training of the models takes a long time, we have also resized the images to a size of 128x128 pixels.

Next, we can see the code lines:

```
train_gen = image_dataset_from_directory(directory="archive/real_vs_fake/real-vs-fake/train",
                                         image_size=(128, 128), shuffle=True)

test_gen = image_dataset_from_directory(directory="archive/real_vs_fake/real-vs-fake/test",
                                        image_size=(128, 128), shuffle=True)

valid_gen = image_dataset_from_directory(directory="archive/real_vs_fake/real-vs-fake/valid/real",
                                         label_mode=None, image_size=(128, 128), batch_size=32, shuffle=True)
```

```
Found 100000 files belonging to 2 classes.
Found 20000 files belonging to 2 classes.
Found 10000 files belonging to 1 classes.
```

In the following code cell we have scaled the pixels of each dataset image.

```
rescale = Rescaling(scale=1.0/255)
train_gen = train_gen.map(lambda image,label:(rescale(image),label))
test_gen  = test_gen.map(lambda image,label:(rescale(image),label))
valid_gen = valid_gen.map(lambda image:(rescale(image)))
```

5.2. Training and testing of the binary classification model

After scaling and shuffling the data, we first built the binary classification model (which distinguishes fake and real faces) and then we trained it on our training set.

In the figure below we can see how we built the model using the Keras library. It is a sequential model consisting of 3 convolutional layers that generate 64, 128, and 64 feature maps respectively. Each feature map is generated by applying a weight filter, also called kernel. For each convolutional layer, 64, 128, 64 filters of size 3x3, 5x5 and 3x3 are applied respectively.

After each convolutional layer we inserted a batch normalization layer followed by a max pooling layer and dropout layer. The batch normalization layer acts as a regularizer of the parameters of our neural network. In fact, this layer reduces overfitting and improves CNN's ability to generalize on unseen instances. The max pooling layer also acts as a regularizer to reduce overfitting: it uses a 2x2 kernel to halve the output size of the previous layer. This process is not done randomly. A filter of 2x2 size is slid from right to left step by step on the output of the previous layer and returns in output only the maximum of the values among those that the kernel is going through in that step. The dropout layer also helps to reduce overfitting by acting directly on the nodes inside the network: during the training phase, some nodes and their connections (weights) are disconnected (not considered) from the network in such a way that the model does not focus only on particular features and does not risk to memorize the data but rather generalizes more effectively.

In sequence with the 3 convolutional layers, we added a flatten layer to arrange on a single vector the values of the outputs of the previous layers and two dense layers interspersed with a batch normalization layer and a dropout layer. The last dense layer has only 2 nodes that will be activated not by our classic ReLu function but by the Softmax function which returns, in our case, 2 probabilities: one is the probability that an instance is classified as a "fake" image, the other it is the probability that instead it is classified as a "real" image.

```
model = Sequential()
model.add(Conv2D(64, kernel_size=(3, 3), activation='relu', input_shape=input_shape))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2)))

model.add(Conv2D(128, kernel_size = (5,5), activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.20))

model.add(Conv2D(64, kernel_size = (3, 3), activation = 'relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.30))

model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(BatchNormalization())
model.add(Dropout(0.20))
model.add(Dense(num_classes, activation = 'softmax'))
```

Our first model has 1,666,882 parameters of which 1,666,114 are trainable ones. They are the reason why the training phase, since we chose to train the model for 15 epochs, took several hours.

The following image shows how we compiled and trained our model.

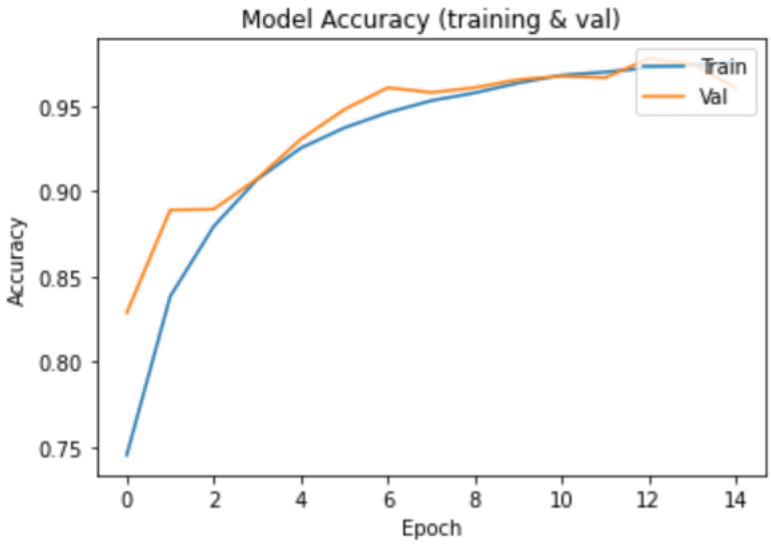
```
model.compile(loss="sparse_categorical_crossentropy", optimizer=tf.keras.optimizers.SGD(), metrics=['accuracy'])
model.summary()
```

```
%%time
history = model.fit(train_gen, validation_data=test_gen, callbacks=callbacks, epochs = epochs, batch_size=batch_size)
```

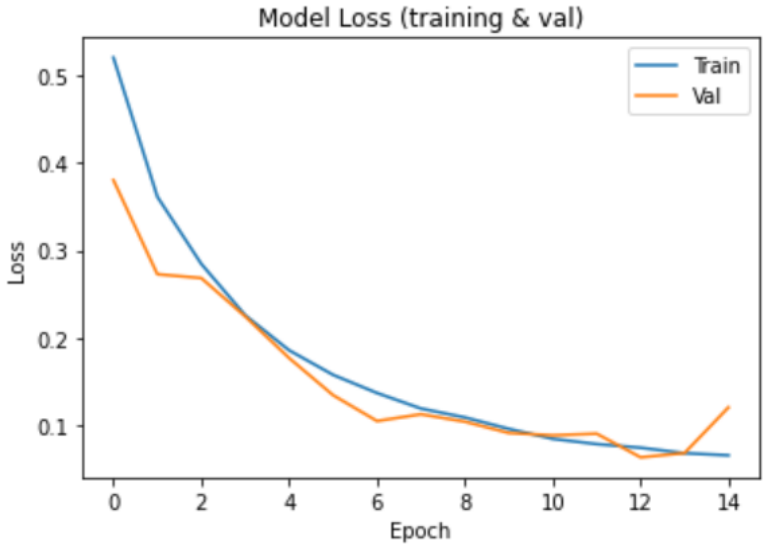
After training our CNN, we tested its ability to predict the right class ("fake" or "real") on the test set images, which were never seen by the model.

Analyzing the graphs below we can notice how the loss and accuracy values vary with respect to the increase of the epochs and we can see how the introduction of the batch normalization, max pooling and dropout layers was very useful in minimizing overfitting. In fact, with the progress of the epochs, the accuracy on the test set increases and always has values very similar to the accuracy on the train set in that same epochs. Similarly, the loss values follow the same trend as the accuracy values: with the passing of the epochs the loss values on the test set decrease and tend to be similar and often lower than the values calculated for the train set in that epoch.

The following plot shows how the accuracy varies for training and testing.



The following is the loss variation graph.



5.3. Deep Convolutional Generative Adversarial Network (DCGAN)

The idea was to create a Deep Convolutional Generative Adversarial Network that uses as a discriminator a CNN identical to the model implemented above except for the last dense layer which instead of having 2 nodes has only 1. This because in the end we want to have the probability of only one class (in our case the "real" class) from which then, with the generator we create fake faces images.

As we have already said, the discriminator is identical to the binary classification model with the exception of the last layer. As generator, we used a model that operates in a way that is contrary to the discriminator. As the first level of the generator, we have inserted a dense layer followed by 4 Convolutional2DTranspose which use 4 kernels of 4x4 size and stride value of 2.

The transposed Convolutional Layer is also (wrongfully) known as the Deconvolutional layer. A deconvolutional layer reverses the operation of a standard convolutional layer i.e. if the output generated through a standard convolutional layer is deconvolved, we can get back the original input. A transposed convolutional layer is usually carried out for upsampling i.e. to generate an output feature map that has a spatial dimension greater than that of the input feature map.

Just like the standard convolutional layer, the transposed convolutional layer is also defined by the padding and stride. These values of padding and stride are the one that hypothetically was carried out on the output to generate the input. i.e. if you take the output, and carry out a standard convolution with stride and padding defined, it will generate the spatial dimension same as that of the input.

After each Convolutional2DTranspose we used the LeakyReLU function as the activation function. It prevents the dying ReLU problem.This variation of ReLU has a small positive slope in the negative area, so it does enable back-propagation, even for negative input values back-propagation, even for negative input values.

This generator has the purpose of reconstructing fake images (but apparently as real as possible) of the same size as the original ones passed in input to the discriminator (we used the images of the validation set which were not used in the previous steps).

In the picture below the generator is shown.

```
latent_dim = 128

generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)
generator.summary()
```

To train the DCGAN (i.e. the discriminator with the generator together) we overrode the train_step(...) method of the Keras GAN class.

In the next figure, we can see the code lines that start the training of the DCGAN on the validation set (where we inserted only real face images) which generates new fake instances which we want to use to test more efficiently the binary classification model.

```
epochs = 30

gan = GAN(discriminator=discriminator, generator=generator, latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

gan.fit(
    valid_gen, epochs=epochs, callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)]
)
```

The DCGAN has been trained for 30 epochs and generated 10 new images for each epoch. The greater the number of epochs, the more the new generated instances (fake) will seem to us images of real faces. However, we have decided to stop at the thirtieth epoch for reasons of time and resources. Nonetheless, we still achieved a good level of image realism for some images.

After generating these new images, we chose the better defined image and we ran our original binary classification model on this instance to further test the robustness of our model and check if it is actually recognized as fake.

The classifier actually recognizes the instance as fake. It must be considered that the model is tested on instances obtained after processing 30 epochs. The faces in the images after 30 epochs have clear human features, and in fact we are able to recognize eyes, nose, mouth, head shape, facial expressions but needed way more epoches to be be well defined. Therefore, we expect to have a reliable accuracy for our binary classification on instances generated by the DCGAN trained on hundreds of epoch. This is because the tested images will be much more accurate, clearer and realistic with a DCGAN trained much longer.

Some other images generated by the DCGAN can be found in the “dcgan_generated_images” folder.

5.4. Real time classification of face images

Lastly, we decided to test our binary classification model on face images captured in real time with our laptop's front camera. We wanted to see how our model was performing on images that were not exactly frontal, with different intensity of brightness and sharpness.

Our model was trained on images with close-up and centrally positioned faces, so it was necessary to use a pre-trained classifier from the OpenCV library which implements a cascade of classifiers (cv2.CascadeClassifier(cv2.data.haarcascades + "haarcascade_frontalface_default.xml")) to detect people's faces in any image. After detecting the faces in each frame of the video in real time, we cut a square around the foreground face and resized the resulting face image to 128x128 pixels (the images we trained our classifier on use images of this size). At this point we ran our model on the obtained image which gives us in real time the probability of that image to belong to the "fake" or "real" class.

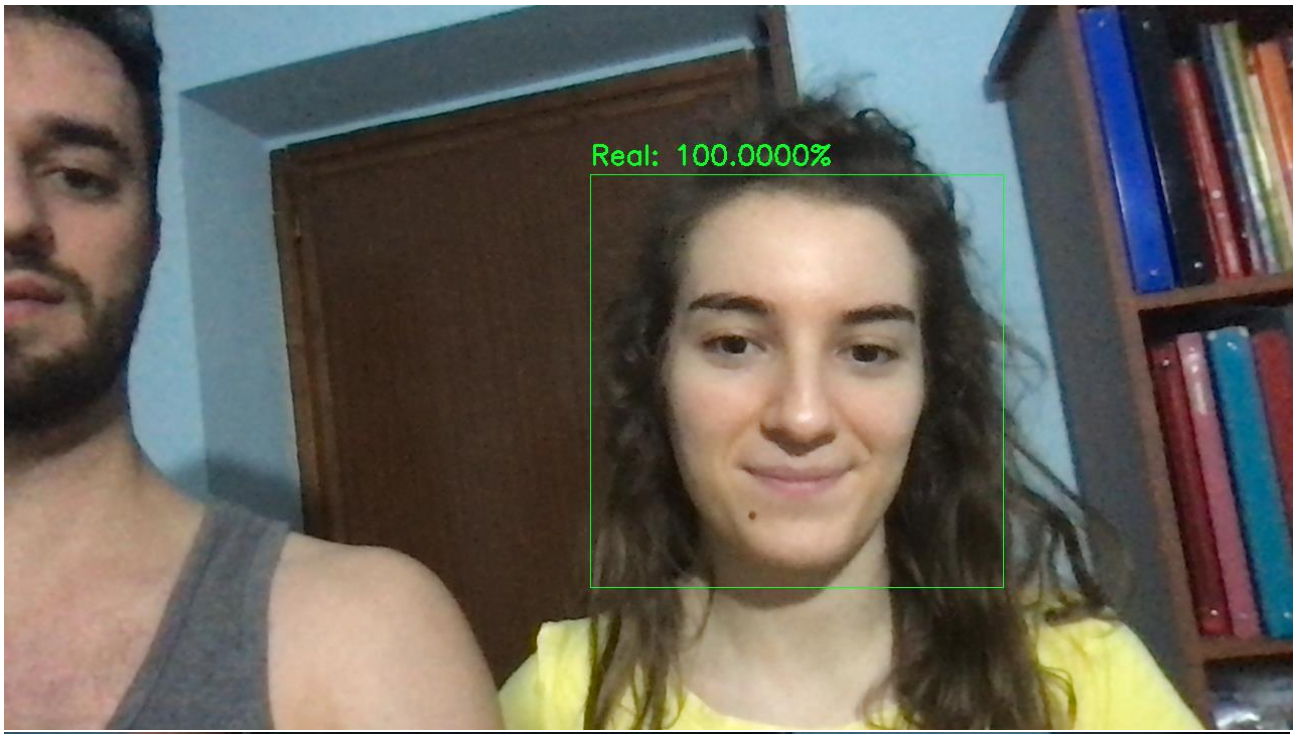
This process is repeated continuously and in real time for each frame.

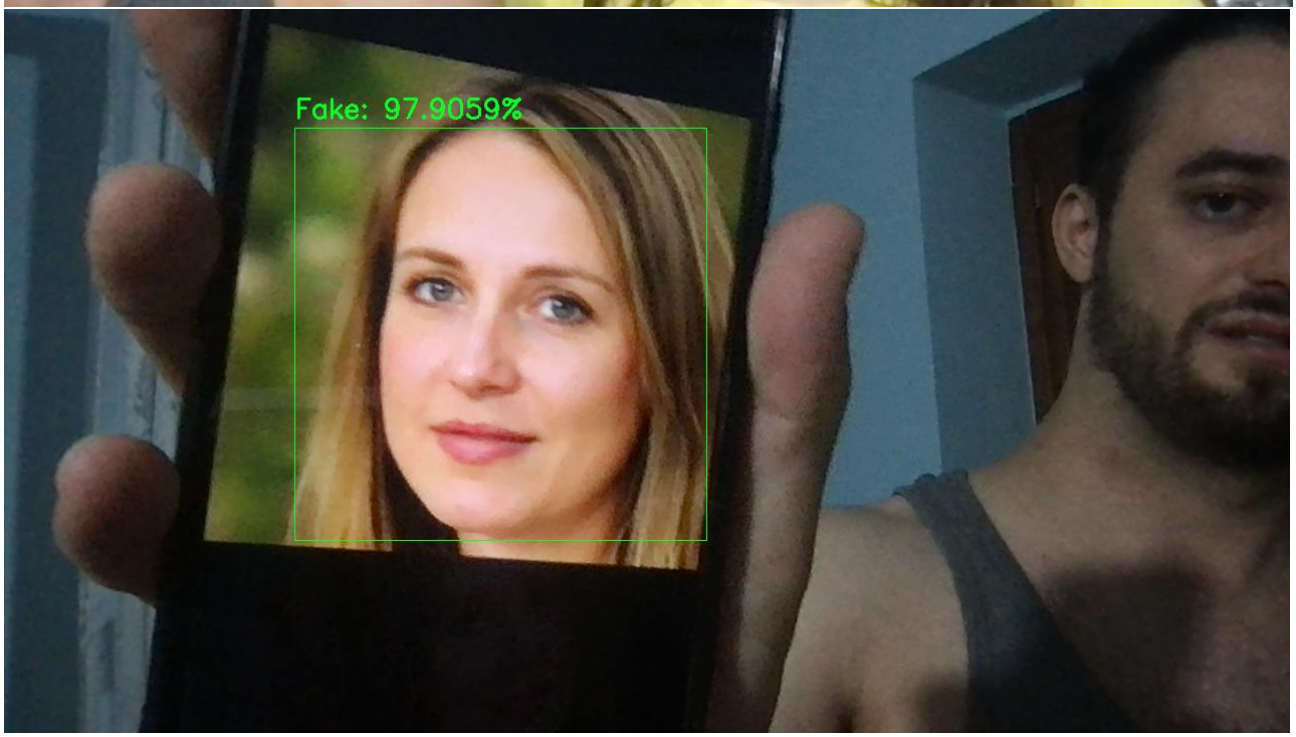
As expected, our CNN does not always perform optimally. Sometimes, even by showing a face of a real person to the camera, the model can classify it as "fake". This occurs especially when the light is not optimal or when the person's face is not perfectly frontal.

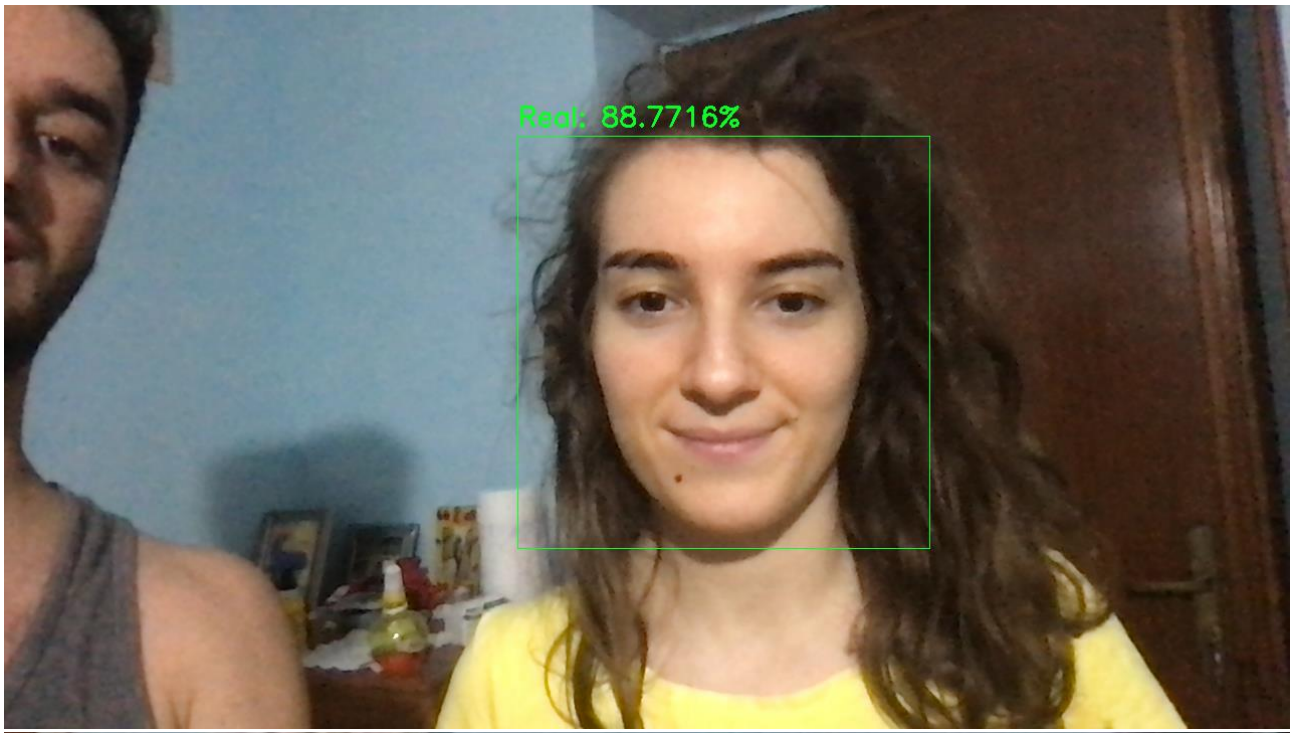
To test fake images in real time we took some images from the popular site “thispersondoesnotexist.com” which freely provides GAN generated fake faces images. We displayed them on our smartphone and placed it in front of the computer's front camera. In several cases the model was able to identify photos of fake faces with good accuracy, in others it classified incorrectly.

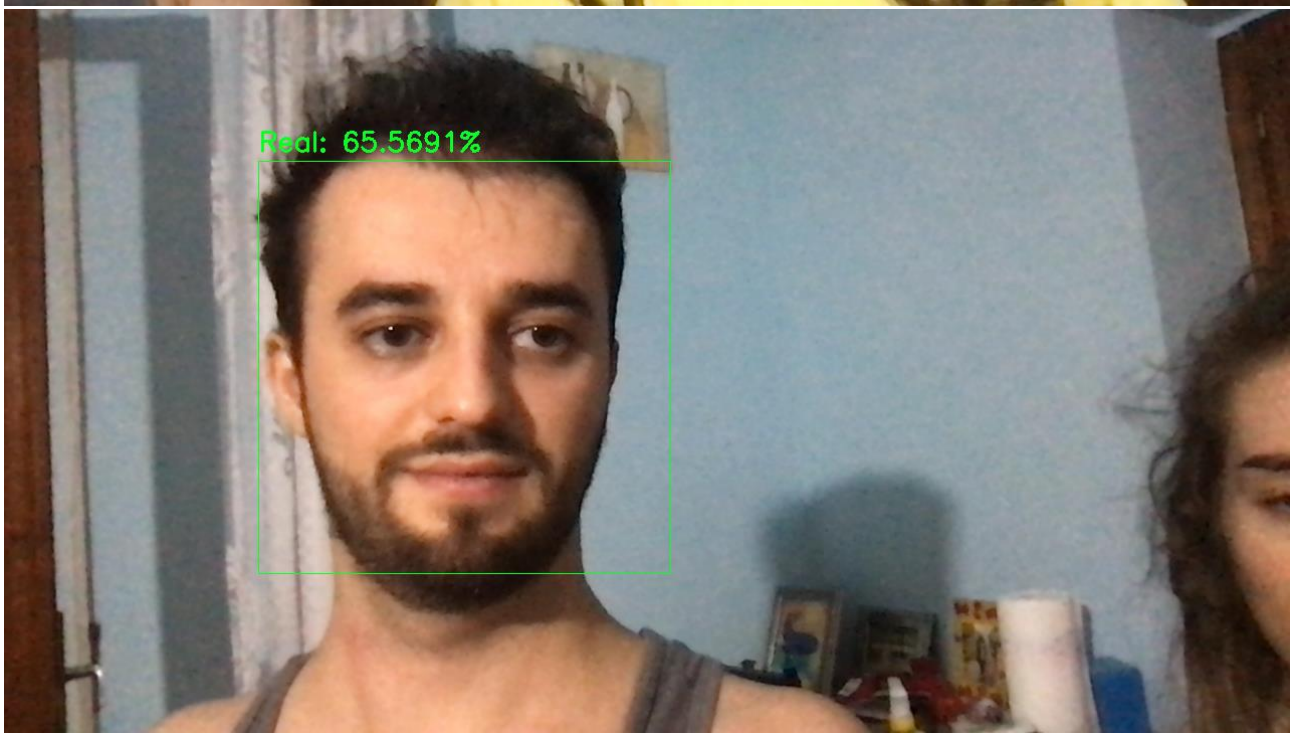
In general, the model has almost always correctly classified the frames with real faces and in good part of the cases the fake ones.

In the following pictures we can see some frames of real time video in which we have captured moments in which the model has predicted the "fake" class and the "real" class with their respective probabilities.


Real: 100.0000%


Fake: 97.9059%





Some other pictures of the real time classification can be found in the “frames_real_time_predictions” folder.

6. Conclusions

In this project we focused on the realization of a binary classification model that could distinguish as accurately as possible instances of fake faces from real ones.

We were able to build a good model that reaches an accuracy of 0.96 in testing on unseen instances. We then wanted to further test the ability of our model to generalize on fake face images generated by a Deep Convolutional Generative Adversarial Network (DCGAN).

Finally, with the support of a pre-trained cascade classifier to detect the faces within the images, we have extrapolated the images of the faces from each frame of a video in real time and classified them with our model.

However, the model achieves good performance when the faces are frontal, uniformly illuminated and detailed. In other cases, it tends to make more mistakes.

In a future development, we could refine the binary classification model and make it more robust even in real time.

As for the DCGAN, the number of epochs with which to train the model could be increased to improve the realism of the generated images.