Università degli Studi di Padova

Department of Information Engineering

Master Thesis in ICT For Internet and Multimedia

Face Forgery Detection Using Conditional GAN

Supervisor
Simone Milani
Università di Padova

Co-supervisor
Marco Fontani
AMPED s.r.l.

Master Candidate
Gianmaria Rossi

16 December 2019 ACADEMIC YEAR 2018-2019

Abstract

Face forgery, specifically in video sequences, is an increasing threat due to an easy access to video manipulation tools and channels where to share those videos. The study of a face forgery detection tools is new, and even if there are already some solutions to detect manipulated videos, the techniques used to create the forgery keep evolving. This work propose a Deep Learning approach to generate a mask showing the region of a tampered face. A Conditional Generative Adversarial Network (cGAN) is used, and a total of 16 features are extracted from the video and used as input of the neural network. This method is tested on video downloaded from YouTube and a dataset provided by a collaboration by Google and Jigsaw. The obtained results are extremely promising , although there is still room for improvements.

Contents

Listing of figures

Listing of tables

1 Introduction

In today's digital age the use of videos are widespread all around the world thanks to Internet and TVs. They are used to entertain, to express ideas, to share news, as legal evidence , for educational purposes. Apart from many good things, videos are also used in a negative way for example : blackmailing, political defamation, terrorism promotion.

The improvement of computers and cameras, together with video editing tools, allows even a novice individual to make unauthorized modifications of the content of a video, thereby affecting its integrity and reliability [5].

In 2017 a new type of forged video has appeared in Internet, the deepfakes. The name comes from the username of a Reddit user which has created a tool to superimpose the face of an actress in a pornographic video using an autoencoder-decoder pairing structure: the autoencoder extract latent features of face images and the decoder is used to reconstruct the face images [6]. In January 2018 a desktop application used to create deepfakes called FakeApp has become available for download. Later in that year, several social media banned deepfakes and associated communities [7]. Since the diffusion of deepfake videos, multiple apps to modified a persons face have been created such as: FaceApp, Zao, Snapchat's filters.

Several corporations and companies have launched a deepfake detection challenge releasing a dedicated dataset to incentivate a progress in this area by inviting participants to compete to create new ways of detecting and preventing manipulated media[8] [9].

In June 2019 a hearing held by the House Intelligence Committee on national security challenges of artificial intelligence, manipulated media, and deepfakes, discusses the dangers

of deepfakes[10].

Forged video detectors can be divided in two categories: active and passive detection. The first approach actively modifies the data to be protected at the origin and pre-embeds some information like watermark, fingerprint into digital images or digital signature, and identify them with an integrity detection of the pre-embedded information [11]. These techniques suffer from the following limitation [12]:

- A person can manipulate the video before applying the watermark.
- Several encryption techniques can be used to avoid unauthorized persons to modify the video, however the file owner can modified it before the encryption.
- The need of a special hardware or software in post processing the digital video to insert the watermark

The passive approaches rely on the unintentional fingerprint traces left during video acquisition and editing to detect tampered region [13]. In [12], this approach is divided in three categories: methods computing statistical correlation of video features, frame-based solutions detecting statistical anomalies, and strategies revealing inconsistencies with respect to the digital equipment.

In this work we have developed a conditional GAN that, by using a set of features extracted from the video sequence, creates a binary mask showing the region where the video has been modified. This neural network is trained specifically on face forgery detection.

Other works have proposed solution to the video forgery detection problem. In [14] two convolutional neural networks are used to classify respectively the codec and quality of the frames: by analyzing the differences in the outcomes it is possible to identify regions where the video has been tampered. The work in [15] proposed a convolutional neural network to estimate quantisation parameter, deblock settings and intra/inter model of pixel patches from an H.264/AVC sequence. The author of [16] proposed a convolutional neural network capable to automatically learn manipulation features directly from training data; to achieve this a new type of convolutional layer has been designed which suppresses an image's content. By constraining the first layer of convolutional filters to learn only a set of prediction error filters the CNN is allowed to adaptively learn a strong set of manipulation detection feature extractors.

Due to the increasing risks of deepfake, several article propose their own solution. In [17] a CNN is used to extract frame-level features that are used to train a Recurrent Neural Network

(RNN) that learns to classify if a video has been manipulated. The creation of deepfake leaves some artifacts in the region of the modified face. The approach in [18] shows how a CNN is able to capture these artifacts. The solution in [19] uses a deep learning approach focused on the mesoscopic properties of images.

This thesis is structured as the following: Chapter 2 describes the key elements of a neural network and some architectures used in the project. In Chapter 3 is described the Generative Adversarial Network and some specific designs. Chapter 4 is dedicated to the architecture of the neural networks. Chapter 5 describes the dataset used to train and test the neural network. In Chapter 6 it is explained how the neural network is trained and the results achieved with it. Chapter 7 is dedicated to the conclusions.

2

Neural Networks

The main components of a neural networks are, as the name suggests, the neurons. The concept of neural networks was created by Warren McCulloch and Walter Pitts in 1943,[20][21]. Their model was very simple, with two classes of neurons: the peripheral afferent neurons (input and output neurons) and the inner neurons. The output of each neuron is either excitatory or inhibitory, but not both. In the network each neuron is in the state of firing or not firing. For the inner neurons this state depends on the state of the input neurons: if the number of firing input neurons is greater than a threshold value and none of the input neuron is an inhibitory one, then the neuron will fire.

In 1949, Donald Hebb, [22], a Canadian psychologist, revolutionized the way that artificial neurons were perceived. In The Organization of Behavior he affirms that if a neuron continues to fire to another neuron, the connection between the two will strengthens; and this is a fundamental activity for learning and memory. With these information a new technique was applied to the artificial neuron of McCulloch and Pitts: a weight is applied to each input.

Between 1950s and 1960s Frank Rosenblatt, using the model of McCulloch and Pitts, and the information of Hebb, developed the perceptron. The perceptron takes in input a finite number of binary values, x_1, x_2, x_3,.., x_n; each of them has a weight, w_1, w_2, w_3,..., w_n. The output of the neuron is determined by the following expression:

$$\begin{cases} 0 & if \quad \sum_{i=1}^{n} w_i x_i < b \\ 1 & if \quad \sum_{i=1}^{n} w_i x_i > b \end{cases}.$$

where b is the threshold value. We can move b to the left part of the equation.

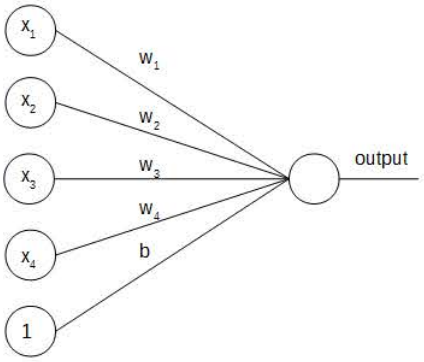


Figure 2.1: Simple model of a neuron

A limitation of the perceptron model is that both the input and output of the neurons are 0 or 1, and this makes the learning process hard since intermediate or soft values are not conceived. For this reason the output is processed by a different activation function: the sigmoid. The perceptron model with sigmoid activation is similar to the perceptron one, but it is possible to apply small changes to the weights to have small changes in the output. Each input x_1,x_2,x_3,..,x_n of the sigmoid neuron has a value included between 0 and 1, and as for the perceptron, each input has its weight w_1,w_2,w_3,...,w_n. The difference relies on the way the output is computed. For the sigmoid model, the output of the neuron is defined as $\sigma((\mathbf{wx} + b)$ where $\mathbf{wx}$ is the dot product between the weight vector $\mathbf{w}$ and the input vector $\mathbf{x}$. The function σ is called sometimes logistic function and it is defined as :

$$\sigma(z) = \frac{1}{1 + e^{-\mathbf{z}}}.$$

We can rewrite it more explicitly using $\mathbf{w}$, $\mathbf{x}$, and b as

$$\sigma(\mathbf{wb}) = \frac{1}{1 + e^{\sum_{i=1}^{n} w_i x_i - b}}.$$

It is possible to see in Fig. 2.2 that the output of σ is bound between 0 and 1; for z very negative $e^{-z} \to \infty$ and $\sigma \approx 0$, while for z very positive $e^{-z} \to 0$ and $\sigma \approx 1$.

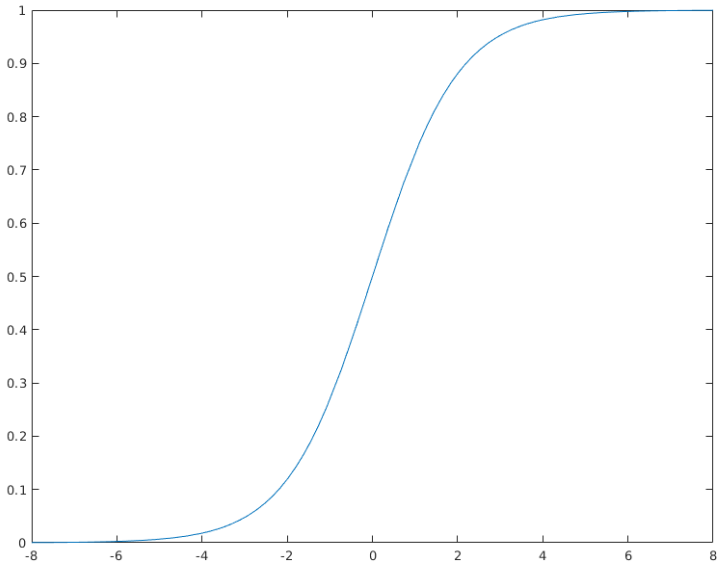


Figure 2.2: Sigmoid activation function

For these cases the behaviour of the sigmoid neuron is similar to the perceptron neuron, but enables real number output between 0 and 1; for this reason, it is commonly used to predict probabilities. The sigmoid function is differentiable, this is very important since the derivative is used for the learning process of the neural network.

Other popular activation functions a(z) are used in different neural networks such as the step function, the hyperbolic tangent (tanh) function, the Rectified Linear Unit (RelU) function, and the leakyRelU function.

The step function is the one used to obtain the perceptron model, in fact its output is 0 if $\mathbf{wx} + b \leq 0$, 1 otherwise.

The tanh function, is defined as

$$a(z) = \frac{sinh(x)}{cosh(x)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

The range of this activation function is (-1,1) and like the sigmoid function it is differentiable and monotonic, but its derivative is not monotonic. Typically it is used in problems where the binary classification is needed.

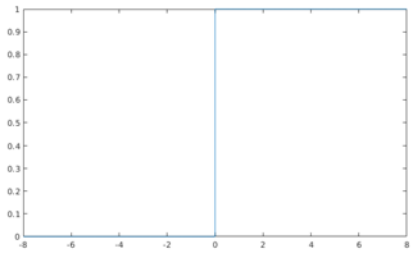
The ReLU function, is defined as

$$a(x) = \begin{cases} 0 & if \quad x < 0 \\ x & otherwise \end{cases} .$$

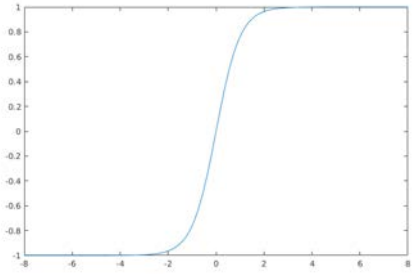
ReLU is a piecewise linear function which prunes the negative values and retain the positive ones [23]. Since it is a simple operation it is much faster to compute than the sigmoid or the tanh activation function. The problem of using a ReLU activation function is the discontinuity "of the derivative" at 0.
LeakyRelU changes from RelU as

$$a(x) = \begin{cases} \alpha x & if \quad x < 0 \\ x & otherwise \end{cases}$$

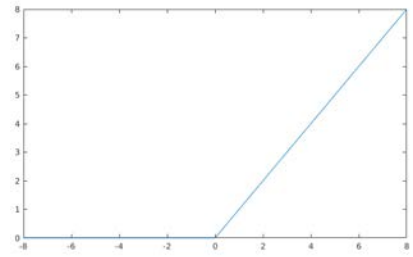
where α is a constant in the range (0,1). Instead of setting to zero the negative values, leakyReLU multiplies them with a constant value, thus avoiding the discontinuity at 0. This activation function is the one that will be used in the neural network used for the deepfake detection task: reasons are explained later.



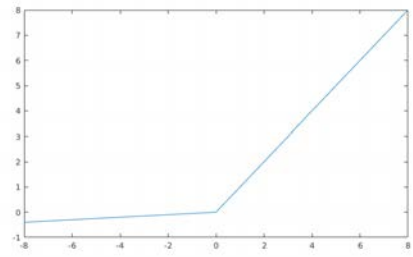
(a) Step activation function.



(b) tanh activation function.



(c) ReLU activation function.



(d) leakyReLU activation function.

Figure 2.3: Activation functions.

Different activation functions are used because each one has its own derivative, and as we have introduced before, the derivative is used to modify the value of the weight of the neurons' inputs. If we focus on a single neuron, a slight change of weights $\Delta\mathbf{w_i}$ and bias Δb will produce a small change to the output, $\Delta output$, that can be approximated as

$$\Delta output \approx \sum_i \frac{\partial output}{\partial w_i}\Delta w_i + \frac{\partial output}{\partial b}\Delta b \ ,$$

where $\frac{\partial output}{\partial w_i}$ and $\frac{\partial output}{\partial b}$ are the partial derivatives of the output with respect of w_i and b respectively. The previous expression shows that $\Delta output$ is a linear function of $\Delta\mathbf{w}$ and Δb. Since each activation function has its own derivative, $\Delta output$ change in different way depending on the activation chosen.

To understand how a neural network works and learns in the next figure it is shown a simple architecture.

Each neural network is made of different layers: the first one is called input layer, and the last layer is called the output layer. Between these two layer we can have one or more layers, each one called hidden layer. Typically the input of each node, with exception of the neurons of the input layer, are the output of all the neurons of the previous layer. This type of network is called feedforward neural network, there are several architectures such as Convolutional Neural Network (CNN), which will be used in this work and will be explained later, and Recurrent Neural Network (RNN), where the input of some layers can be made of output of the same layers at previous instant (i.e., there are loops in the network). To better understand future expressions, we introduce the following notation: w^l_{jk} is the weight between node k of the l^{th} layer and node j of the $(l-1)^{th}$ layer.

Given an input set $X = x_1, x_2, .., xn$ and a label set $T = \mathbf{t_1}, \mathbf{t_2}, .., \mathbf{t_n}$ associated to the input, the goal of a neural network is to have the output $\mathbf{y}(\mathbf{x_i})$ equal to $\mathbf{t_i}$. To measure the accuracy of the neural network a cost function C is associated to y, for example we can use the mean squared error (MSE): $C(w,b) = \frac{1}{2N}\sum_{i=1}^{N}||\mathbf{y}(\mathbf{x}) - \mathbf{t}||^2$,

where w and b represent all the weights and biases of the network. We can notice this cost function is non-negative and when it is close to o, the output is almost equal to the label. To minimize the cost function, the gradient descent is used, if we consider the cost function C a function of n variables, $C(w_1, w_2, ..., w_n)$, a change ΔC of C is caused by Δw ,

$$\Delta C \approx \nabla C \Delta w.$$

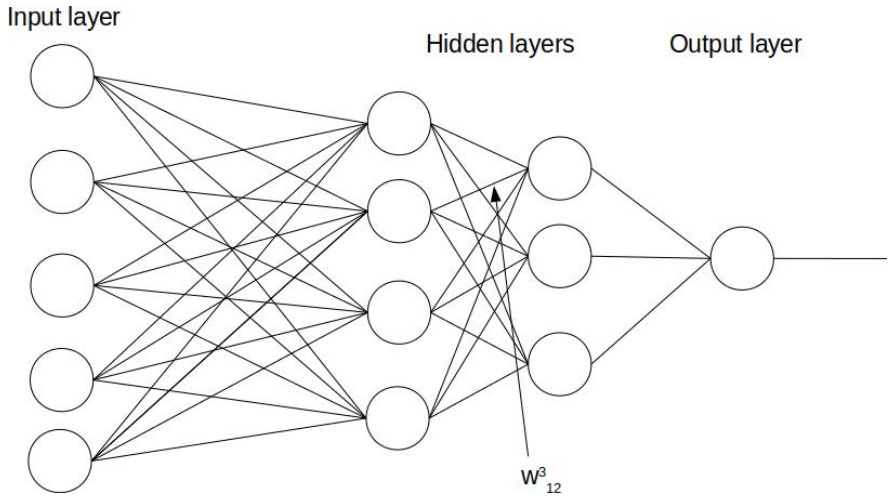


Figure 2.4: Fully Connected Neural Network

where ∇C is the gradient of the cost function $\nabla C = (\frac{\partial C}{\partial w_1})$.

We can make ΔC negative by choosing an appropriate value of Δw, for example we can have

$$\Delta w = -\eta \nabla C,$$

where η is a small positive value that we will call learning rate. We want a small value to avoid changing too much ΔC and risking to never reach a minimum point.

If we substitute Δw in the equation for ΔC we obtain

$$\Delta C \approx -\eta \nabla C \nabla C = \eta ||\nabla C||^2.$$

Since $||\nabla C||^2$ is always positive we are sure that ΔC is always negative. Now, let us consider $C(w, b)$, ∇C is function of $\frac{\partial C}{\partial w_j}$ and $\frac{\partial C}{\partial b}$, more precisely

$$\nabla C = (\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, ..., \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b}).$$

The weights and bias are updated according to:

$$w_j \leftarrow w_j - \eta \frac{\partial C}{\partial w_k}$$
$$b \leftarrow b - \eta \frac{\partial C}{\partial b}.$$

One problem arises from these operation, since the cost function computes the MSE over all the training samples. When we are training our neural network with a lot of samples, the

time needed for a single update of each weight increases rapidly; to overcome this problem, the stochastic gradient descent (SGD) method is used. The SGD works by computing the gradient descent over a small subset of training samples of size m taken randomly from the whole set.

More precisely, suppose we have a set $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_m}$ of input value of size m called mini-batch taken randomly from a the input set of size n, with m big enough to obtain the following approximation:

$$\frac{\sum_{i=1}^{m} \nabla C_{x_i}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C.$$

We can then update the vector of weights w_j and the vector of bias b_j as shown in the following

$$w_j \leftarrow w_j - \frac{\eta}{m} \sum_{i=1}^{m} \frac{\partial C_{x_i}}{\partial w_k}$$

$$b_j \leftarrow b - \frac{\eta}{m} \sum_{i=1}^{m} \frac{\partial C_{x_i}}{\partial b_j}.$$

2.1 Backpropagation

In order to minimize the cost function, the backpropagation is used.

The first implementation to run on a computer was done by Seppo Linnainmaa in 1970[24], but the first theoretical models were made during the 1960s. The work referenced by [24], dating back to 1986, describes the back-propagation as a new learning procedure for networks of neuron-like units to change the weights in order to minimize the cost function. The change of weights is done in parallel for nodes in the same layer, while for nodes in different layers the change is made sequentially.

To continue the explanation of the back-propagation two consideration need to be made: the first one is that the cost function $C(w, b)$ can be written as an average of the cost function of each sample; the second is that the cost function can be written as a function of the output of the neural network. The reason for the first consideration is that in the backpropagation we calculate the partial derivative $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ of each sample individually, then we obtain $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging over all the training sample.

If we remember the input of a neuron of layer l is:

$$z_k^l = \mathbf{w}^{l-1} a(\mathbf{z}^{l-1}) + b^{l-1} \,,$$

where $\mathbf{w}^{l-1}$is the vector of weights between the neuron of the previous layers and node k of

layer l, a is the activation function, $\mathbf{z^{l-1}}$ is the vector of output of the previous layer, and $\mathbf{b}$ is the bias. In a neural network with L layers and m output neurons we can define $\delta_k^l = \frac{\partial C}{\partial z_k^l}$ as the error of neuron k in layer l. For the error of each neuron of the last layer we obtain

$$\partial_k^L = \frac{\partial C}{\partial a_m^L} a'(z_k^L) = \sum_k (\mathbf{t}_k - a(\mathbf{z_k^L})) \odot a'(\mathbf{z}^L),$$

while for a general layer we have

$$\delta^l = ((\mathbf{w}^{l-1})^T \delta^{l+1}) \odot a'(\mathbf{z}^l),$$

then it is possible to backpropagate the derivative to

$$\frac{\partial C}{\partial w_{jk}^l} = \delta_k^l a_k^{l-1} \qquad\qquad \frac{\partial C}{\partial b_m l} = \delta_m^l.$$

Now that we have these equations we can apply the backpropagation method to the whole network. The first operation we do is the forward pass, for each layer l=2,3,...,L we compute $\mathbf{z}_l = \mathbf{w}^l \mathbf{a}^{l-1} + \mathbf{b}^l$ and $a^l = \sigma z^l$.

The error vector of the last layer is computed, $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

From this value we backpropagate the error to layers l=L-1,L-2,...2 and we obtain

$$\delta l = ((\mathbf{w}^{l-1})^T \delta^{l+1}) \odot a'(\mathbf{z}^l).$$

Lastly we calculate the gradient of the cost function as

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad and \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l,$$

and update the value of the weights and bias with the following expression:

$$w_{jk} \leftarrow w_{jk} - \eta \frac{\partial C}{\partial w_{jk}}$$
$$b_j \leftarrow b_j - \eta \frac{\partial C}{\partial b_j},$$

and if we are using a stochastic gradient descend with mini-batch of size m we have

$$w_{jk} \leftarrow w_{jk} - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$
$$b_j \leftarrow b_j - \frac{\eta}{m} \sum_x \delta^{x,l}.$$

A problem we can encounter during the training of a neural network is that an artificial neuron has a poor learning performance whenever the estimation error is high[25]. To understand why it learns very slowly we have to analyze the partial derivatives of the cost function, $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$, with the quadratic cost function we have

$$C = \frac{(y-o)}{2},$$

where o is the output of the neuron when its input is $x = 1$, and the desired output is $y = 0$. Whenever the neuron is learning slowly, the value of the partial derivatives are small. If we write the output in terms of the weight and bias we have $o = a(z) = a(wx + b)$, and by using the chain rule to differentiate with respect to the weight and bias we obtain

$$\frac{\partial C}{\partial w} = (o-y)a'(z)x = oa'(z)$$
$$\frac{\partial C}{\partial b} = (o-y)a'(z) = oa'(z).$$

If we look at the plot of the sigmoid activation function, when the neuron's output is very close to 1, the curve is flat, hence $a'(z)$ gets very small. This is the cause of the slow learning.

A solution to this problem is achieved by substituting the quadratic cost function with the cross-entropy cost function. Given a neuron with input $x_1, x_2, ..., x_n$, corresponding weights $w_1, w_2, ..., w_n$, and bias b; its output is given by $o = a(z) = a(\sum_{i=1}^{n} w_i x_i + b)$. The cross-entropy cost function is defined as

$$C = -\frac{1}{m}\sum_{x}[y\ ln\ o + (1-y)ln(1-o)],$$

where m is the number of sample and y is the desired output.

The previous equation can be used as a cost function since it has the following properties.

- It is non-negative ($C > 0$) since the argument of the logarithms are between 0 and 1: the sum of logarithms are always negative, and there is a minus sign that make it positive.
- If the desired output is 0 or 1, then when the output of the neuron is close to the desired output then cross-entropy is close to zero.

To see that the cross-entropy does not have the problem of slow learning we compute the partial derivatives of the cost function with respect to the weight as before. If we apply two

times the chain rule we obtain

$$\frac{\partial C}{\partial w_i} = -\frac{1}{m}\sum_x (\frac{y}{a(z)} - \frac{1-y}{1-a(z)})\frac{\partial o}{\partial w_i}.$$

$$\begin{aligned}\frac{\partial C}{\partial w_i} &= -\frac{1}{m}\sum_x (\frac{y}{a(z)} - \frac{1-y}{1-a(z)})\frac{\partial o}{\partial w_i} \\ &= -\frac{1}{m}\sum_x (\frac{y}{a(z)} - \frac{1-y}{1-a(z)})a'(z)x_j.\end{aligned}$$

By putting everything in a common denominator and after simplifying it we have

$$\frac{\partial C}{\partial w_i} = \frac{1}{m}\sum_x (\frac{a'(z)x_j}{a(z)(1-a(z))})(a(z)-y).$$

With the sigmoid activation function, $a(z) = \frac{1}{1+e^{-z}}$, its derivative is $a'(z) = a(z)(1 - a(z))$, so in the previous expression we can remove the fraction, and we obtain

$$\frac{\partial C}{\partial w_i} = \frac{1}{m}\sum_x x_i(a(z)-y).$$

We can observe that we no longer use $a'(z)$ to change the value of the weight, so we do not have a slow training caused by a strong error of the neuron; moreover the rate at which the weight learns is controlled by $a(z) - y$, that is the error on the output, so a big error means that the neuron will train faster.

2.2 Convolutional Neural Networks

The previous sections described how a deep neural network works. Unfortunately this model has a problem, the number of parameters that need to be trained. The number of parameters for a neural network with L layers is given by $\sum_{l=2}^{N}(nl * n(l-1)) + nl$. Suppose we have a deep neural network with four layers, the first layer has 5 nodes, the two hidden layers have 8 nodes each, and the output layers has 2 nodes, all the layer are fully connected. Using the previous formula we obtain 138 parameters. In this project each training sample is made of a matrix 88 x 160 x 16, which means that the input layer used has 225280 nodes. Since the architecture of the network is made of several layers, a fully-connected neural network need to train a number of parameters on the order of hundreds of millions. Another deficiency of using a FNN on images is that it has no built-in invariance with respect to translation, or local distortion of the input[26]

To solve this problem, the Convolutional Neural Network (CNN) model is employed. These models are typically used on images, and more in general multi-channel images. The CNN architecture are made of combination of three types of layers: the convolutional layers, the pooling layers, and the fully connected layers.

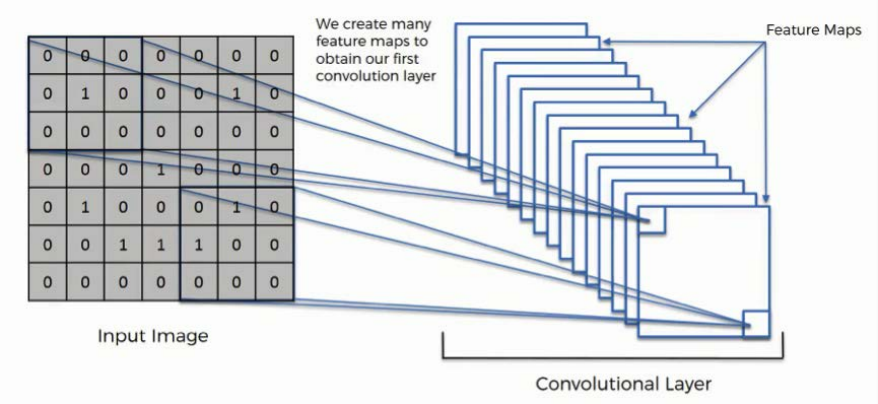


Figure 2.5: Convolutional Layer[1]

From the figure it is possible to see that instead of connecting each neuron of a layer to each neuron of the next layer,in the convolutional layer only a portion of the input is used. The key component is the kernel, i.e. a matrix of weights with size m x m x d, where m is the length of convolution and d is the number of filters. Typically the size of the kernel is small, but it spreads along the depth of the input. Starting from the top left corner of the input we compute the convolution between the kernel and the image for each filter in the kernel, thus we obtain as output a 1 x 1 x d vector.

The kernel is then moved along the image and another convolution is computed: this process is repeated for the whole image. The output of all the convolutions of a single filter is called feature map. The feature map of all the filter are then stacked together to create the input of the next layer. The value $z^l_{j,k,f}$ where (j,k) is the location of the feature on the f-th feature map of the l-th layer is equal to

$$z^l_{j,k,f} = (\mathbf{w}^l_f)^T \mathbf{x}^l_{j,k} + b^l_f,$$

where $(\mathbf{w}^l_f)^T$ and b^l_f are the weights vector and the bias of the f-th filter of the l-th layer. An important aspect is that for a feature map the same kernel and biases are used for the convolutions. With multiple feature maps, we have the opportunity to detect multiple local feature regardless of their position on the image [27]. Kernels permit reducing the number

of parameters that must be trained, i.e. we have only m x m x d weights and d bias to train. Some parameters that are used in the convolutional layer are:

- Receptive field m: the size of the kernel used in the convolution, typically it is an odd number.
- Depth d: number of filters of the kernel, it determines the number of feature maps that will become the input of the next layer
- Stride s: it indicates by how many pixels the kernel move after the convolution. If the value is less than m then we are overlapping the convolution, if it is greater or equal to m there is no overlapping.
- Zero-padding p: how many pixels are padded in to the input. Since the combination of receptive field and stride reduce the size of the output, the padding is used to modify the output size.

We can compute the output size of a convolution of a single filter with applied to an image of size n x n:

$$(n - (m - 1) + 1) + 2 * p)/s + 1.$$

The number of parameters we have to train with receptive field of size m x m and depth $(m * m + 1) * d$.

Like for FNN, the output of each convolution is then processed by an activation function. It can be one of the previously described, but the most used is ReLU..Next we apply the pooling-layer, this layer aims at reducing the dimensionality of the output, In this way we further reduce the number of parameters and the computational complexity of the architecture[28]. Another advantage is that whenever we find a feature, its precise position is irrelevant for identifying the pattern, sometimes it can be harmful because the position can change for different samples. If the network learns to find a feature in a very specific position, it may be unable to find it when it is in a slightly different position [26], by downsampling the output with the pooling layer we avoid this problem.

Two common pooling layer are the max pooling-layer and the mean pooling-layer. Just like the convolutional layer we are using a kernel to compute the operation: the receptive field size is 2x2 and it has a stride of value 2: this mean that we do not have any overlapping and at the end the size of the feature map will be halved. In a max pooling-layer we keep the maximum value inside the receptive field, while for the mean pooling layer we average the values.

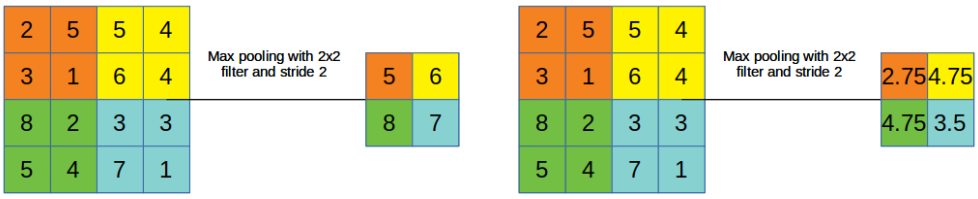


Figure 2.6: Pooling Layer

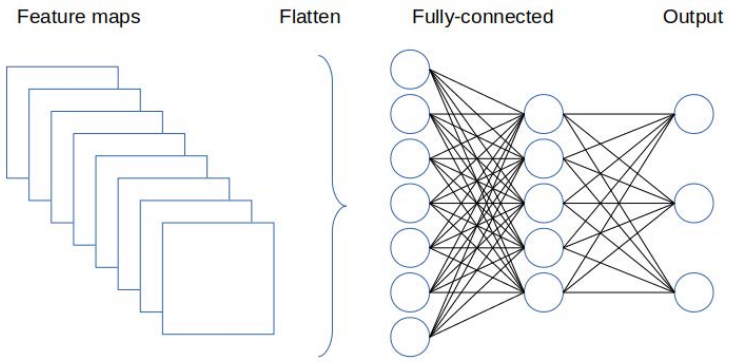


Figure 2.7: Convolutional Neural Network

Before the output layer we have one or more fully-connected layers.

There are several techniques that are used to improve the performance of a CNN; one of these is the batch normalization. The training of a neural network is very difficult, the inputs of each layer are affected by the parameters of the preceding layers; this means that small changes to the network parameters amplifies as the network becomes deeper[29]. The changes of the distribution of layers' input is a problem, and when it happens it is called covariate shift of the layer[30]. Ioffe and Szegedy in[29] propose the batch normalization as a solution for the covariate shift of the layers. It accomplishes it with a normalization of the layer inputs where the mean and the variance of the input are fixed. This affects the gradient flow through the network since it reduces the influence of the gradients on the scale of parameters. Using this it is possible to use a higher learning rate thus increasing the learning speed. Let x be the input of a layer, and X the set of these input, the normalization can be written as

$$\hat{x} = norm(x, X).$$

The normalization depends not only on the single input x but also on all examples X. The

backpropagation requires the computation of the Jacobians:

$$\frac{\partial norm(x, X)}{\partial x} \quad , \quad \frac{\partial norm(x, X)}{\partial X}.$$

If our training set contains a lot of samples it is computationally expensive to compute the covariance matrix $Cov[x] = E_{x \in X}[xx^T] - E[x]E[x^T]$, its inverse square root, and all the derivatives of these transformation used for the backpropagation. To simplify the computation, authors of [29] propose two strategies.

The first , instead of normalizing the features in layer inputs and outputs jointly, normalizes each scalar feature independently. For a layer with input $x = (x^{(1)}, x^{(2)}, ..., x^{(d)})$ each dimension is normalized as:

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}},$$

where the expectation and variance are computed over the training samples. This poses a problem: changing each input layer may change what the layer can represent. To avoid this, a transformation is inserted in the network that can represent the identity function. For each $x^{(k)}$ a pair of parameters, $\gamma^{(k)}$, $\beta^{(k)}$, is associated which scale and shift the normalized value.

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}.$$

These parameters are learned during the training phase of the network, it is possible to recover the original input by setting $\gamma^{(k)} = \sqrt{Var[x^{(k)}]}$ and $\beta^{(k)} = E[x^{(k)}]$

Since the gradient descent algorithm uses mini-batches of samples, each mini-batch produces an estimate of the variance and the mean of the input instead of the whole training set. The normalization of the input using mini-batches is an efficient way to improve the training phase, but it is not desirable during the inference phase[29]. For this reason, once we have trained our model, we use the following normalization:

$$\hat{x} = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}},$$

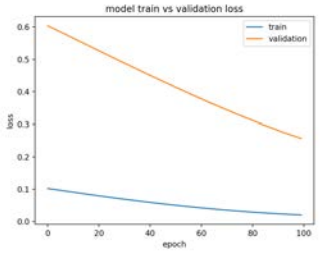
where we compute the mean and variance of the whole input set,ϵ is parameter set close to 0 to avoid having 0 at the denominator.

While we are training our model, it is possible to fall in two conditions: underfitting or overfitting. In the former case our model fails at learning from the training set and it per-

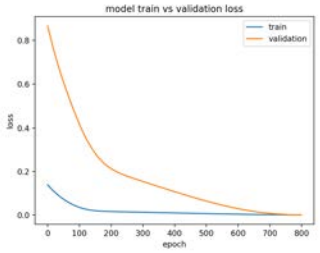
form poorly on a test dataset. A possible solution is to increase the capacity of the model by increasing the number of layers and/or increasing the number of nodes for each layer.

In the second case our model learns to well the training set and does not generalize. There are several approaches to mitigate this problem: we can train the model using more training samples or change the architecture reducing its capacity.

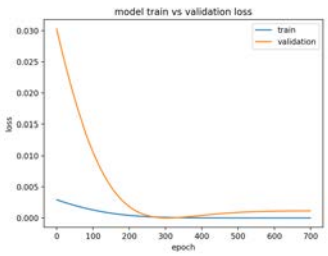
Comparing the error over the training set with the error of the validation makes is possible to recognize an overfitting.



(a) Underfitting.The Validation loss is greater than the training loss, but it is decreasing. It needs more iteration to reach the optimum.



(b) Good fitting. The training loss is very close to the validation loss, if we keep training the neural network it is possible to overfit the model.



(c) Overfitting. The validation loss gets close to the training loss, but then it increases

Figure 2.8: Plots of the loss over the training set and the validation set[2]

Sometimes the minimization of free parameters is not wanted or not needed, for this reason regularization techniques are used.[31][23]

Two regularization techniques that will be used in the model for the face forgery detection are the l_p norm regularization and the dropout.

The l_p norm regularization works by changing the cost function with the addition of the regularization term that penalizes the model complexity. If our initial cost function is $C(\mathbf{w}, \mathbf{b})$, the regularized function becomes $L(\mathbf{w}, \mathbf{b}) = C(\mathbf{w}, \mathbf{b}) + \lambda R(\mathbf{w}, \mathbf{b})$ with $R(\mathbf{w})$ our regularization term and λ our regularization strength. Typically l_p norm regularization use $R(\mathbf{w}, \mathbf{b}) = \sum_j ||w_j||_p^p$ with p=2: the l_2 norm regularizer is commonly called weight decay. This regularization tries to force the network to use smaller weights rather than big ones, this compromise is regulated by the value of λ. It is possible to choose a l_1 norm regularizer, where we compute the absolute value of the weights. It still penalizes the large weights, but by using it the weight shrinks by a constant amount toward 0, while for the l_2 it is proportional to $|w|$.

Another type of regularization is the dropout. The term refers to dropping out unit in

a neural network [32]. We do not remove permanently the neuron from the network, but temporarily strip it and all the related connections off from the network. The choice for a neuron to be dropped is random, typically with a probability p independent from the other neuron.

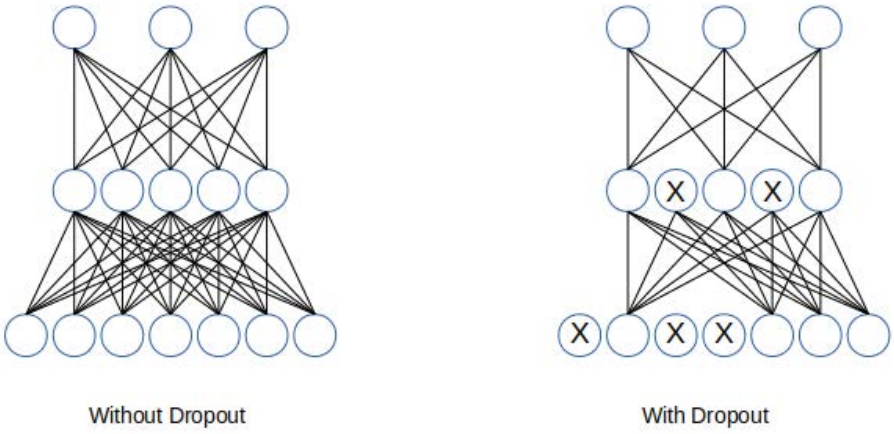


Figure 2.9: Dropout in a neural network

Since at each iteration of the training different nodes are dropped out, we are practically training different architectures. One motivation on why dropout improves the training is that each neuron in a network trained with dropout must learn to work with a random set of neuron. This should make the model more robust and able to create useful features.

An important decision must be made about the initial value of the weights at the beginning of the training. The first type of initialization set the value of the weights following a Gaussian distribution with zero mean and variance equal to 0.01; the bias was set to one for some layer[33]. There can be some issues regarding the activation and/or gradient magnitude of the final layer. If the layers are not properly initialized, their input is rescaled by k: if we have a network with L layers, at the last layer the input is scaled by k^L [23][33]. Values of k>1 lead to extremely large value of output layer while k<1 leads to diminishing signal and gradient. Additionally, large initial values for the weights lead to divergence due to large updates; on the contrary, with small initial weights, the network is not able to learn since the updates are in the order of 0.0001% [33].

Two popular weight initialization are the Xavier initialization and the He initialization. The former is proposed by Glorot and Bengio in [34], the weight $w_{i,j}$ of each layer is initial-

ized with the following distribution

$$w_{i,j} \sim U[-\frac{\sqrt{6}}{\sqrt{n_i + n_o}}; \frac{\sqrt{6}}{\sqrt{n_i + n_o}}],$$

where U is the uniform distribution, n_i is the number of neuron of the previous layer, and n_o the number of neuron of the layer we are at the moment. We can initialize the weight by drawing the values from a Gaussian distribution with zero mean and variance $var = \sqrt{\frac{2}{n_i+n_o}}$. Its derivation is based on the assumption that the activation are linear, this assumption is invalid if we use the ReLU activation. The work in [35] proposes a weight initialization that address non linearities, the He initializer, that allows for extremely deep model to converge, while the Xavier method cannot. With this initialization, we select the value for the weight from a Gaussian distribution with variance $\frac{2}{n_i}$.

The last improvement to increase the efficiency of a neural network is about the optimization of the gradient descend. It is known[36] that gradient descend is not suitable for optimizing objectives that present pathological curvature. The 2nd-order optimization method has been demonstrated to be quite successful on such objectives. For this reason it is possible that deep learning problem could be resolved using those techniques. An example of pathological curvature is shown in Fig 2.10. The problem of this graph is not the presence of low or high curvature directions, but the mixture of both together.

One method to overcome the pathological curvature is the Adam optimization. Its name derived from Adaptive Moment Estimation (Adam); it only requires first-order gradient with little memory requirement and it computes individual adaptive learning rates for different parameters from estimates of first and second moment of the gradients.[37] It is a combination of two previous optimization methods, the RMSProp and Adagard. The advantage of Adam are that the magnitude of the parameters' update is invariant to rescaling of the gradient, the stepsizes are approximately bounded by the stepsizes of hyper-parameters, a stationary objective is not required, the training works with sparse gradient, and a form of step size annealing is naturally performed.

Given an objective function $f(\theta)$ with parameters θ, the sequence of operation of Adam to minimize the expected value $E[f(\theta)]$ are

Compute the gradient at timestamp t $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$

Update biased first moment estimate $m_t \leftarrow \mathrm{B}_1 \cdot m_{t-1} + (1 - \mathrm{B}) \cdot g_t$

Update biased second raw moment estimate $v_t \leftarrow \mathrm{B}_2 \cdot v_{t-1} + (1 - \mathrm{B}_2) \cdot g_t^2$

Compute bias-correct first moment estimate $\hat{m}_t \leftarrow \frac{m_t}{1-\mathrm{B}_1^t}$

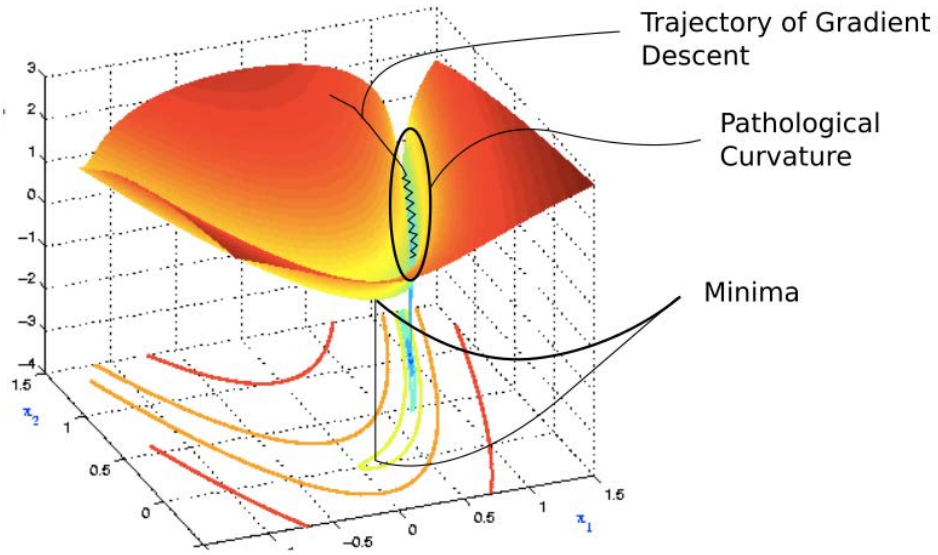


Figure 2.10: Example of a pathological curvature[3]

Compute bias-correct second raw moment estimate $\hat{v}_t \leftarrow \frac{v_t}{1-\mathrm{B}_2^t}$

Update parameters $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t}+\epsilon}$

Fig. 2.11 shows that Adam, Adagard, and RMSProp, with dropout converge faster than the SGDM using the Nastarov momentum; moreover the Adam algorithm is the fastest.

2.3 Autoencoders

Another type of neural network that is useful for our work is the autoencoder. Autoencoders are simple learning circuits which aim to transform inputs into outputs with the minimum amount of distortion[38]; in other words it is a neural networks that is trained to attempt to copy its input to its output[25].

The first idea of an autoencoder was done by Hinton and the PDP group to address the problem of the backpropagation without a teacher.

Typically, an autoencoder will learn from the data without the use of human labeling; for this reason they can be classified as unsupervised learning algorithm.

The model can be seen as the combination of two parts, the encoder and the decoder. The encoder maps the input to a code represented by a hidden layer $h = f(x)$, and the decoder

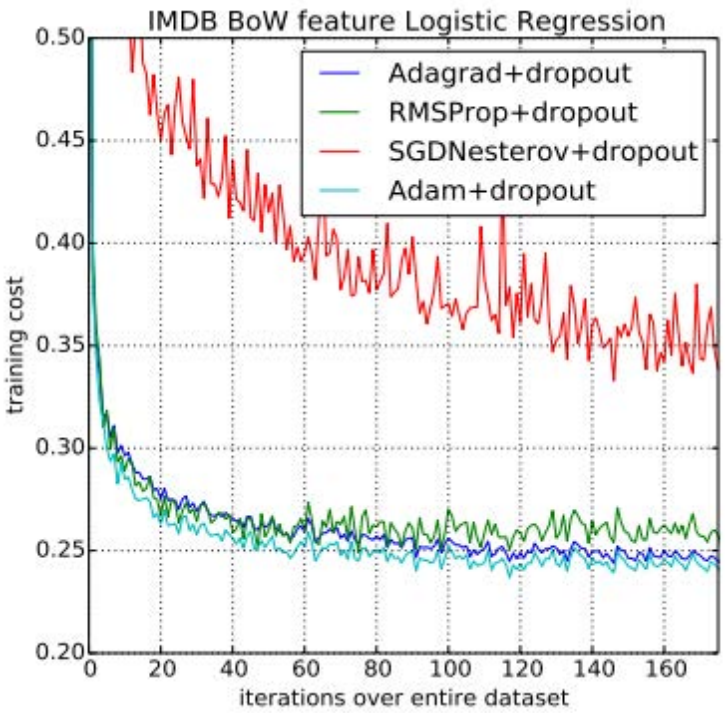


Figure 2.11: Logistic regression training negative log likelihood on IMDB movie reviews with 10,000 bag-of-words (BoW) feature vectors.[37]

maps the code to the output $\hat{x} = g(h) = g(f(x))$.

If the autoencoder learns to set $g(f(x)) = x$ everywhere is not useful since it simply copies the input to the output. What we want is to extract from the input useful properties of the data. We achieve that with some constraint applied to the architecture, one way is to have h with a smaller dimension of x, in this case the autoencoder is called undercomplete[25].

A loss function that can be used to measure the difference between the input x and the output recovered from the input x,$\hat{x}$, is the MSE.

$$L(x, \hat{x}) = \frac{1}{m} \sum_{i=1}^{m} (x_i - \hat{x}_i),$$

where m is the size of the output.

If the dimension of the hidden layer is greater than the input size, it is possible for the autoencoder to simply learn the identity function. To obtain some useful results we need to add some constraints. These can be in the form of regularization, for instance to ensure sparsity of the hidden layer representation, or restriction on the classes of function of the

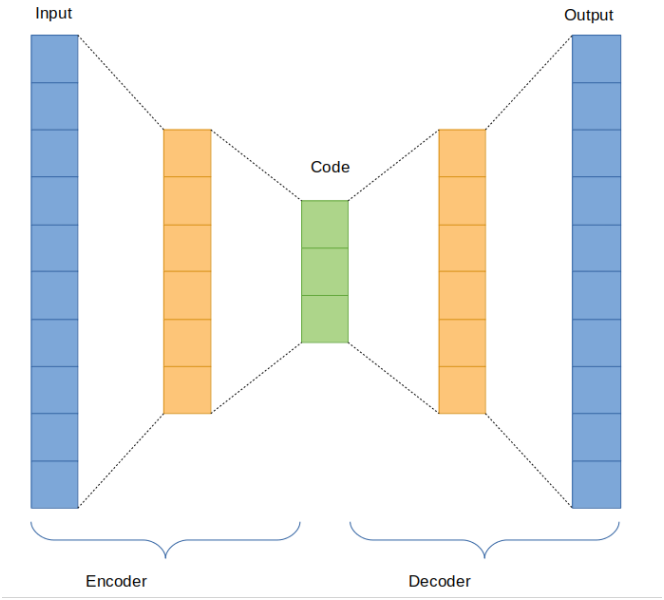


Figure 2.12: Autoencoder architecture

encoder and the decoder, or noise in the hidden layer. These constraints force the hidden layer to assume k different value, when $k < m$ the autoencoder tries to implement some form of compression or feature extraction[38].

3

Generative Adversarial Networks

The architecture used for the face forgery detection is based on the Generative Adversarial Network (GAN) architecture.

In deep learning discriminative models have been widely employed in many classification and estimation problems. The success of these solutions is primarily based on the computing and estimation efficiencies of the previously mentioned solutions, such as the backpropagation and the dropout.[39] Deep generative models are less fortunate, due to the difficulties in approximating maximum likelihood estimation and related strategies. Despite these results, there are several reason to study them[40]:

- Training and sampling from generative models is an excellent test of our ability to represent and manipulate high-dimensionality probability distributions.
- They can be incorporated into reinforcement learning in several ways. Reinforcement learning algorithms can be divided in model-based and model-free algorithm; the former one contains the generative models. Generative models of time-series data can be used to simulate possible future data. A generative model enables learning in a simulated environment, where mistaken action does not cause harm to the agent.
- Generative models can be trained with missing data and can provide predictions whenever some data are missing. One example of these is the semi-supervised learning, where the labels are not available.
- They also enable classification with multi-modal outputs. Sometimes a single input may have different correct answers, each of them acceptable. With a traditional ma-

chine learning algorithms, which uses, for example, the mean squared error between the model's output and the desired one we are unable to train models that can produce multiple different correct outputs.

GANs, and other generative models, perform a maximum likelihood estimation[40]. The core idea of maximum likelihood is to define a model that provides an estimate of a probability distribution, parametrized with parameter θ. The likelihood is then the probability that the model assigns to the training data: $\prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \theta)$, given m samples $X^{(i)}$. The maximum likelihood wants to find the parameter of the model that maximizes it. To simplify the computation of the derivatives and reduce computational load, we will do the maximum likelihood in the log space, where the products turn into sums.

$$\begin{aligned}\theta^* &= arg\max_\theta \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \theta) \\ &= arg\max_\theta log \prod_{i=1}^{m} p_{model}(\mathbf{x}^{(i)}; \theta) \\ &= arg\max_\theta \sum_{i=1}^{m} log p_{model}(\mathbf{x}^{(i)}; \theta).\end{aligned}$$

The maximum likelihood estimation can be seen as the minimization of the Kullback Leibler (KL) divergence between the data generating distribution and the model.

$$\theta^* = arg\min_\theta D_{KL}(p_{data}(\mathbf{x})||p_{model}(\mathbf{x}; \theta).$$

Unfortunately we do not have p_{data} but only a training set of m samples taken from p_{data}. These values are used to define an empirical distribution, $\hat{p}_{data}$. The action of minimizing the KL divergence between $\hat{p}_{data}$ and p_{model} is the same of maximizing the log-likelihood of the training set[40].

In [39] it is proposed an adversarial network where two models are trained one against the other. More precisely, a generative model is trained to fool a discriminator, which learns to determine if the sample in input is from the generator or the data distribution. Competition in this game forces both models to improve their method until the generator output is indistinguishable from the true data sample.

It is possible to train both models using backpropagation and dropout algorithm, without the need of using Markov chain. Moreover a variety of factors and interaction can easily be incorporated in the model[41].

From [41], we want to learn the generator's distribution p_g over data x. An input noise variable $p_z(z)$ is defined, along with a mapping to data space as $G(z;\theta_g)$, where G is multilayer perceptron with parameters θ_g that represents a differentiable function. Another multilayer perceptron is defined, $D(x,\theta_d)$ that outputs a single scalar. $D(x)$ represent the probability that x belongs to the data rather than to p_g.

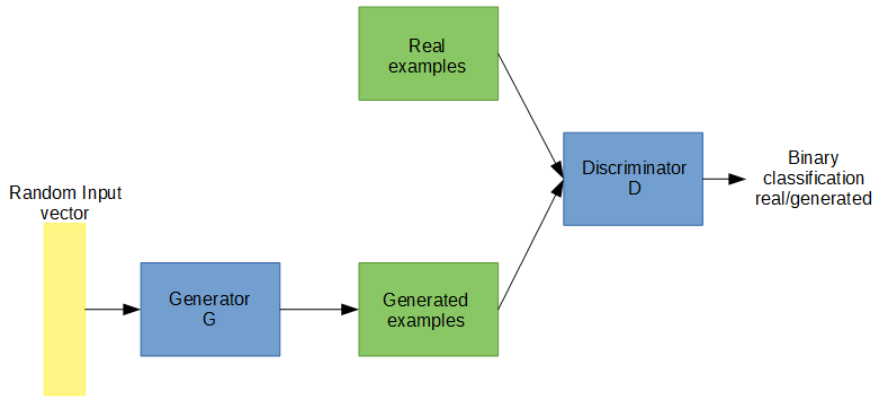


Figure 3.1: GAN architecture

The goal is to train $D(x)$ to maximize the probability of correctly labeling the data sample and the samples generated by G. G on the contrary wants to minimize $log(1-D(G(z))$. In other words,[39] D and G play the following two-player minimax game with value function

$$\min_G \max_D (V(D,G)) = \mathbb{E}_{x\sim p_{data}(x)}[logD(x)] + \mathbb{E}_z[log(1-(D(G(z))].$$

Unfortunately, the training of GANs can be difficult when G and D are represented by neural networks and $\max_D V(D,G)$ is not convex. Simultaneous gradient descent on two players' costs is not guaranteed to reach an equilibrium. If we consider the following example where the value function $v(a,b) = ab$ with player 1 controlling a having payoff ab, and player 2 controlling b and having payoff $-ab$. If each player make infinitesimally small gradient steps, each player reduce their own payoff at the expense of the other player. Thus, parameters a and b go into a stable circular orbit rather than arriving at the equilibrium point at the origin.

The equilibria point for a minimax game are not local minima of v; they are points that are simultaneously minima for both players' costs. They are saddle point of v that are local minima with respect to player 1's parameters and local maxima with respect to player 2's parameters. It can happen that the two players take turns increasing and decreasing the value

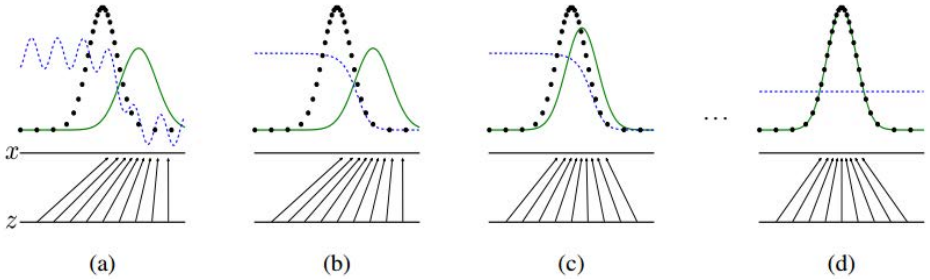


Figure 3.2: Generative adversarial nets are trained by simultaneously updating the discriminative distribution (D, blue, dashed line) so that it discriminates between samples from the data generating distribution (black,dotted line) p_x from those of the generative distribution p_g (G) (green, solid line). The lower horizontal line is the domain from which z is sampled, in this case uniformly. The horizontal line above is part of the domain of $\mathbf{x}$. The upward arrows show how the mapping $\mathbf{x}$ = G(z) imposes the non-uniform distribution p_g on transformed samples. G contracts in regions of high density and expands in regions of low density of p_g. (a) Consider an adversarial pair near convergence: p_g is similar to p_{data} and D is a partially accurate classifier. (b) In the inner loop of the algorithm D is trained to discriminate samples from data, converging to $D^*(x) = \frac{p_{data}}{p_{data}+p_g}$. (c) After an update to G, gradient of D has guided G(z) to flow to regions that are more likely to be classified as data. (d) After several steps of training, if G and D have enough capacity, they will reach a point at which both cannot improve because $p_g = p_{data}$. The discriminator is unable to differentiate between the two distributions, i.e. D(x) = 1/2[39]

function forever, instead of landing exactly on the saddle point where neither player is capable of reducing its cost [42].

Training GANs consists on finding a Nash equilibrium to a two player non-cooperative game [43]. Each player want to minimize its own cost function, for the discriminator we have $C_D(\theta_G, \theta_D)$, and it can control only $\theta_{\mathbf{D}}$, while for the generator we have $C_G(\theta_G, \theta_D)$, with control over $\theta_{\mathbf{G}}$.

The Nash equilibrium is a tuple (θ_G, θ_D) that is a local minimum of C_D with respect to θ_D and a local minimum of C_G with respect to θ_G[40]

The cost function used for the discriminator is:

$$C_D(\theta_{\mathbf{G}}, \theta_{\mathbf{D}}) = -\frac{1}{2}\mathbb{E}_{x\sim p_{data}(x)}[logD(x)] + \mathbb{E}_z[log(1 - (D(G(z))].$$

where we used a cross-entropy cost that is minimized when training a standard binary classifier with a sigmoid output. To consider also the generator we can represent in different ways the game. In the first version it is represented as the simultaneous train of a two player min-max game where the sum of the two players' costs is always 0.

$$C_G = -C_D.$$

Since C_G can be written as the negative of C_D we can summarize the entire game using a value function V(G,D) specifying the discriminator's payoff:

$$V(G, D) = -C_D(\theta_G, \theta_D).$$

In the value function we have omitted the parameters θ_G and θ_D for simplicity, we know that they are parameters only of the generator and discriminator respectively. The solution of the minimax game involves a maximization in the inner loop and a minimization in the outer loop

$$\min_G \max_D V(G, D) = \mathbb{E}_{x \sim p_{data}(x)}[logD(x)] + \mathbb{E}_{z \sim p_z(z)}[log(1 - D(G(z)))].$$

To achieve the optimal results we need to implement the game with an iterative approach. If we try to completely optimize D in the inner loop, we could obtain an overfitting. For this reason we compute k iterations of optimization of D and then one step of iteration of G. If we do so, D is maintained near its optimal solution as long as G changes slowly.

The generator G implicitly defines a probability distribution p_g as the distribution of the samples $G(z)$ obtained when $z \approx p_z$[39]. Our goal is to model G in order to have a good estimation of p_{data}.

If we have a fixed generator G, then our optimal discriminator D is:

$$D^*_G = \frac{p_{data}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_g(\mathbf{x})}.$$

The minimax game is not very effective, since the generator wants to minimize a cross-entropy and the generator wants to maximize it, if the discriminator successfully rejects generator samples with high confidence, then the gradient of the generator vanishes,[40][39] and it is unable to learn and therefore it cannot fool the discriminator. To avoid this problem, instead of flipping the sign on the discriminator's cost function to obtain the cost function of the generator, we flip the target used to construct the cross-entropy cost of the generator[40]:

$$C_G = -\frac{1}{2}\mathbb{E}_z log(D(G(z))).$$

As we can see, now the generator wants to maximize the probability of the discriminator being mistaken. With this variation of the game both player have a strong gradient when they are losing the game. Since we are no more in a minimax game we cannot describe it

with a single value function.

Some disadvantages of an adversarial network are[40][44]: there is no explicit representation of $p_g(x)$, the generator and discriminator may oscillate rather then converging, and D must be synchronized well with G during the training, otherwise we fall into the Helvetica Scenario.

3.1 Deep Convolutional Generative Adversarial Networks

For the GAN used in this project we want our generator to output an image, [45], shows how to build good image representation using GANs. It is possible to separate in two categories generative image models: non-parametric and parametric. In the former category we find models that do matching from a database of existing images. They are used in super-resolution problems, in-painting, and texture synthesis.

Parametric models for generating images has been studied extensively, however generating images of the real world have had not much success.

To create a GAN able to generate good image representation the following approaches are used.

- Replace the deterministic spatial pooling functions, like the maxpooling, with strided convolutions; this allow the network to learn its own spatial downsampling.
- Eliminate the fully connected layer on top of convolutional features.[45] found that global average pooling increases model stability at the expense of convergence speed.
- Lastly is the use of batch normalization, which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance. The batch normalization helps the gradient flow in deeper models and also improve the training in the case of poor initialization.

The batch normalization is not applied in each layer, otherwise we could have sample oscillations and model instability. It is applied to each layer but the generator output and the discriminator input. As for the activation function,the ReLU activation is used in all the layers of the generator expect the last one, where in our case a sigmoid activation function is used. For the discriminator, a leakyReLU activation function is used, this function is preferred to a normal ReLU since it does not have discontinuity.

The last topic relevant to our model that we are going to elaborate on is the conditional GAN [41]. A cGAN is an extension of a traditional GAN where both the generator and discriminator are conditioned on some extra information $\mathbf{y}$[41]. The extra information y is an embedding space used to condition the generative model on some external information taken from the training sample; with this it is possible to define a density model $p_y(y)$.

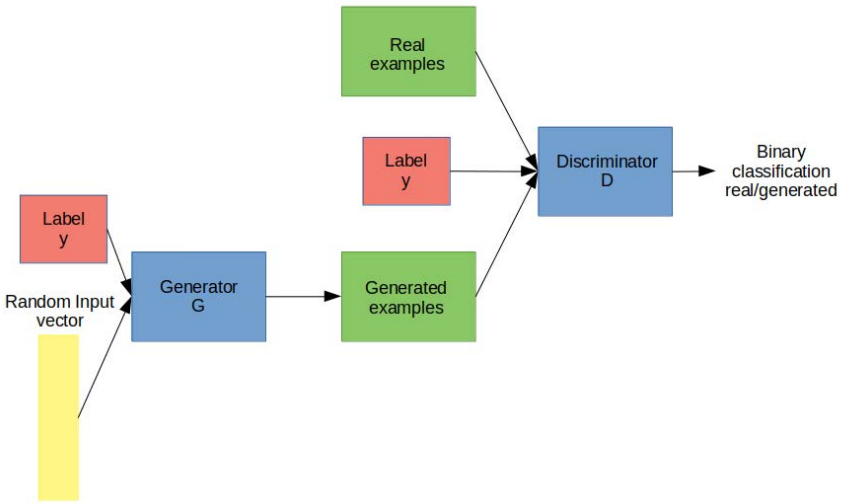


Figure 3.3: Conditional GAN

The generator implicitly defines a conditional density model $p_g(x|y)$; it is possible to combine it with the existing conditional density $p_y(\mathbf{y})$ yielding the joint model $p_g(x, y)$[46].

The objective function would be

$$\min_G \max_D (\mathbb{E}_{x,y\sim p_{data}(x,y)}[logD(x,y)] + \mathbb{E}_{y\sim p_y, z\sim p_z(z)}[log(1-(D(G(z,y)),y)].$$

We can notice that both terms involves some conditional data $\mathbf{y}$ sampled from an independent distribution or the training sample, additionally the second term is an expectation over two independent random variables: $\mathbf{z}$ and $\mathbf{y}$.

If we have a batch of n training samples x_i paired with conditional data y_i, and $z_i \sim p_z(z)$ is a noise data taken from the noise distribution p_z, we can write the cost function of the

discriminator using the logistic cost expression as:

$$C_D = -\frac{1}{2n}(\sum_{i=1}^{n} logD(x_i, y_i) + \sum_{i=1}^{n} log(1 - D(G(z_i, y_i), y_i))),$$

where the discriminator assigns a positive label to true pair (x_i, y_i), and negative label to generated pair $G(z_i, y_i)$

For the cost function of the generator we have:

$$C_G = -\frac{1}{n}\sum_{i=1}^{n} log(D(G(z_i, y_y), y_i)).$$

4

Architecture

In this chapter we will present the architecture used to detect and localize face forgeries in video sequences.

Before using the dataset for the training, a pre-processing has been used to extract different features from the frames of the video sequence that is analyzed in order to detect whether it contains deepfake face or not. Each feature is computed on overlapping or non-overlapping pixel blocks of size 64 x 64 or 16 x 16, depending on the feature, leading to sequences of 88 x 160 values. Since the algorithm computes 16 features for each frame, the neural network processes samples of size 88 x 160 x 16. The frames in the dataset have a resolution of 1280 x 720 or 1920 x 1080 pixels(with 3 channels for the colour component).

Feature computation is performed using an ad-hoc MATLAB script. On each image block, the script computes some feature values, which will be described in the following. The different features are stacked together and will create the input feature sequence that will be processed by the following conditional GAN.

- The first 8 features are the result of two neural networks described in [14] to extract information about the video codec and coding quality used on video. A discrepancy in the output of these two networks over the duration of the video permits detecting the presence of a temporal splicing. Among these initial features, the first 4 represent the probability for each pixel to have been compressed with one of the codecs* in a

*A codec (combination of the words coder and decoder), is a computer program used to encode or decode a digital data stream or signal.

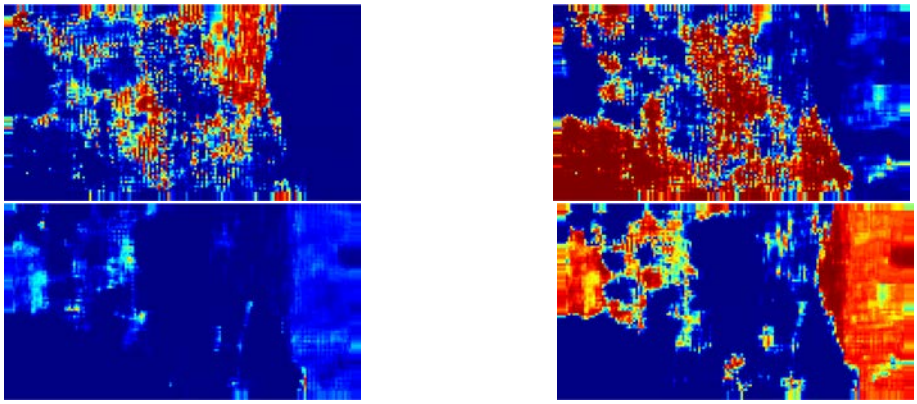


Figure 4.1: Codec features

training set. To create these features, a pre-trained neural network has been used: this takes in input a frame and a value for the patch stride, and works on patches of size 64 x 64 pixels. With a patch stride of 8 pixels, we obtain as output a frame of size 83 x 153 x 4 if the frame is of size 720 x 1280; the output has size 128 x 233 x 4 if the frame has a resolution of 1080 x 1920 pixels. The codecs this neural network is able to distinguish are H264, H265, MPEG2, MPEG4. Since each output value represent a probability, it is in the interval [0,1].

The other 4 features represent the probability for each pixel with respect to a particular compression quality: low, mid-low, mid-high, high. A similar pre-trained network has been used to extract these features. As before the output values are in the interval [0,1] since they are probabilities. To train this network, five uncompressed videos are encoded using the FFmpeg library to obtain 60 versions combining the aforementioned codecs and quality. The quality level is identified by the quantization step Δ. However, in the majority of the codecs it is not possible to control directly the quantization step, instead a high-level quality parameter q is used, and the relationship between q and Δ depends on the codec used.

$$\Delta_{H264} = \frac{5}{8} \cdot 2^{\frac{q}{6}}$$

$$\Delta_{MPEG} = \begin{cases} 8 & 1 \leq q \leq 4 \\ 2q & 5 \leq q \leq 8 \\ q + 8 & 9 \leq q \leq 24 \\ 2q - 16 & 25 \leq q \leq 31 \end{cases}.$$

The original paper used these features as a solution to the video splicing detection problem. That is to detecting if video sequence is a composition of at least two videos, or it is a single original video, based only on pixel level analysis.

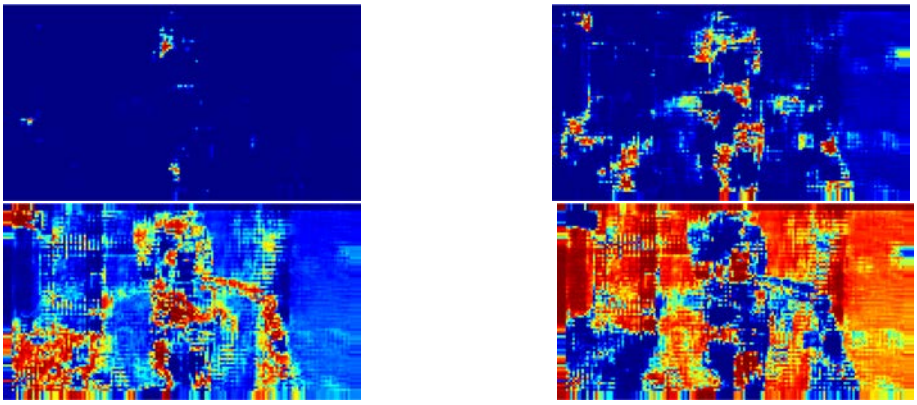


Figure 4.2: Quality features

- Feature 9 depends on the difference between the current frame and the next one. The MSE computed from the pixel wise difference is calculated for patches of size 8x8 on each channel separately (with no overlapping). Since we have 3 channels for each frame the results is combined in a single matrix. The choice to use MSE over a simple mean is to avoid to obtain zero as result due to cancellation of the noise. This feature is useful to find if a region of the frame has been reused in the next frame. Since the same value are used in two consecutive frame in a specific region, we expect that the result of the MSE for that region to be close to 0. To scale the values of the resulting matrix in the interval [0,1], we compute the square root of the matrix, then we divide it by the square root of the maximum achievable value, considering that we are working on pixels in the range [0,255] we have $\sqrt{255^2 * 3}$.

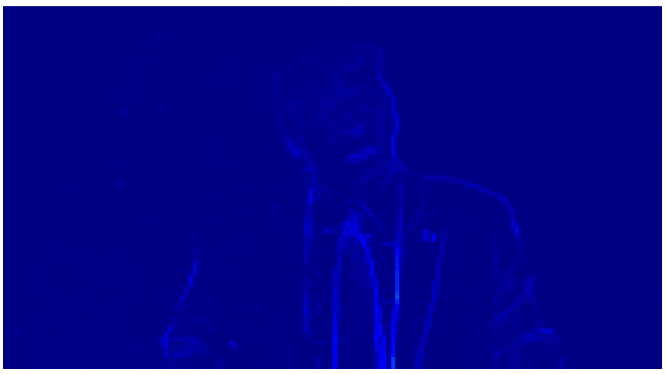


Figure 4.3: Frame difference

- Features 10 and 11 are obtained by applying the optical flow on two consecutive frames. The optical flow is the distribution of apparent pixel velocities of brightness pattern in an image. The optical flow can arise from relative motion of objects with respect

to the viewer [47]. It can give information about the spatial position of the object viewed and the rate of change of this position. The optical flow is extracted using the Matlab function $opticalflowLK$. This function uses the Lucas-Kanade method to estimate the optical flow. The outputs of the optical flow are two matrices: the first one contains the magnitude of the optical flow flow with respect to the x axis; the second matrix is the same but with respect to the y axis. The values are scaled to be in the interval [0,1].

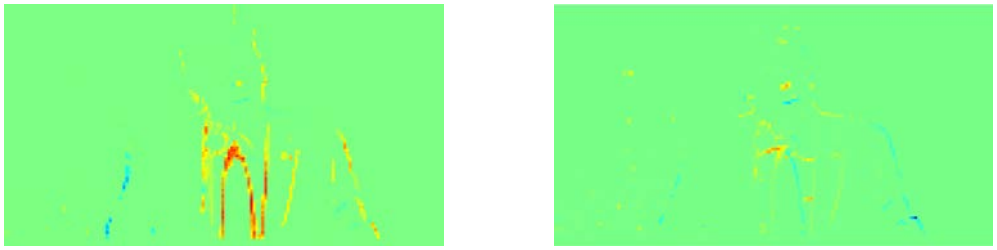


Figure 4.4: Optical flow features

The study of optical flow was already used to detect deepfake videos in [48]. In that paper, the optical flow is extracted using a CNN model and given as input to a semi-trainable network based on some pre-trained network, the VGG16 and the ResNet50.

- Feature 12 is a mask of 0 and 1 that shows the region where a forgery of a face has been applied. This feature is extracted by using the Viola-Jones algorithm of Matlab to detect faces in a frame. The result of this algorithm on frames where the face is not modified are ignored. The mask obtained using Viola-Jones is rectangular, to avoid the neural network from learning this shape a chain of erosion and dilation are used. More precisely we alternate an erosion with a dilation three times to obtain the new mask. The structuring element of the erosion is a square whose width is 8 pixel; for the dilation we use as structuring element a disk with radius of 4 pixels.

  Obviously this feature is only used during the training phase of the neural network: it is the conditional information of our cGAN.

- Feature 13 represents the macroblock type used to encode the frame, it is obtained by using ffmpeg with the option -debug vis_mb_type. The decoding of macroblock types is MPEG-specific, so it is not possible to apply it to all videos. It works on patches of size 16 x 16 and for each of them a symbol is assigned depending on a macroblock type condition. The symbol is then converted in an integer value between 0 and 5 with a Matlab script. The meaning of each value is the following:

  – 0, A video that is not compatible with the ffmpeg command has a matrix of 0 as feature.

(a) Before erosion/dilation **(b)** After erosion/dilation

Figure 4.5: Face mask

- 1, a macroblock with this value correspond to skipped macroblock, either of P type or B type.
- 2, this value indicates an intra macroblock.
- 3, the macroblock predicted is P.
- 4, the macroblock predicted is B, with reference to a future frame.
- 5, the macroblock predicted is B, this time with a reference to a previous frame and a future frame.

The values are divided by 5 in order to have them in the interval [0,1]. Since the extraction of the macroblock works on patches 16x16 the resulting matrix has size 80 x 45; to bring it to the size required for the neural network an interpolation where the closest value is copied has been applied.

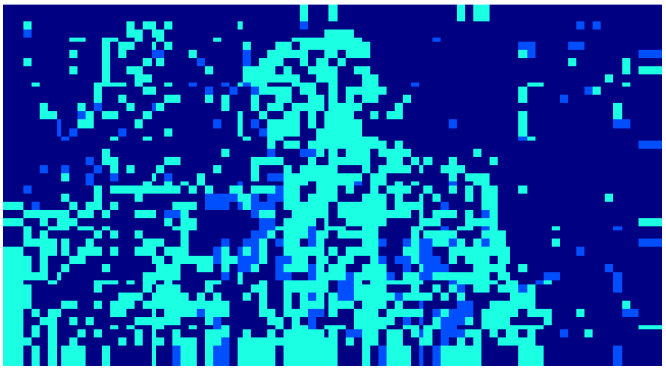


Figure 4.6: Macroblock

- The last three feature maps are based on the Benford's Law applied to the DCT component of the frame. To compute these features, each frame is saved as a JPEG and

leading digit	1	2	3	4	5	6	7	8	9
probability	0.301	0.176	0.125	0.097	0.079	0.067	0.058	0.051	0.046

Table 4.1: Probability distribution of the leading digits according to the Benford's Law

using a MATLAB script the images' coefficients are read. The JPEG image compression standard follows a block diagram which partition the image in 8x8 pixel block $\mathbf{X}$ and compute the Discrete Cosine Transform (DCT) of each block to obtain $\mathbf{Y}$[49]. The transformed coefficient are then quantized into integer-value Y_{q1}.

$$Y_{q1}(i,j) = sign(Y(i,j))round\left(\frac{|Y(i,j)|}{q_1(i,j)}\right),$$

where i,j are the indexes of the element in the block. The value $Y_{q1}(i,j)$ are then ordered following zig-zag scan starting from the top-left value increasing the spatial frequency. The quantization step $q_1(i,j)$ changes accordingly to the index (i,j) of the DCT coefficient and is typically defined by a quantization matrix. The quantization matrix adjusts the quantization step according to the spatial frequency in order to represent with a higher precision the low frequency components and quantize to zero the components at the highest frequencies. In our case we selected the maximum value, i.e. we haven't applied any quantization. If an image has been forged in a localized area and then saved again, unmodified regions will present double quantization artifacts due to the double compression of the image, while the modified region will present DCT coefficients resulting from a single compression; in fact, traces of possible previous compression are disrupted by editing or by the disalignment of the compression grid. It is possible to detect difference on the compression by using the Benford' Law.

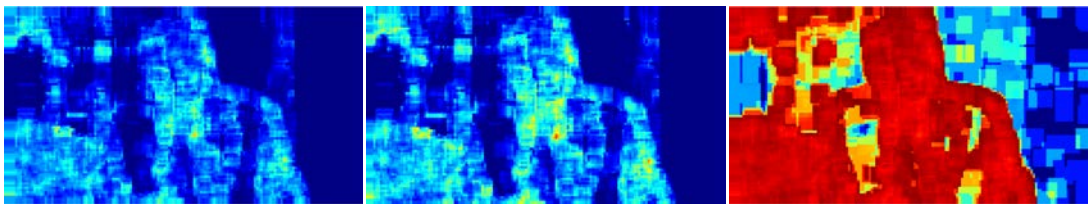
Benford Law's, or the first digit law, is an empirical law. It states that the probability distribution of the first digit in a set of natural numbers is logarithmic; more precisely we have that the significant digits of a data set satisfies the Benford's Law if they follow the distribution[50]:

$$p(x) = log_{10}\left(1+\frac{1}{x}\right), x = 1,2,..,9\,.$$

In our case we first read the syntax elements of the image, then we select a macroblock of 64x64 pixel, in this macroblock we have 8-by-8 blocks generated by the DCT. This macroblock is converted in 64 x 64 matrix where in each column we have the values of a single block. After that, for each index of frequency coefficient to be analyzed (which means, for each row of the matrix) we compute the statistic of the first digit and compare it with the fitted statistic (ideal) according to the Benford's Law, 4.1.

We compare these two statistics by computing the divergence between two arrays us-

ing the Jensen-Shannon(JS) divergence, the Renyi divergence, and the Tsallis divergence; the mean of each divergence is the respective feature for that macroblock. We continue the extraction of the feature by moving the window of the macroblock by 8 pixel each time until we analyze the whole frame. At the end for each divergence we have a matrix of size 83 x 153.



(a) Result using the JS divergence. **(b)** Result using the Renyi divergence. **(c)** Result using the Tsallis divergence.

Figure 4.7: Benford's Features

The size of the above features are not the same, for features 1 to 8, and 14 to 16, with frame of resolution of 1280 x 720, we have a size of 83 x 153, for the other features the size is 88 x 160. The reason for this difference is that since the first features are computed using a stride the last few results of each dimension are not possible to obtain, because, due to the stride, we should operate outside the frame. Because of this, we first crop all the features to the size 83 x 153 by ignoring the last value of the features of size 88 x 160; then we copy the value of the border of the feature map to reach the size 88 x 160. We opted for a copy of the border and not using a padding to avoid creating a region of discontinuity on the feature map, which could have been interpreted wrongly by the neural network. Moreover we decided to increase the size of the feature from 83 x 153 to 88 x 160 to maintain an even value of the size through the downscaling of the network.

For a video of size 1920 x 1080 the output of the pre-processing is 233 x 128, we simply crop the frame to include the face, since we are focusing on face forgery detection, to obtain a feature map of 88 x 160.

Noticeably, feature 9 is the difference of a frame with the next one, which mean that on a video of n frames we will obtain n-1 matrices, while for the other features we have the result for all the frames. For each video we discard the features obtained from the last frame except from feature 9 to have the same number of frame for each feature.

The extraction of the features is done with a MATLAB script, with a GeForce GTX 1070 graphic card a video sequence of 100 frames requires almost 50 minutes.

The model is inspired on the neural network used in [51]. The authors of the paper used a cGAN to detect if a shadow was removed from an image.

In our case, the goal is to detect a modification of a face in a video. As said before the frame is pre-processed into a matrix of size 88 x 160 x 16, for this we define a face forgery mask M of size 88 x 160, this is the feature 12 we have extracted during the pre- processing. For each coordinate (i,j) of this matrix we have that:

$$M(i,j) = \begin{cases} 1, & \text{if } (i,j) \text{ is forged} \\ 0, & \text{otherwise} \end{cases} .$$

We wish to create a mask $\hat{M}$ as an estimate of M. If $\hat{M} \approx 0$ we conclude that there was no modification and the frame is real. On the opposite if $\hat{M} \not\approx 0$ somewhere, then we can conclude that the frame has been forged and the mask visualize the region of the image that contain the forgery. The mask $\hat{M}$ is obtained through the generator of the cGAN.

The output of a discriminator is binary, 0 the input come from the real samples dataset, 1 the input come from the generator. GANs are intended to work when the discriminator estimates a ratio between two densities, but deep neural networks typically produce confident outputs that identify the correct class with a very high probability. To encourage the discriminator to estimate soft probabilities instead of extrapolating extremely confident classifications, the one-sided label smoothing technique is applied [43][40].

The idea of one-sided label smoothing is to replace the value used to identify the real input with a value slightly less than 1, for example 0.9. With this change if the discriminator learns to assign an extremely high logits corresponding to a probability approaching 1, it tries to bring the logits back to a smaller value.

We have to smooth only the labels for the real samples. If we use a label of α for the real samples and β for the fake samples, then the optimal discriminator function is:

$$D^*(\mathbf{x}) = \frac{(1-\alpha)p_{data}(\mathbf{x}) + \beta p_{model}(\mathbf{x})}{p_{data}(\mathbf{x}) + p_{model(x)}}.$$

If β is zero, then smoothing by α simply scales down the optimal value of the discriminator. When β is nonzero, the shape of the optimal discriminator function changes. Precisely, in a region where $p_{data}(\mathbf{x})$ is very small and $p_{model}(\mathbf{x})$ is larger, $D^*(\mathbf{x})$ will have a peak near the spurious mode of $p_{model}(\mathbf{x})$. Because of this the discriminator will reinforce incorrect behaviour of the generator that will be trained to create samples that resemble the data or

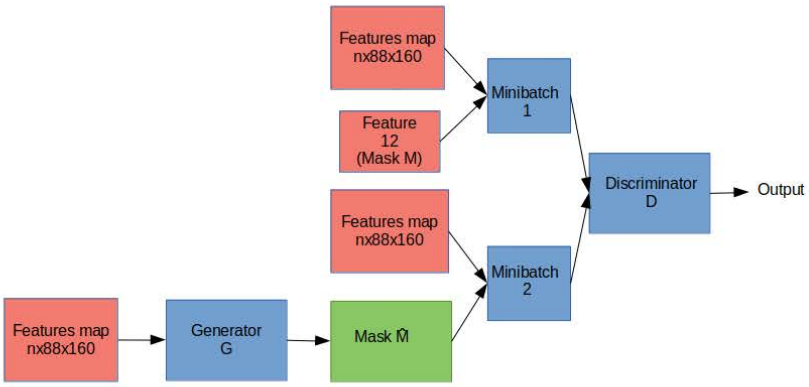


Figure 4.8: In the training of the discriminator we feed two minibatches, the first one containing the true mask, in the second the mask generated by the generator. Ideally the output of the discriminator is 0 if the the frame is forged; 0.9 if the frame is real

samples it already makes. For the training of the discriminator we need two minibatches, the first one contains all the features obtained during the pre processing; to obtain the second minibatch we remove from the features the face forgery mask and use it as the input of the generator. The generator creates a mask, which is going to be concatenated to the second minibatch. These two minibatches are fed to the input of the discriminator that need to distinguish in which minibatch there is the true face forgery mask and the mask generated from the generator. At the same time, the generator wants to output a mask that the discriminator is not able to distinguish from the true face forgery mask, Fig. 4.8.

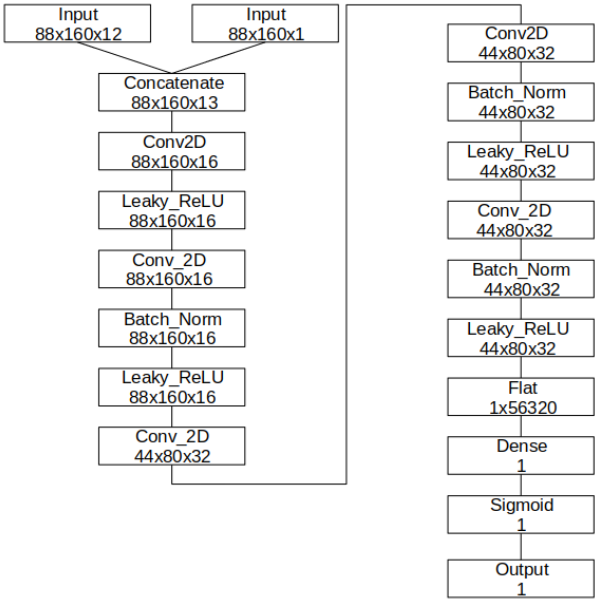


Figure 4.9: Discriminator's architecture

Our generator works as an autoencoder and it based on the U-net from[4]. A characteristic of this type of autoencoder is its use of skip layers as Figure 4.3 shows.

The left contracting part is followed by layers where pooling operators are replaced by upsampling operators; these layers increase the resolution of the output. Moreover high resolution features from the contracting part are concatenated to the upsampled output. In this architecture there is a large number of feature channels in the upsampling which allow the network to propagate context information to higher resolution layer. For this reason the right expansive path is almost symmetric to the contracting path, thus giving the net its u-shape.

The contracting part of our generator consists of the repeated application of two convolutional layers with receptive field of size 3 x 3 with a padding of 1 pixel to maintain the size of the input. Each repetition is followed by a ReLU activation function, and a max pooling layer with receptive field of size 2 x 2 and stride 2. At each downsampling step we halve the size of the input but we double the number of feature channels to maintain a constant number of neurons through the architecture. In the expansive part we have the same architecture of the contracting part but instead of using the max pooling layer we use a 2 x 2 convolution to upsample the input and at the same time we halve the feature channels. The final convolution layer is made with a receptive field of size 1 x 1 with a single feature channel and a sigmoid

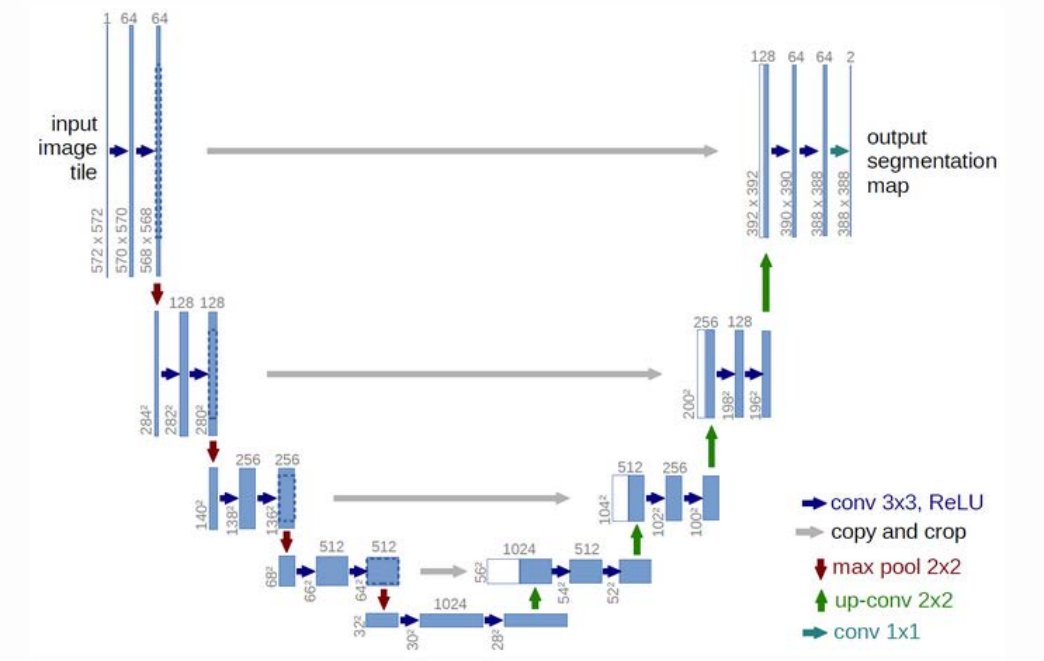


Figure 4.10: Unet[4]

activation function. The network does not contain any fully connected layers and only uses the valid part of the convolution; this mean that the segmentation map only contains the pixels for which the full context is available in the input image.

To train the generator, we concatenate the model of the generator with the model of the discriminator; during the backpropagation phase of the training we block the weights of the discriminator, and update only the weights of the generator. The goal of the generator is to output a mask $\hat{M}$ that given in input together with the other features to the discriminator is classified with the value 0.9, the value associated with samples from the dataset containing real data.

The loss function used for the training of the discriminator is the binary cross-entropy loss function; for the generator we have used the sum of the binary cross-entropy loss function of the discriminator and the binary cross-entropy loss function between the output of the generator and the true mask. This last loss is multiplied by a factor λ used as a regularizer between the two losses. This means that the generator not only needs to create a mask $\hat{M}$ that fools the discriminator, but it need also to be similar to the real mask.

After the training, the discriminator network is discarded and the generator is used for the testing.

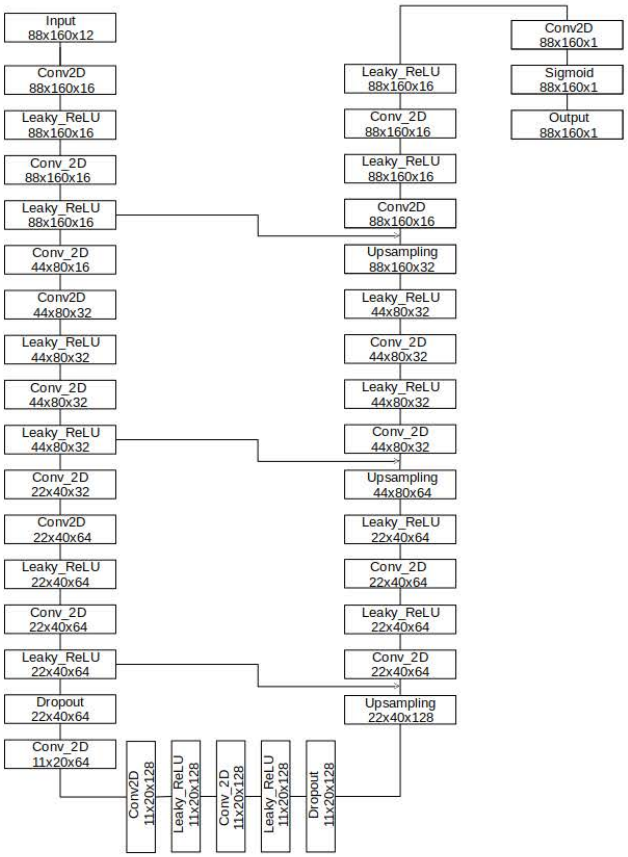


Figure 4.11: Gnet

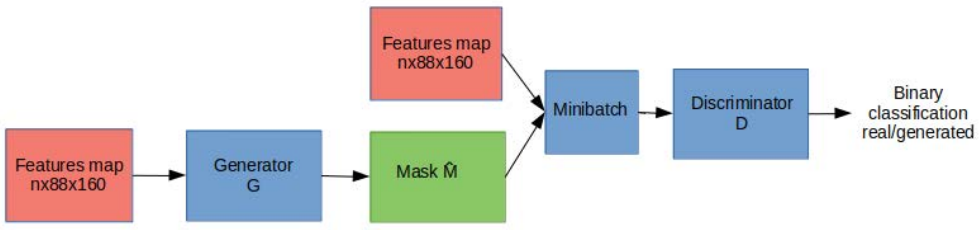


Figure 4.12: Architecture of the generator

5 Dataset

A dataset has been created to train the neural network to detect the frames where a face has been modified and to localize that face.

The videos used in the dataset are taken from YouTube, and a database published by Google in collaboration with Jigsaw [17]. One problem of YouTube's videos is that the algorithm used to upload them on the platform compresses the frames and we lose a significant amount of information.

The dataset is divided in 3 subset: Training set, Validation set, and Test set. More precisely in the Training set we have 5300 frames, in the validation set 1626 frames, and in the test set 2280 frames. The frames of the Training and validation set come from the same group of videos: 12 videos are downloaded from YouTube, 8 are deepfakes, the other are real; 16 videos are taken from the Google dataset, 6 of these are deepfake and the other are real.

	Youtube		Google			
			YouTube		Original	
	Real	Fake	Real	Fake	Real	Fake
Videos	8	4	3	5	3	5
Frames	2569	1799	447	832	447	832

Table 5.1: Train Dataset

For the Test set a completely different set of videos is used. We have a total of 15 videos, 5 taken from YouTube, 10 from the Google dataset, of which the first 5 are from YouTube,

the other 5 are original. In order to train efficiently the model we need to feed to the input a dataset that contains in equal part both real videos and forged videos. In this dataset we have 3463 real frames and 3463 forged frames.

	Youtube		Google			
			YouTube		Original	
	Real	Fake	Real	Fake	Real	Fake
Videos	3	2	3	2	3	2
Frames	450	300	450	300	450	300

Table 5.2: Test Dataset

6 Experiment

The code of the neural network was written in Python 3.6 using Tensorflow and Keras.

Tensorflow is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML powered applications [52].

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. It was developed with a focus on enabling fast experimentation[53].

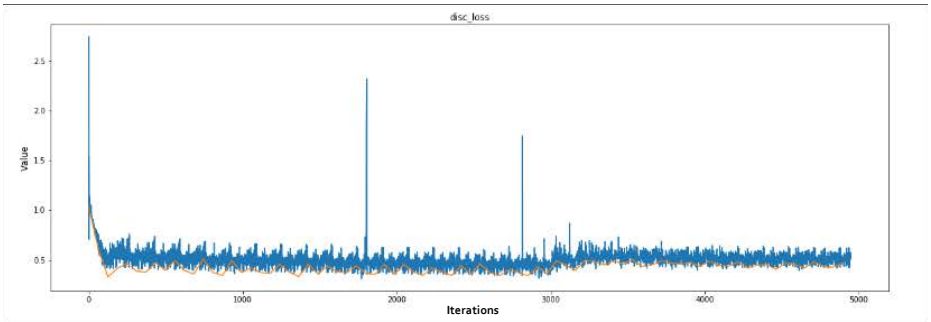
As said before, training a neural network is a difficult task since each problem requires an ad hoc architecture. Even for a single architecture it is possible to use a lot of combinations of different parameters, such as the optimizer, the batch size given in input of the neural network, the learning rate, the loss function, the number of epochs, the decreasing of the learning rate after a certain amount of epochs.

Some of these parameters were chosen by taking inspiration from the state of the art. For example we have chosen the Adam optimizer, and the binary cross-entropy loss function.

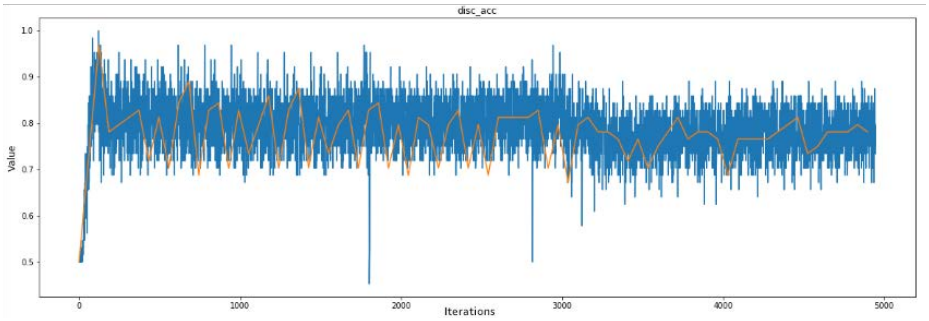
For the other parameters, a brute force approach was used. Given a number of epochs used for training, two for-loops are used, one inside the other, the first that cycles through four values of initial learning rates: 0.0001, 0.0002, 0.0003, 0.0004, the second cycles on three values of batch size: 16,32,64. Additionally, the value of the learning rate is reduced by a factor of 1/5 every 5 epochs.

From Fig 6.1a we can see that the loss of the discriminator is a little higher than 0.5. In an

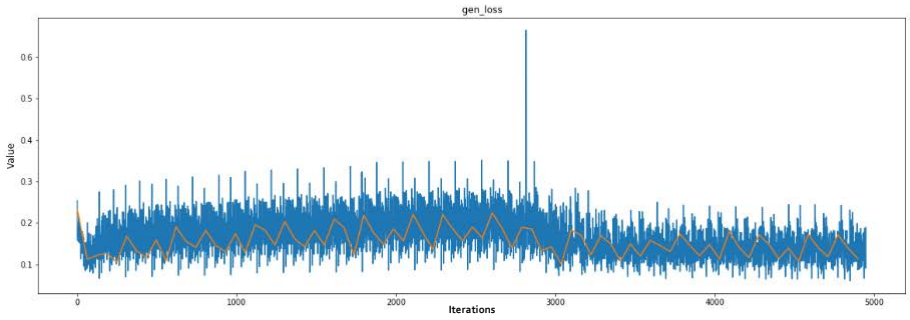
optimal GAN architecture it is supposed to have a discriminator that is unable to distinguish the real samples from the generated samples resulting in a loss of 0.69 using the cross-entropy loss function. Since we have changed the value of the label from 1 to 0.9 for the real samples, our optimal result is 0.62, close to what we obtain. Fig 6.1c shows the loss of the discriminator with only the generated samples. We can see that it is close to 0.1, meaning that the generator is able to create mask able to fool the discriminator. At the end it was chosen the model with the lowest loss on the generator, this model was trained with learning rate 0.0001 and batch size of 32 frames for 30 epochs; the training lasted for 32 minutes.
The generated mask is a matrix of values between 0 and 1, in order to compute how good the mask is we need to apply a threshold t such that each value greater than t will become 1, the other 0. To measure the quality of the generated mask $\hat{M}$, four classes of pixels are used. If the pixel in the real mask has value 1, $M(i,j) = 1$, and $\hat{M} = 1$, then that pixel belong to class TP, True Positive; if $\hat{M} = 0$, it belongs to class FN, False Negative. If the pixel in the real mask has value 0, $M(i,j) = 0$, and $\hat{M} = 0$, the pixel belongs to class TN, True Negative; otherwise if $\hat{M} = 1$, the pixel belongs to class FP, False Positive. In the next operations we consider TP, TN, FP, and FN as the number of pixels in each class.



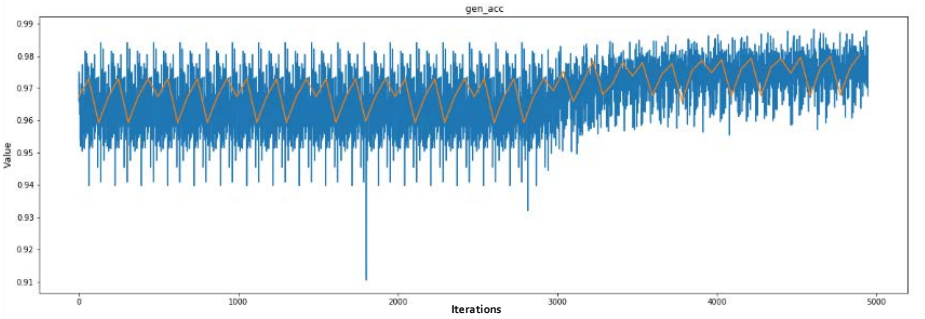
(a) Discriminator loss.



(b) Discriminator accuracy.



(c) Generator loss.



(d) Generator accuracy.

Figure 6.1: Network training

The metrics used are several, each of them tells us some information on the quality of the result.

Accuracy, it is the proportion of the true results among the total number of results:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}.$$

Precision is defined as the fraction of true positive results among the positive results, both true and false.

$$precision = \frac{TP}{TP + FP}.$$

Recall, is the number of correct positive results divided by the number of all samples that should have been identified as positive, it is also called True Positive Rate

$$recall = \frac{TP}{TP + FN}.$$

Specificity, or True Negative Rate, it measures the proportion of actual negatives that are correctly identified as such.

$$Specificity = \frac{TN}{TN + FP}.$$

False Positive Rate, is the fraction of negative results wrongly classified as positive

$$FPR = \frac{FP}{FP + TN}.$$

F1 score, it is a measure of accuracy that consider both the precision and the recall; more precisely is the harmonic mean of the two values.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}.$$

The Matthew Correlation Coefficient takes in to account the cardinality of all the classes, and can be used even if the classes are of very different size. The score has value [-1,1], with 1 representing perfect prediction, 0 random prediction, -1 all wrong prediction.

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}.$$

The reason we use multiple accuracy measures is the following: as explained in [54], all

the previous metrics, except for the MCC, can be misleading, since they do not consider the size of the four classes in their computation.

Suppose we have a dataset with 100 samples, 95 are positive and 5 are negative; the tested model always predicts positive. In this situation the value of accuracy and F1 are respectively 95% and 97.44%, while the MCC value is undefined since the denominator is 0. Instead if the model gives the following result: TP=90, FP=5, TN=1, FN=4, we have again a high value for the accuracy and F1, 91% 95.24% respectively, and MCC=0.14.

However the first 6 metrics are still useful as we can see on the results' tables, because some videos have $M(i, j) = 0 \; \forall i, j$ thus TP and FN are always 0, hence MCC is undefined for these videos, so we have to use the other metrics to evaluate the accuracy.

The results on the test dataset is the following, the values in the table are the mean of the respective metrics of each video sequence.

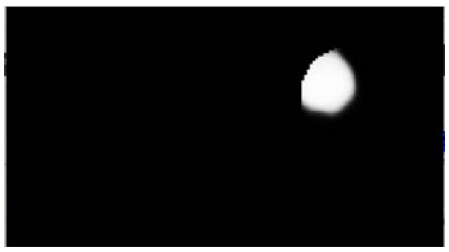
Video	Accuracy	Precision	Recall	TNR	FPR	F1	MCC
DeepfakeYT 1*	0.99438	0.00000	0.00000	0.99974	0.00026	0.00000	-0.00039
DeepfakeYT 2	0.95284	0.00000	0.00000	1.00000	0.00000	0.00000	-0.00005
DeepfakeYT 3	0.80921	0.16298	0.29001	0.85877	0.14123	0.20690	0.11516
TrueYT 1	0.98115	0.00000	0.00000	0.98115	0.01885	0.00000	undefined
TrueYT 2	0.93650	0.00000	0.00000	0.93650	0.06350	0.00000	undefined
DeepfakeYT 4	0.95166	0.73266	0.47162	0.99260	0.00740	0.53297	0.54693
DeepfakeYT 5	0.91371	0.10630	0.04211	0.98760	0.01240	0.05781	0.03510
DeepfakeYT 6	0.81503	0.00679	0.04261	0.84038	0.15962	0.01169	-0.05638
TrueYT 3	0.99783	0.00000	0.00000	0.99783	0.00217	0.00000	undefined
TrueYT 4	1.00000	0.00000	0.00000	1.00000	0.00000	0.00000	undefined
Deepfake 1	0.90210	0.34608	0.45842	0.93641	0.06359	0.38317	0.34211
Deepfake 2	0.96060	0.00000	0.00000	1.00000	0.00000	0.00000	0.00000
Deepfake 3	0.92810	0.45765	0.13959	0.99094	0.00906	0.19550	0.21320
True 1	0.99004	0.00000	0.00000	0.99004	0.00996	0.00000	undefined
True 2	0.98187	0.00000	0.00000	0.98187	0.01813	0.00000	undefined

Table 6.1: Results of the metrics for each video in the test dataset. *The first 101 frames of this video are original, the remaining frame are forged. The MCC value is considered only for this last part.

It is also possible to have a visual result. Unfortunately the output of the generator is not very precise, but we can use the Viola-Jones algorithm to cross-check the result. More precisely, we can analyze the pixels that fall inside the region determined by the Viola-Jones algorithm to detect faces with the pixels outside the region.

	Accuracy	Precision	Recall	TNR	FPR	F1	MCC
Forged	0.90934	0.21330	0.16932	0.95387	0.04625	0.16272	0.14013
Real	0.98311	0.00000	0.00000	0.98311	0.01689	0.00000	undefined

Table 6.2: Average results for real and forged videos



(a) Pixels inside the mask created using the Viola-Jones algorithm

(b) Pixels outside the mask created using the Viola-Jones algorithm

Figure 6.2: Visual Results

If the majority of the pixels are inside the region then we conclude that the frame is forged, if the image has no white pixels then the image is real; finally if the image presents regions of white pixels we are not sure of the results, but since the neural network detected some inconsistencies other techniques to detect forgery should be applied. If we sum the value of the pixel inside and outside of the region detected using Viola-Jones and plot the result we obtain the graph in Fig. 6.3

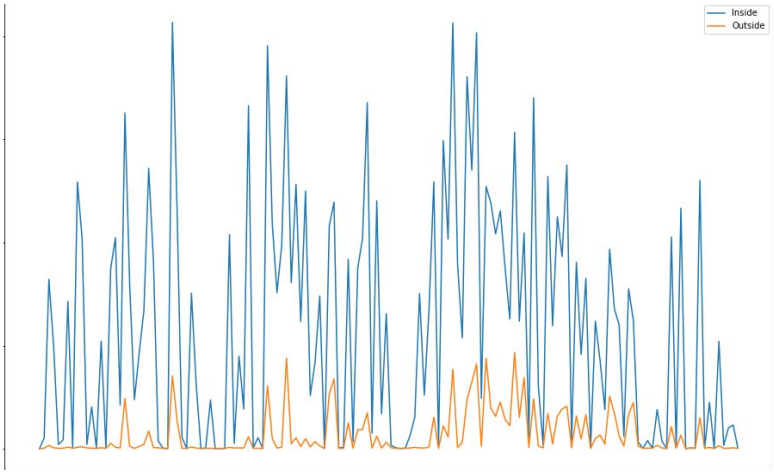


Figure 6.3: Plotting of the visual result for each frame of a video

If we upscale the map and overlap it with the frames of the video sequence we obtain the following results.

Figure 6.4: Results on a forged video. The neural network is able to localize the tampering

Figure 6.5: Results on a forged video. The neural network wrongly localize forged region of the image.

Figure 6.6: Results on a real video. The neural network correctly output a mask of zeros

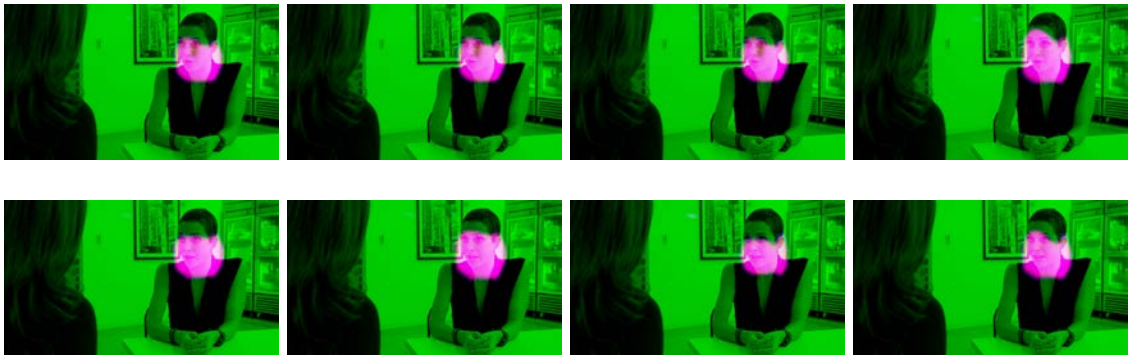


Figure 6.7: Results on a forged video. The neural network is able to localize the tampering

7 Conclusion

The objective of this thesis was to develop a tool to localize face forgery in a video sequence. In fact, the constant increasing quality of forged videos makes them a serious threat in the near future where it will be almost impossible to distinguish a real content from a fake one.

The proposed solution relies on a cGAN, a deep learning framework where two neural networks play one against the other. To maximize the information retrieved from a video sequence, a total of 16 features are extracted for each frame: 4 for the codec used, 4 for the quality of the compression, 1 for the pixel-wise difference between two consecutive frames, 2 for the optical flow, 1 for the mask showing the region of the forgery (used as the conditional element of the network), 1 for the macroblock type, 3 for the Benford's Law applied to the DCT component of the frame. The combination of these features allows the neural network to recognize regions where there are discrepancies with respect to the rest of the frame. The dataset used to train and test the architecture is made of videos downloaded from YouTube and a dataset provided by a collaboration between Google and Jigsaw.

This solution is still in its prime, and several modifications can be applied to increase its accuracy. First of all it is possible to increase the dataset used for the training; we can also modify some layers of the neural network or introduce some techniques to improve it such as the mini-batch discrimination or by using an unrolled GAN to avoid falling in the Helvetica scenario. Additionally we can add or modified some of the feature used, or use a combination of features depending on the scene, but this require a dedicated network for each case.

References

[1] The ultimate guide to convolutional neural networks (cnn). [Online]. Available: https://www.superdatascience.com/blogs/the-ultimate-guide-to-convolutional-neural-networks-cnn

[2] https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/. [Online]. Available: https://machinelearningmastery.com/diagnose-overfitting-underfitting-lstm-models/

[3] Intro to optimization in deep learning: Momentum, rmsprop and adam. [Online]. Available: https://blog.paperspace.com/intro-to-optimization-momentum-rmsprop-adam/

[4] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," *ArXiv*, vol. abs/1505.04597, 2015.

[5] S. Rohini and S. Manoj, "A review of video forgery and its detection," *IOSR Journal of Computer Engineering*, vol. 20, 2018.

[6] T. T. Nguyen, C. M. Nguyen, D. T. Nguyen, D. T. Nguyen, and S. Nahavandi, "Deep learning for deepfakes creation and detection," 09 2019.

[7] deepfakes, explained the rise of fake realistic videos online. [Online]. Available: https://www.businessinsider.com/deepfakes-explained-the-rise-of-fake-realistic-videos-online-2019-6?IR=T

[8] Google ai blog: Contributing data to deepfake detection research. [Online]. Available: https://ai.googleblog.com/2019/09/contributing-data-to-deepfake-detection.html

[9] Creating a data set and a challenge for deepfakes. [Online]. Available: https://ai.facebook.com/blog/deepfake-detection-challenge/

[10] National security challenges of artificial intelligence, manipulated media, and deepfakes. [Online]. Available: https://intelligence.house.gov/calendar/eventsingle.aspx?EventID=653

[11] O. Al-Sanjary and G. Sulong, "Detection of video forgery: A review of literature," *Journal of Theoretical and Applied Information Technology*, vol. 74, pp. 207–220, 01 2015.

[12] A. W. A. Wahab, M. A. Bagiwa, M. Y. I. Idris, S. Khan, Z. Razak, and M. R. K. Ariffin, "Passive video forgery detection techniques: A survey," *2014 10th International Conference on Information Assurance and Security*, pp. 29–34, 11 2014.

[13] C.-C. Hsu, T.-Y. Hung, C.-W. Lin, and C.-T. Hsu, "Video forgery detection using correlation of noise residue," 10 2008, pp. 170–174.

[14] S. Verde, L. Bondi, P. Bestagini, S. Milani, G. Calvagno, and S. Tubaro, "Video codec forensics based on convolutional neural networks," *2018 25th IEEE International Conference on Image Processing (ICIP)*, 2018.

[15] J. Pamela, E. Eyad, and C. Jayne, "Video tampering localisation using features learned from authentic content," *Neural Computing and Applications*, 2019.

[16] B. Bayar and M. Stamm, "A deep learning approach to universal image manipulation detection using a new convolutional layer," 06 2016, pp. 5–10.

[17] A. Rössler, D. Cozzolino, L. Verdoliva, C. Riess, J. Thies, and M. Nießner, "FaceForensics++: Learning to detect manipulated facial images," in *International Conference on Computer Vision (ICCV)*, 2019.

[18] Y. Li and S. Lyu, "Exposing deepfake videos by detecting face warping artifacts," 11 2018.

[19] D. Afchar, V. Nozick, J. Yamagishi, and I. Echizen, "Mesonet: a compact facial video forgery detection network," 09 2018.

[20] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *Bulletin of mathematical biophysics*, vol. 5, pp. 115–133, 1943.

[21] S. C. Kleene, "Representation of events in nerve nets and finite automata," *Project Rand, Research Memorandum*, pp. 1–98, 1951.

[22] History of the perceptron. [Online]. Available: https://web.csulb.edu/~cwallis/artificialn/History.htm

[23] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, L. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," 2017.

[24] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[25] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[26] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceeding of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[27] S. Albawi, T. Abed Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," 08 2017.

[28] K. O'Shea and R. Nash, "An introduction to convolutional neural networks," *ArXiv e-prints*, 11 2015.

[29] S. Ioffe and C. Szegedy, "Batch normalization: accelerating deep network training by reducing internal covariate shift," *Proceedings of the 32nd International Conference on International Conference on Machine Learning*, vol. 37, pp. 448–456.

[30] H. Shimodaira, "Improving predictive inference under covariate shift by weighting the log-likelihood function," *Journal of Statistical Planning and Inference*, vol. 90, pp. 227–244, 10 2000.

[31] A. Krogh and J. A. Hertz, "A simple weight decay can improve generalization," *NIPS'91 Proceedings of the 4th International Conference on Neural Information Processing Systems*, pp. 950–957, 1991.

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 06 2014.

[33] D. Mishkin and J. Matas, "All you need is a good init," 05 2016.

[34] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," *Journal of Machine Learning Research - Proceedings Track*, vol. 9, pp. 249–256, 01 2010.

[35] K. He, X. Zhang, S. Ren, and J. Su, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *Proceedings of the 2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1026–1034, 2015.

[36] J. Martens, "Deep learning via hessian-free optimization," *Proceedings of the 27th International Conference on Machine Learning*, pp. 735–742, 2010.

[37] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[38] P. Baldi, "Autoencoders, unsupervised learning, and deep architectures," *UTLW'11 Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning workshop*, vol. 27, pp. 37–50, 2012.

[39] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Proceedings of the 27th International Conference on Neural Information Processing Systems*, vol. 2, pp. 2672–2680.

[40] I. Goodfellow, "Nips 2016 tutorial: Generative adversarial networks," 12 2016.

[41] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *ArXiv*, vol. abs/1411.1784, 2014.

[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[43] T. Salimans, I. Goodfellow, W. Zaremba, V. Cheung, A. Radford, and X. Chen, "Improved techniques for training gans," *NIPS'16 Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 2234–2242, 2016.

[44] L. Metz, B. Poole, D. Pfau, and J. Sohl-Dickstein, "Unrolled generative adversarial networks," *ArXiv*, vol. abs/1611.02163, 2016.

[45] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," 11 2015.

[46] J. Gauthier, "Conditional generative adversarial nets for convolutional face generation," 2015.

[47] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 08 1981.

[48] I. Amerini, L. Galteri, R. Caldelli, and A. Del Bimbo, "Deepfake video detection through optical flow based cnn," in *The IEEE International Conference on Computer Vision (ICCV) Workshops*, Oct 2019.

[49] S. Milani, M. Tagliasacchi, and S. Tubaro, "Discriminating multiple jpeg compression using first digit features," *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2253–2256, 2012.

[50] D. Fu, Y. Q. Shi, and W. Su, "A generalized benford's law for jpeg coefficients and its applications in image forensics," *Proc SPIE*, vol. 6505, 02 2007.

[51] S. K. Yarlagadda, D. Guera, D. M. Montserrat, F. M. Zhu, E. J. Delp, P. Bestagini, and S. Tubaro, "Shadow removal detection and localization for forensics analysis," *ICASSP 2019 - 2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 2677–2681, 2019.

[52] Tensorflow. [Online]. Available: https://www.tensorflow.org/

[53] Keras. [Online]. Available: https://keras.io/

[54] D. Chicco, "Ten quick tips for machine learning in computational biology," *BioData Mining*, vol. 1, 2017.