DEGREE PROJECT IN MATHEMATICS,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2020

Graphical Glitch Detection in Video Games Using CNNs

CARLOS GARCÍA LING

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Graphical Glitch Detection in Video Games Using CNNs

CARLOS GARCÍA LING

Degree Projects in Mathematical Statistics (30 ECTS credits)
Master*s Programme in Applied and Computational Mathematics
KTH Royal Institute of Technology year 2020
Supervisor at ES SEED: Linus Gisslén
Supervisor at KTH: Henrik Hult
Examiner at KTH: Henrik Hult

TRITA-SCI-GRU 2020:090
MAT-E 2020:052

Royal Institute of Technology
School of Engineering Sciences
KTH SCI
SE-100 44 Stockholm, Sweden
URL: www.kth.se/sci

Abstract

This work addresses the following research question: *Can we detect videogame glitches using Convolutional Neural Networks?* Focusing in the most common types of glitches, texture glitches (Stretched, Lower Resolution, Missing and Placeholder). We first systematically generate a dataset with both images with texture glitches and normal samples.

To detect the faulty images we try both Classification and Semantic Segmentation approaches, with a clear focus on the former. The best setting in classification uses a ShuffleNetV2 architecture and obtains precisions of 80.0%, 64.3%, 99.2%, and 97.0% in the respective glitch classes Stretched, Lower Resolution, Missing, and Placeholder. All of this with a low false positive rate of 6.7%.

To complement this study, we also discuss how the models extrapolate to different graphical environments, which are the main sources of confusion for the model, how to estimate the confidence of the network, and ways to interpret the internal behavior of the models.

Sammanfattning

Detta projekt svarar på följande forskningsfråga: *Kan man använda Convolutional Neural Networks för att upptäcka felaktiga bilder i videospel?* Vi fokuserar på de vanligast förekommande grafiska defekter i videospel, felaktiga textures (sträckt, lågupplöst, saknas och platshållare). Med hjälp av en systematisk process genererar vi data med både normala och felaktiga bilder.

För att hitta defekter använder vi CNN via både Classification och Semantic Segmentation, med fokus på den första metoden. Den bäst presterande Classification-modellen baseras på ShuffleNetV2 och når 80.0%, 64.3%, 99.2% och 97.0% precision på respektive sträckt-, lågupplöst-, saknas- och platshållare-buggar. Detta medan endast 6.7% av negativa datapunkter felaktigt klassifieras som positiva.

Denna undersökning ser även till hur modellen generaliserar till olika grafiska miljöer, vilka de primära orsakerna till förvirring hos modellen är, hur man kan bedöma säkerheten i nätverkets prediktion och hur man bättre kan förstå modellens interna struktur.

Extracto

El principal objetivo de este proyecto es responder a la siguiente pregunta: *¿Pueden las redes neuronales convolucionales ser usadas para detectar anomalías gráficas en imágenes de videojuegos?* Nuestra atención se centra en errores relacionados con texturas (Estirada, Baja Resolución, Ausente y Predeterminada), pues son los que con más frecuencia padecen los videojuegos. Así, hemos generado tanto muestras normales como anomalías de manera sistemática con el fin de entrenar nuestros modelos.

Para esta tarea hemos empleado dos enfoques, clasificación y segmentación semántica, aunque con especial predilección por el primero. De esta manera, alcanzamos los mejores resultados mediante la arquitectura ShuffleNetV2 que obtiene precisiones 80.0%, 64.3%, 99.2% y 97.0% en las clases Estirada, Baja Resolución, Ausente y Predeterminada respectivamente, todo ello con un ratio de falsos positivos del 6.7%.

De forma complemetaria, exponemos cómo los modelos permiten extrapolar y detectar errores en diferentes entornos gráficos, cuáles son las principales fuentes de confusión en el modelo, así como formas de estimar la certidumbre de las predicciones obtenidas y modos de intrepretar el funcionamiento interno de los modelos

Acknowledgements

This work with the support and collaboration of *SEED* a cross-disciplinary team within EA Worldwide Studios, whose mission is to explore, build, and help define the future of interactive entertainment. In particular, I would like to thank Linus Gisslén for his continuous support and supervision during this project, thank also the people from the DICE team at EA involved in the project and in particular to Christian Deri, Jesper Klittmark and Jan Schmidt for the resources and feedback provided to steer the project towards a more applied tool.

I would like to especially thank the institution *Fundación Barrié de la Maza* for providing financial support during my studies within the Master's Program in Applied and Computational Mathematics, which are finalized with this project.

To end, I thank my friends and relatives for their continuous support, and in particular Ivar Eriksson, that helped me with the Swedish translation of the abstract.

Contents

Chapter 1

Introduction

When developing a new videogame there are many steps starting from the concept to the final release. Towards the end of the project, one fundamental part of the process is the testing phase. Here, different quality requirements should be met ensuring the optimal game experience. Since graphics are one of the main components of the game, it is mandatory to assure the absence of glitches or malfunctions that may hinder the experience.

These graphical errors are usually hard to spot and occur in small proportions. Right now, they are identified by testing the game in a manual way, and, when detected, they are diagnosed and addressed. Although useful, this process can be extremely time-consuming due to the scarcity of the glitches, and, since it is not systematic, many of them might be overlooked.

As a leader company in the videogame industry, EA, is looking forward to automating these testing processes. All this in order to speed up the developing stage, while delivering high quality, glitch-free products.

Due to the nature of the problem, taking unexpected forms of graphic failure, automatizing this process is not an easy task. In spite of being diverse, these graphical glitches are usually: stretched textures, missing textures, and textures clipping among others. The objective of this project is to investigate the development of a tool to detect the most common anomalies in videogames.

Due to its wide range of applications anomaly detection is currently a broad and essential research branch. In the latest years, surveys like [1] and [2] have kept compiling the state of the art techniques, adding new and developing methodologies. The applications of anomaly detection are very diverse, among others video surveillance [3, 4], medical image [5, 6, 7], or fraud de-

tection [8] have been successful fields.

In parallel, Deep Learning techniques have been drawing a lot of attention in recent years. After the development of AlexNet [9], many other Convolutional Neural Networks (CNNs) have excelled at the classification task, like [10] or [11]. Due to this success, there have been many researchers trying to apply these methods to anomaly detection [12]. Is this success what motivates our main research question: *Can we detect videogame glitches using CNNs?*

The methodology the project will focus on is the use of CNNs, with a classification problem approach. Different structures and classification settings are explored. Using artificially generated data the models are tested and assessed for the described task. The desirable outcome of this project is to obtain an effective glitch classifier, with a reasonable false positive rate and feasible execution time to be implemented in testing.

To address the research question proposed, this project is structured as follows. First, we present current approaches to similar problems and describe the techniques to be used, Chapter 2. In Chapter 3 we describe how the data is generated, and the particularities of this problem that motivate it. How the models are trained and evaluated is described in Chapter 4, and the outcomes of these are presented and discussed in Chapter 5. To finish future lines of research are presented in Chapter 6 with a final conclusion wrapping up in Chapter 7.

Chapter 2

Theoretical Background

For this section first, we present an overview of the anomaly detection problem and the most used approaches when handling image data. Then, we introduce the core method for this project, Deep Neural Networks, going further in depth with CNNs explaining their components and how they allow for the exploitation of the spatial properties in image data. To finish, we present the state of the art CNN architectures that are going to be used in this project, together with techniques that provide a further understanding of their behavior.

2.1 Anomaly detection

Anomaly detection refers to the problem of finding samples in the data that do not correspond to the expected behavior. These samples are usually denoted as anomalies or outliers. Solving this problem has use in a broad variety of applications such as fraud detection [8], fault detection [13], intruder detection [14] or surveillance [3]. The main challenge to this approach is to define what the expected behavior is. The definition of an anomaly can sometimes be imprecise and depend on other elements such as the context, this is the case when for example dealing with anomalies temporal data.

The importance of anomaly detection lays in the fact that these outliers in the data can be translated into actionable information. Unexpectedly high vibrations may indicate an early fault in a bearing [13] or Abnormal traffic in a computer may indicate that someone is trying to access or control the device illicitly [14]. In this project, the focus is on videogame images where the presence of a graphical anomaly (glitch) usually indicates an error in the

implementation of the game. For instance, a missing texture often indicates a texture that has forgotten to be implemented, while a clipped or stretched texture can indicate that there is some error in the game implementation.

Dealing with images in anomaly detection is complicated, due to the high dimensionality of the input, the classical applications were usually reduced to detecting changes in the images [15], or regions in the image that are abnormal relative to the rest of the image [16]. These methods were designed with clear heuristics and focused on particular characteristics that would define an anomaly, due to computational constraints. In recent years, the increase of computational power allowed for the surge of complex and non linear deep learning models in computer vision. Consistent with this, new approaches have been presented for image anomaly detection.

The detection methods used for image are a part of Deep Learning based Anomaly Detection, which have become increasingly popular greatly surpassing the performance of traditional methods [12]. The success of these methods is strongly reliant on the availability of big amounts of data. Although, in anomaly detection, not all kinds of instances are equally common. Depending on the problem the anomalous data that we can make use of may be scarce. Based on the availability of labeled anomalies we find three distinct approaches to anomaly detection.

2.1.1 Supervised Approach

This method involves the training of a deep supervised binary or multi-class classifier, with labels for the normal instances as well as for the anomalous instances. Because of the definition of an anomaly, there is, in general, a lack of anomalous samples available, which makes these not the preferred methods in most of the applications. The data can be scarce for several reasons, such as the direct lack of adequate samples such as in medical research [17], or a very low proportion and diversity of anomalous samples like in fraud detection [8].

Another thing that may happen is that the number of samples can be big enough but the problem not well defined, which makes the supervised classifier miss the anomalies for which it was not trained on. This is for instance what happens in video surveillance since the amount of normal data is very big with quite some positive examples of anomalies. Although these anomalies have a lot of variance and representing very distinct situations which may lack a precedent example. Although in this particular scenario some soft supervised

approaches such as the one described in [3] have been notably successful, even detecting situations the model was not trained on.

On the other hand, the supervised approach works well if the dataset is well balanced and the images trained to follow a clear pattern as seen in [18]. Supervised Deep Anomaly detection translates the problem into a classical classification problem, which has been one of the most successful applications of CNNs [9].

To the extent of our knowledge, the application to synthetic images such as videogames images has not been tested for supervised anomaly detection. But seems a promising line of research due to the possibility of generating a large number of training samples with a balanced proportion of classes. As a result, this is the approach we have selected more suitable for our project.

2.1.2 Semisupervised Approach

Hybrid methodologies towards anomaly detection have been becoming popular in recent years [19]. Since in many cases obtaining normal instances is usually simpler than obtaining examples of anomalies, semi-supervised deep anomaly detection is widely adopted in high dimensional problems [20]. These techniques use the normal examples to learn the structure of the normal data distribution, after when a new sample is obtained this is compared to the normal distribution of data using some criteria to determine whether it is or not an anomalous instance.

These approaches have been successfully used in applications such as waveform analysis for brain waves [21], intrusion detection [22], or clinical observation data [23]. In computer vision, this has also been a very powerful tool too. The usual setting includes a generative model such as a Variational autoencoder (VAE) [24] or a Generative Adversarial Network (GAN) [25], using reconstruction error as an anomaly score.

In a more sophisticated approach GANomaly [26] has obtained very interesting results. This technique uses a conditional generative adversarial network to learn at the same time the generation of high-dimensional image space and the inference of latent space. Making use of encoder-decoder-encoder subnetworks in the generator network enables the models to map the input image to a lower-dimensional vector which is then used to reconstruct the generated output image. The use of the additional encoder network maps this generated image to its latent representation. Minimizing the distance between the images

and the latent vectors and the latent vectors during training aids in learning the data distribution for the normal samples. As a result, a larger distance metric from this learned data distribution at inference time is indicative of an outlier from that distribution.

2.1.3 Unsupervised Approach

Unsupervised anomaly detection techniques detect outliers based only on the intrinsic properties of the data analyzed, i.e. without any labeled data. These methods are based on one assumption being that the anomalies are much less common than the normal instances. This is the classic setting for anomaly detection where anomalies are solely characterized by their discordance with the rest of the data.

In particular for highly dimensional data the classical approaches in this setting include Principal Component Analysis (PCA) [27], Support Vector Machine (SVM) [28] and Isolation forest [29]. These methods though are now outperformed by the Unsupervised Deep Anomaly Detection techniques, which can use different approaches, examples include Restricted Boltzman Machines (RBM) [30] or Long Short Term Memory (LSTM) Networks [31].

When used in image it is common to encounter VAEs [5] or GANs [6, 7], with a similar setting to the semisupervised approach. Taking the instances that the network has more trouble recognizing as anomalies. These methods are often used in the field of medical image where are used in disease diagnose and monitoring [5, 6, 7].

2.2 Deep Neural Networks: Convolutional Neural Networks

In this section, we will introduce the concept of deep neural networks [32] for classification and regression. First we introduce the feed-forward neural networks:

Let $f : X \to \mathbb{R}$ be a composition of functions $f^{(i)}, i = 1, ...n$:

$$f(x) = f^{(n)} \circ ... \circ f^{(1)}(x) \tag{2.1}$$

In this schema every $f^{(i)}$ is the transition between hidden layers and $f^{(i)}$ is often of a simple form:

$$f^{(i)}(h^{(i-1)}) = g^{(i)}(W^{(i)}h^{(i-1)} + b^{(i)}) = h^{(i)} \tag{2.2}$$

In this notation $h^{(i)}$ is interpreted as the ith hidden layer of the model, $W^{(i)}$ is a matrix that denotes the weights of the layer, $b^{(i)}$ a vector, the bias term, and $g^{(i)}$ is the activation function. Thanks to the activation function the neural network is able to capture the non linearities present in the data, $g^{(i)}$ have to be a non linear function, otherwise, the network could be reduced to a linear function. When fitting the network the activation function plays a big role, since most times the fitting is done by gradient descent, its derivative has to be computed. Thus, a common choice for these are the ReLu activation function where $g^{(i)}(z) = max(0, z)$, having $g^{(i)}(z) = I_{z>0}$.

2.2.1 Convolutional neural networks

Equation 2.2 represents the most basic structure of a neural network denoted fully connected layer or dense layer since each of the elements from the next layer is dependant on all the elements from the previous one. When working with structured data, such as images, this structure can be exploited using Convolutional Neural Networks. Since the work presented in [9] these structures have become increasingly popular due to their high performance.

A typical convolutional neural network architecture consists on several blocks of layers, each of those containing convolutional layers, detection layers, and pooling layers, which are ended by a or a set of fully connected layers at the end. Below the three mentioned layers are described:

Convolutional layers

Given a 2D input of dimensions $n_x x n_y$, $I = \{I_{i,j}\}_{i,j=0}^{n_x,n_y}$. And a filter of dimensions $m_x x m_y$ $W = \{W_{i,j}\}_{i,j=0}^{m_x,m_y}$. Will give an output of dimension $n_x - m_x + 1 x n_y - m_y + 1$ $\{H_{i,j}\}_{i,j=0}^{n_x-m_x+1,n_y-m_y+1}$ computed as follows

$$H_{i,j} = \sum_{k=1}^{m_x}\sum_{l=1}^{m_x} W_{k,l}I_{i+k,j+l} \tag{2.3}$$

In contrast with the fully connected layer, the output elements are not dependant on all the input values, instead only to a certain neighborhood of values. This reduces the dependencies between elements. Also, the filter W is the same when computing every element $H_{i,j}$ thus reducing considerably the number of parameters with respect to a dense layer.

Detection layer

This introduces the non linearity to the network. Is simply an activation function $g^{(i)}$ applied to every element of the previous layer. Again, having a 2 dimensional input $n_x x n_y$, $I = \{I_{i,j}\}_{i,j=0}^{n_x,n_y}$. The output is has dimensions $n_x x n_y$ and is computed as follows:

$$H_{i,j} = g(I_{i,j}) \tag{2.4}$$

Again, as in the dense layers, one common activation layer among the literature is the ReLu activation function.

Pooling layers

These layers reduce the dimension output of a convolutional layer by replacing the layer with a summary statistic of the nearby locations. Max pooling layers and average pooling layers are the most common. For instance the output of a max-pooling layer with size $m_x x m_y$ given an input $n_x x n_y$, $I = \{I_{i,j}\}_{i,j=0}^{n_x,n_y}$ is:

$$H_{i,j} = \max_{k=0,\ldots,m_x;l=0,\ldots,m_y} (I_{i+k,j+l}) \tag{2.5}$$

And similarly for an average pooling layer:

$$H_{i,j} = \frac{1}{m_x m_y} \sum_{k=0,\ldots,m_x;l=0,\ldots,m_y} I_{i+k,j+l} \tag{2.6}$$

After the pooling layer, the information is downsampled.

To further reduce the size of the output often some positions are skipped, using what is denoted as a stride (s_x, s_y). The output would then be:

$$H_{i,j} = \sum_{k=1}^{m_x} \sum_{l=1}^{m_x} W_{k,l} I_{s_x \cdot i+k, s_y \cdot j+l} \tag{2.7}$$

Sometimes the opposite is of interest and the size of the input would be maintained. Since the convolutional layers naturally downsample the image is increased in dimension by padding. This means adding in a symmetric way rows and columns to the input so the output is of the same dimension (Adding $m_x - 1$ columns and $m_y - 1$ rows). Usually, this is done by adding zeros or reflecting the image.

Tensors

All that we have described refers to two-dimensional data. In particular, when working with images (except for the black and white images) we represent them using 3D tensors, with three channels (red, green, and blue). Also often when training neural networks we work with a mini-batch using an extra dimension for indicating the sample.

In practice, the convolutional filters used do not only take information for one channel but instead from all the channels from the image. In essence, every layer has several filters that produce different channels and at the same time, each filter takes as input all of the previous channels. The scheme is as follows:

$$H_{i,j,k} = \sum_{l,m,n} W_{i,l,m,n} I_{l,j+m,k+n} \tag{2.8}$$

When dealing with pooling layers or detection layers these are applied to one filter at a time, keeping always the same number of output filters.

2.2.2 Classification

This is the main problem setting throughout the project. The objective is to give an input image $X_i \in X$ being X the image space determines which is the class.

Consider first the binary classification problem. Suppose we have training data in the form of $\{(x_i, y_i)\}_{i=1}^n$ that are outcomes of independent pairs $\{(X_i, Y_i)\}_{i=1}^n$, being X_i the explanatory variables and Y_i the corresponding label. We get a

new sample x_{n+1} from X_{n+1} and want to predict the label Y_{n+1}, that takes values in $\{-1, 1\}$. Let $X = (X_1, Y_1, ..., X_n, Y_n, X_{n+1})$ be the observations and $Y = Y_{n+1}$ the label to predict. The conditional probabilities are modelled as:

$$Pr(Y_i = y_i|\Theta = \theta, X_i = x_i) = \sigma(y_i f_\theta(x_i)) \tag{2.9}$$

Being σ the sigmoid function and f_θ a neural network with parameters θ and a single output. When working with neural networks the usual setting is to estimate the model parameters using the maximum likelihood estimator, θ_{ML}. To maximize the likelihood the log-likelihood is considered:

$$\log p_{X|\Theta}(x|\theta) = ... = \sum_{i=1}^{n} \log \sigma(y_i f_\theta(x_i)) + \sum_{i=1}^{n+1} \log p_{X_i}(x_i) \tag{2.10}$$

Maximizing the previous expression is equivalent to minimize the function $-\sum_{i=1}^{n} \log \sigma(y_i f_\theta(x_i))$ over the parameters $\theta = \{(W^{(i)}, b^{(i)})\}$. The resulting decision rule will classify $\delta(x) = 1$ if $p(1|x_{n+1}, \theta^{ML}) \geq 1/2$. This is the same as defining the decision rule with the neural network as $\delta_\theta(x_i) = f_\theta(x_i)$ providing the label $2I f_\theta(x_{n+1}) \geq 0 - 1$ minimizing the empirical risk function defined as:

$$R(\delta_\theta) = \frac{1}{n} \sum_{i=1}^{n} L(y_i, f_\theta(x_i)) \tag{2.11}$$

where the loss function is $L(y, a) = -\log \sigma(ya)$

When having a multi class classification problem with classes $\{1, ..., C\}$, we represent the labels as one hot vectors, $y_i = (y_i^1, ..., y_i^C)$, meaning that all the components are zero except for the correspondent to the class c which takes value $y_i^c = 1$. In this setting the probabilities for each class are expressed as:

$$Pr(Y_i = a_c|\Theta = \theta, X_i = x_i) = softmax(y_i f_\theta(x_i))_c = \frac{e^{f_\theta^c(x_i)}}{\sum_{c'=1}^{C} e^{f_\theta^{c'}(x_i)}} \tag{2.12}$$

Where a_c refers to the one hot vector for class c and f_θ is a neural network with a C-dimensional output. Analogously as in the binary case the parameters of the network are estimated through maximum likelihood. Obtaining the following minimization problem:

$$\sum_{k=1}^{n}\sum_{c=1}^{C} -y_k^c \log \pi_k^c = \sum_{k=1}^{n} H(y_k|\pi_k) \tag{2.13}$$

Being $\pi_i^c = softmax(f(x_i))_x$, and H referred to as the relative entropy. As in the binary classification, the same decision rule is obtained if we parametrize the decision rule as $\delta_\theta(x_i) = softmax(f_\theta(x_i))$, choosing the label as the argument of the maximum element and training the decision rule by minimizing the empirical risk:

$$R(\delta_\theta) = \frac{1}{n}\sum_{i=1}^{n} L(y_i, f_\theta(x_i)) \tag{2.14}$$

Using the loss function $L(y, a) = H(y|a)$ usually referred as the *cross-entropy loss*.

2.2.3 Regression

In this project, the regression problem is also considered, during the confidence estimation described in the following section a regression framework is used. Here we describe the general setting of a regression problem using neural networks.

We consider regression with additive noise. The provided training samples come in the form $\{(x_i, y_i)\}_{i=1}^n$ being outcomes of independent pairs $\{(X_i, Y_i)\}_{i=1}^n$. Here X_i represents the explanatory variables and Y_i the response variable. We get a new sample x_{n+1} from X_{n+1} and want to predict the label Y_{n+1}. Let $X = (X_1, Y_1, ..., X_n, Y_n, X_{n+1})$ be the observations and $Y = Y_{n+1}$. The regression model is:

$$Y_i = f_\theta(X_i) + \epsilon_i \tag{2.15}$$

Where f_θ is a neural network paramterized by θ. Again as in the classification setting, the parameters are estimated using maximum likelihood obtaining the probability distribution $p_{Y|X,\theta}(y|x, \theta_{ML})$. To obtain an estimator for Y, if the squared loss is used, then a formal bayes rule is the conditional expectation $E(Y|X = x, \theta = \theta_{ML}) = f_{\theta_{ML}}(x)$. The likelihood is:

$$\log p_{X|\theta}(x|\theta_{ML}) = \sum_{i=1}^{n} \log p_\epsilon \left(y_i - f_\theta(x_i)\right) + \sum_{i=1}^{n+1} \log p_{X_i}(x_i) \tag{2.16}$$

Thus maximizing the likelihood is equivalent to minimize the function:

$$-\sum_{i=1}^{n} \log p_\epsilon \left(y_i - f_\theta(x_i)\right) \tag{2.17}$$

Which taking ϵ a Gaussian noise with known variable is equivalent to minimize the empirical square loss:

$$\frac{1}{n}\sum_{i=1}^{n} \left(y_i - f_\theta(x_i)\right)^2 \tag{2.18}$$

2.2.4 Fitting the model

In the previous sections we have defined the objective functions to minimize for the problems stated (Equations 2.11, 2.14 and 2.18). Now, we will explain the core concepts to reach the parameters that minimize this values.

Backpropagation

The most common setting when working with neural networks is using gradient descent methods to find the network parameters. To do so is necessary to be able to compute the gradient. Considering the neural network as a directed computational graph, it is very convenient to use the chain rule as a way to express the derivatives. This is the core idea of the backpropagation algorithm, standard in training neural networks.

In a graph with n nodes: $h_1, ..., h_n$, being h_n a scalar output, the leaf, and $h_1, ..., h_p$ are the ancestors. The following algorithm computes the partial derivatives:

- Initiate the calculation by feeding the values into $h_1, ..., h_p$ and make a forward pass to compute the values of $h_1, ..., h_n$.
- Put $\frac{\partial h_n}{\partial h_n} = 1$ and let $L = n$

- For each $j \in \text{Parent}(L)$ compute:

$$\frac{\partial h_n}{\partial h_j} = \sum_{i:j \in \text{Parent}(i)} \frac{\partial h_n}{\partial h_i} \frac{\partial h_i}{\partial h_j}$$

  and store the values. Update $L = \text{Parent}(L)$ and repeat until L is the empty set

When training a neural network we represent h_n as the scalar loss over a mini-batch of training examples and h_i as the hidden variables. In this case, the derivatives to be computed are the ones to respect the parameters $W^{(i)}$ and $b^{(i)}$, weights and biases of the networks, which implies that we will include the derivatives of the hidden nodes with respect to the parameters as well in the graph.

Stochastic Gradient descent

One particularity when training neural networks is that this use to be high computationally demanding both in computational power but also in memory. When considering classical gradient descent the following scheme is used for updating the parameters of the model:

$$\theta_{t+1} = \theta_t - \lambda \nabla_\theta R^{\text{emp}}(x, y; \theta_t) \tag{2.19}$$

Where λ is the learning rate. The gradient of the empirical risk is computed as follows when we have independent samples:

$$\nabla_\theta R^{\text{emp}}(x, y; \theta_t) = \frac{1}{n} \sum_{k=1}^{n} \nabla_\theta L(x_k, y_k; \theta) \tag{2.20}$$

Here it can be seen that for each gradient update the computation of n different gradients is required which can be slow. To overcome this the gradient is substituted by an estimate which selects n' samples of the dataset and computes the gradient. This is denoted as mini-batch gradient descent and results in the following updating equation:

$$\theta_{t+1} = \theta_t - \lambda \frac{1}{n'} \sum_{k=1}^{n'} \nabla_\theta L(x_k, y_k; \theta) \tag{2.21}$$

When the mini-batch size is 1 this optimization method is known as Stochastic Gradient Descent(SGD). To compensate for the reduced number of computations, this method has the inconvenience that the computation of the gradient has some variance (highest in the case of SGD). An alternative to overcome this is to introduce a momentum term that controls for the variance of the gradient obtaining the following update scheme:

$$m_{t+1} = \gamma m_t + \lambda \nabla_\theta L(x, y; \theta_t) \tag{2.22}$$

$$\theta_{t+1} = \theta - m_{t+1} \tag{2.23}$$

Further methods have been developed in the same gradient descent framework such as Adam [33] or Adagrad [34]. Adam one of the most commonly used among fitting neural networks, and it can be seen as a further iteration of SGD with momentum.

2.3 CNNs: State of the art

This section focuses on describing the methodologies considered in this project. The main focus is in the classification approach since most part of the thesis will be centered around these networks, the core concepts from Resnet and ShuffleNet will be described. In contrast to the other approaches, Semantic Segmentation is just outlined, but the reader is referenced to the original papers for further information.

2.3.1 Classification

After Alexnet [9] won ILSVRC-2012 competition a lot of focus has been put on the power of Deep CNNs. With 5 convolutional layers and 3 fully connected layers, the main highlights of the network is the use of ReLu activation functions, dropout use in the fully connected layers for regularization, and overlapping pooling layers for reducing the size of the network. Further networks came surpassing AlexNet such as VGG16 (13 convolutional layers + 3 dense layers) [35], increasing the number of convolutional layers to 16.

According to the Universal Approximation Theorem [36], a feed-forward neural network with a single hidden layer containing a finite number of hidden units can approximate continuous function on compact subsets of $\mathbb{R}^n$ under

certain assumptions on the activation function. This meaning than increasing the width of a network increases the capacity for approximation. This was also proven to be true when the width and depth are increased [37]. Based on these terms, the trend on DCNNs was to increase the complexity of the networks, in particular, by increasing the depth. Although this seemed to have a limit in practice since with the existing structures the error started to increase when reached a certain number of layers.

ResNet

This limitation was overcome when ResNet appeared, the network was presented for the first time in [38], and presents an innovation that allowed for substantially deeper networks. This is referred to as residual networks, using shortcut connections to facilitate the training of the networks. Formally these shortcuts are expressed as follows:

$$y = F(x, \{W_i\}) + x \tag{2.24}$$

Where x and y are respectively input and output vectors of the block considered. The function $F(x, \{W_i\})$ represents the residual mapping to be learned, this is a neural network, with at least one layer. In the original paper, the number of layers of F is explored, but in the final implementations, this varies between 2 (Figure 2.1) and 3 convolutional layers. The structure of ResNet, consists of the concatenation of these blocks, with 4 downsampling steps when the dimension of the network is reduced finished by an average pooling layer and a fully connected layer in the end. In the case of the downsampling steps, the shortcuts are expressed as follows.

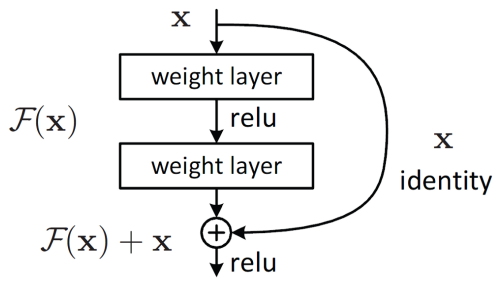


Figure 2.1: Diagram of the residual structure for a function $F(x, \{W_i\})$ consisting of two layers [38]

$$y = F(x, \{W_i\}) + W_s \cdot x \tag{2.25}$$

Since the output from the function $F(x, \{W_i\})$ and x have to match, a matrix W_s is introduced to enforce the dimensions to be compatible. This structure is replicated with different variants from ResNet18 to Resnet152, with the number indicating the number of convolutional layers.

Intuitively, this is motivated by the degradation problem, which in the original paper is justified saying that the added layers can be constructed as identity mappings which are harder to approximate by neural networks. The idea then is to instead approximate the residual F, which translates in the case of having an identity mapping in $F(x, \{W_i\}) = 0$, which is easier to approximate by driving all the weights to zero.

ShuffleNet

Although powerful these DCNNs like ResNet or further advances like DenseNet [10] are very expensive computationally. Looking into real application further developments like MobileNet [39] or ShuffleNet [40] were recently presented as efficient networks for computational tight constrained environments such as mobile phones.

ShuffleNets main innovations are the use of two new operations, points wise group convolution and channel shuffle, to reduce computation cost while maintaining accuracy. In a further iteration of ResNet denoted as ResNeXt [41] the increase in complexity of the network was compensated introducing group convolutions, which restrict the number of channels every filter size takes as an input. This means the computation of the filters is modified from Equation 2.8 to:

$$H_{i,j,k} = \sum_{l=i_t}^{i_{t+1}-1} \sum_{m,n} W_{i,l,m,n} I_{l,j+m,k+n} \tag{2.26}$$

Being $i \in \{i_t, ..., i_{t+1} - 1\}$ with $i_0 = 0$, $i_t < i_{t+1}$ and $i_T = n_i$, n_i the number of filters from the input channel, T the number of groups. This makes the network able to keep more information while keeping the same number of parameters since the filters will have a smaller dimension. The group convolution is only used in the ResNeXt in the 3x3 filters, in contrast, Shufflenet uses

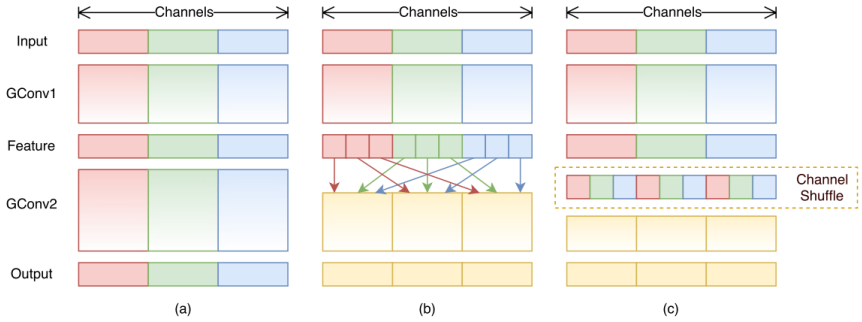


Figure 2.2: a) Group convolution without shuffling channels. b) and c) present the same operation, shuffled convolution c) stating the intermediate step of shuffling the channels. Shuffling allows the further layers to obtain information from all the filters in the previous ones. From [40]

also this approach in the 1x1 convolutions. To avoid the features not "sharing" information the channel shuffle operation is performed (Figure 2.2)

In essence, the final architecture of ShuffleNet is similar to ResNet, with the repetition of several residual blocks with the particularity of the group convolution combined with the channel shuffle operation, finalized with a dense layer. The model considered in this project is described in [11] which has a total of 50 layers.

Activation

The classification approach only provides an output regarding whether the input image is an anomaly or not. Taking advantage of the convolutional structure of the network though, it is possible to provide some information regarding the localization of the anomaly. In [42], some methodologies are provided to understand which are the regions of the image that trigger the activation of each class, which can be used as localization information.

In this work, the authors present a procedure to generate class activation maps using the global average pooling in CNNs. The global average pooling is an adaptive pooling layer such as the one defined in Equation 2.6, with the particularity, that it averages the whole layer, this simply gives one output for every channel of the input.

$$H = \frac{1}{n_x n_y} \sum_{i=0,\ldots,n_x;j=0,\ldots,n_y} I_{i,j} \tag{2.27}$$

After this layer, if the network performs a fully connected layer for the desired output, the importance of the image regions can be obtained by projecting back the weights of the output layer on to the convolutional feature maps. This is if the normal setting is that of a fully connected layer we can write in the following way.

$$S_c = \sum_k w_k^c \sum_{x,y} f_k(x,y) = \sum_{x,y} \sum_k w_k^c f_k(x,y) = \sum_{x,y} M^c(x,y) \tag{2.28}$$

Being S_c the score for class c, w_k^c the weights from the fully connected layer for class c and $f_k(x,y)$ the terms before the average pooling. As it can be seen the score S_c can be expressed as a sum of a map of scores that identify with the different regions of the image. In [42] they obtain really interesting results with this approach, which really showed how the networks "focused" on the parts of the image that provided the most information regarding classification.

Both networks used in this project (ResNet And ShuffleNet) have a structure that enables this method to be used, by having a Global Average Pooling layer followed by a dense layer at the end. Thus, this approach is applicable to both networks.

Confidence estimation

The final softmax layer of the neural network in the classification problem enforces that all the probabilities add to one. Due to its internal structure, the soft-max layer results in models that usually are very confident about making incorrect predictions. This is particularly true when the network is doing extrapolations, i.e. used in environments which it was not trained in. For instance, a classifier trained on telling apart the classes *cat* and *dog* may be shown an image of a flower and be 99% confident in that being in class *cat*.

To provide a better estimate of the confidence of Deep Neural Networks different approaches have been proposed [43, 44]. Motivations can vary from being aware that the Network is working with a different data distribution to protect against adversarial attacks [45].

In this project, we have adopted the approach presented in [46]. Here the authors formally prove the equivalence between softmax based classification and K-means clustering. Taking the following representation of the neural network used in the classification problem $F(x)$

$$F(x) = softmax(f_p(x)^T W) \tag{2.29}$$

Where $f_p(x)$ is the penultimate layer of the network with d components and $W \in \mathbb{R}^{dxC}$ the weights form the last dense layer. With this notation:

Theorem 1 (from [46]): *Let the dimension of the penultimate layer d be at least as large as the number of classes: $d \geq c - 1$. Given a network whose predictions are calculated as $y = \arg\max_c fp(x) > W_{.d}$, there exist C class centroids $Z_{.c} \in \mathbb{R}^d$, equidistant to the origin, such that every point x is assigned to the class whose center is closest in the transformed space:*

$$y = \arg\min_c \|f_p(x) - Z_{.c}\|^2 \tag{2.30}$$

This theorem and one further are proved in the original paper, yielding from these results a confidence measure denoted as *Gauss-confidence*. For every class c this is defined as :

$$\kappa(x)_c = \exp(-\|f_p(x) - C_{.c}\|^2) \tag{2.31}$$

Where $C_{.c}$ are the centroids mentioned in Theorem 1. These centroids together with the penultimate layer of the network form what is denoted as a Gauss-Network that will output a confidence measure for every class classifying each instance according to the class with the higher confidence. The parameters of this model are fitted with the algorithm outlined in [46] which has two parts:

- **Update the centroids:** using the following expression $C = f_p(X)Y(Y^TY)^{-1}$ where $Y \in \mathbb{R}^{mxc}$ and $x \in \mathbb{R}^{mxn}$ the labels and input data respectively
- **Update $f_p(x)$:** by fitting the parameters of the network minimizing the expression $\|fp(X)^T - YC^T\|$

2.3.2 Semantic Segmentation

Although really efficient, the classification approach provides limited information. One label indicating whether the image contains a glitch is output by the model, but no information is obtained regarding where the anomaly is located. Exploring the activation before the softmax layer may give us an idea of what the network may be focusing on but it was not specifically trained on that.

Due to the shortcomings of classification regarding glitch localization, semantic segmentation presents itself as a solution regarding the spatial information. Semantic segmentation is a fine level classification approach, producing pixel level output. This meaning that for each pixel we obtain a probability of it belonging to each of the possible classes.

With the development of deep learning techniques and annotated datasets [47], several advances have come up in this area [48]. These approaches have shown successful results in applications such as autonomous driving [49], medical images [50], pose estimation [51] or satellite images [52].

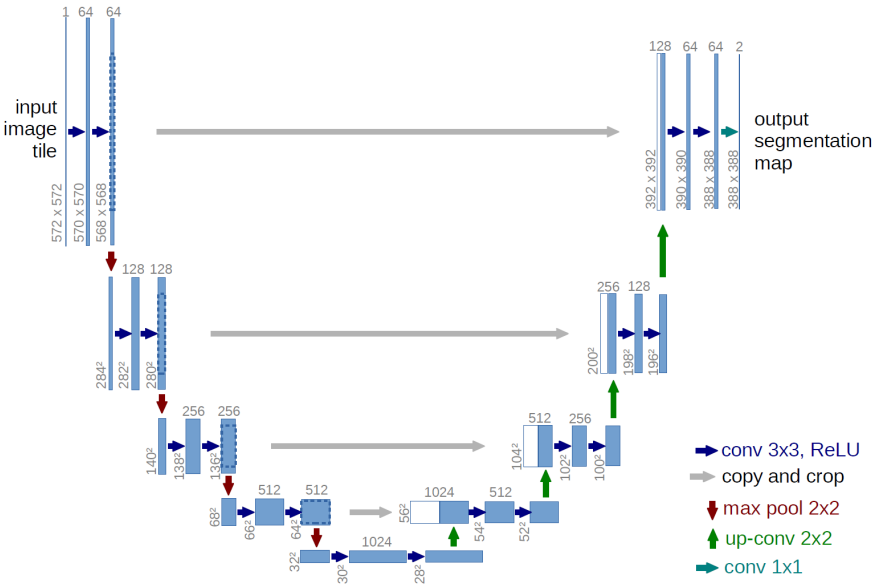


Figure 2.3: U-Net architecture from [53]. The Encoder Decoder structure is common in every Semantic Segmentation approach

In this project has focused on three architectures: Deep Lab V3 [54], Fully Convolutional Networks (FCN) [55] and U-Net [53]. Each focusing in a particular innovation. In this section, the main ideas of the methods will be outlined but not developed in depth. Since they are not the main focus of this

work.

The structure of Semantic Segmentation is usually divided into two main parts. The first is the downsampling part which is the same as a traditional convolutional network after it follows an upsampling part that recovers the information of the deep features using deconvolutional layers (essentially a transposed convolutional layer) to provide an output with the same dimension as the input image (Figure 2.3). This upsampling part lacks contextual information that is lost during the first stage of the network, to avoid this, the U-Net architecture [53] proposes a method that combines information from shallow features. FCN provides a structure that allows the network to take an arbitrary input size, using also the idea of U-Net, by combining information from the first features. And DeepLabV3 features the use of Atrous convolution this allows a broader field of view with similar computational complexity, which is translated in the network being able to obtain more contextual information.

Chapter 3

Data

When working with Deep Learning a big part of the success of a model can be linked to having a good dataset. Using artificial images means that the data used for training does not have to be sourced. Instead, it can be generated in a controlled environment. This potentially presents a good advantage in front of other fields where the availability of data is limited

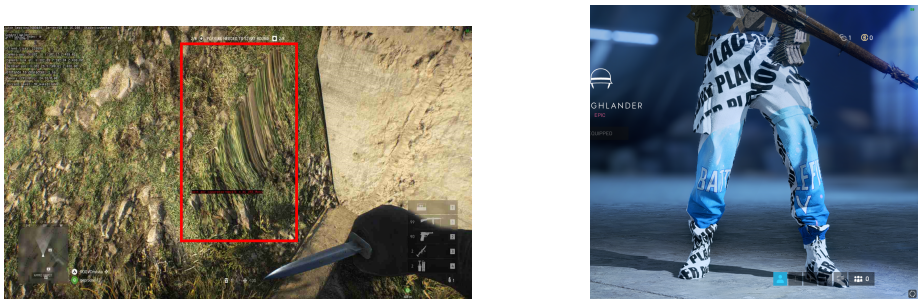
Due to the current unstructured process associated with glitch detection, a lack of a real application dataset is a problem. Most of the glitches recorded in the current system are not specifically labeled, and the image might have been corrupted (See Figures 3.1a and 3.2a), which make them not adequate for training.

As a first approach to the research question, this though does not present a big problem. A custom dataset was generated to explore the viability of glitch detection. The creation of the data is discussed in the section below. First, some general aspects of the data are explained, the glitches considered, and similarity with other kinds of data. Then the procedure for the generation of the data is described in detail.

3.1 General Considerations

In videogames, most of the graphical glitches found in the testing process are related to textures, which are the digital representation of the surface of an object in 3D graphics. Thus, the data used will mainly focus on texture glitches, as discovering these malfunctions is crucial in testing to ensure the best possible graphical experience for players.

The two main types of texture glitches in which the data is centered are the corruption of the textures and missing textures. The causes can be traced easily in both cases, corruption can be due to some logical error such as the stretch of an object or a resolution error. Missing texture is also a known issue usually due to the lack of a texture file or an error when loading.



(a) Stretched texture (b) Placeholder texture

Figure 3.1: Two images that show texture glitches like the ones that are analyzed in the project. The images are taken from real glitch reports when testing Battlefield V

Other examples of glitches involve T-pose which is due to a lack of animation of the object, or clipped objects or total visual corruption (Figure 3.2). There are several other common graphical glitches, but the focus was oriented to textures since they are easy to generate, common and usually have a known cause.

Videogames allow the player to immerse into a wholly interactive experience where his decisions and actions, make him or her an essential part of the game. To do so videogame studios create environments that resemble reality and where the line between what is a real image and what is a videogame image is blurred, allowing players to deeply engage.

This fidelity to natural images is something to account for when developing our models. Adapting the methodologies from natural images or even using models pretrained on those seems to be justified due to these similarities.

3.2 Training data: Unity

To develop a dataset to train a model Unity was the main tool used to do so. Unity is a game engine developed by Unity Technologies. The aim of this

(a) Clipped object (b) Visual corruption

Figure 3.2: Two images with glitches that will not be analyzed in this project. Figure 3.2a shows quite a common glitch in videogames, although this is not trivial to deal with, because of how the objects are defined. Some degree of intersection between objects is usually accepted but is complicated to draw the line, which also complicates generating adequate samples. The visual corruption example takes place when the whole render crashes, these are very critical in videogame testing although we will not address it in this project.

product was to democratize game development. Giving the ability to create both 3D and 2D games. Due to its widespread among game developers, it was considered a good choice to generate data for this project. Also it is easy and quick to learn interface made it a reasonable choice for a first approach to the problem.

The data used to train the model was generated with 5 distinct classes that attend to the type of glitch in the image. The classes are described below:

- **0: Normal.** No glitch in the image, Figure 3.3
- **1: Stretched texture.** The texture is stretched anisotropically, different magnitudes in different directions, Figure 3.4a
- **2: Lower Resolution texture.** The texture is stretched isotropically, which makes it appear blurry, Figure 3.4b
- **3: Missing texture.** The object completely lacks texture, this shows by having a plain color where the object is located, Figure 3.4c
- **4: Placeholder texture.** The object does not have its correspondent texture, instead they are replaced by a white texture, Figure 3.4d. In practice a striking texture is used to draw attention to this and be able to locate this mistakes.

Also, these classes admit a grouping. Classes 1 and 2 are glitches where the texture itself is rendered but it was somehow corrupted, these can be categorized under the same category of corrupted textures. Classes 3 and 4 on the other side happen when the texture is not rendered at all, which can be because the texture is not yet developed, usually the case with the placeholder glitch, or when there has been another kind of error with the object to be rendered. Both types can be grouped into one missing texture group.

The general approach to generate glitches is object centered. Since most of the glitches are linked to an object (*asset*) in particular, the images taken were centered in objects that would later be modified applying the correspondent glitch. Figure 3.3 illustrates that several camera views centered on the object form the dataset.

(a) (b)

(c) (d)

Figure 3.3: Normal samples with different camera positions for the same object. The positions were assessed individually for every object so there are no other objects occluding the view. Then we can ensure that a glitch will be visible when we modify the object

To generate different datasets for use cases with different styles and compo-

(a) 1: Stretched

(b) 2: Low Resolution

(c) 3: Missing

(d) 4: Placeholder

Figure 3.4: The four classes of glitches generated for the view in Figure 3.3a

nents in videogames three environments were used to generate a dataset for training: Natural, Stylized, and Objects.

3.2.1 Natural Environment

It is a constantly developing area in the videogame industry to increase the fidelity of the graphics, generating more realistic environments like forests mountains or cities that may even be mistaken as real to the human eye. This may be the case in videogames such as Battlefield V [56].

This environment represents a forest, here we try to mimic a real situation where some elements of the terrain may crash such as a tree, a bush, or a rock. Samples can be seen in Figure 3.3. The data is obtained from an available environment in Unity [57]. To generate the following procedure was followed.

- **Object selection**. One object from the natural environment is chosen.

This object has to be selected located in an area where the images look realistic (There are some objects that are only used for background purposes).

- **Camera setting**. An available set of camera positions is defined. This process is done manually attending to the placement of the object (Avoid occlusions with other objects), and to their size (The object is visible)
- **Glitch generation**. The glitch is generated by modifying the objects textures as it was described before in the previous section. An image with a glitch is generated by taking a snapshot of the scene, the same camera position is used for obtaining all the different kind of glitches (Figure 3.4). The samples from unity were always obtained with a resolution of 800x800 pixels. This size provides a good balance between game resolution (high resolution), and the design size for the networks used (200 - 300).

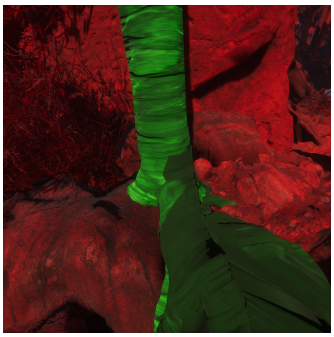
Following this procedure, we assure that the images generated contain at most one glitch per image. This is important to avoid confusing the model when several anomalies are presented the model may predict two labels being an ambiguous answer to the question the network is responding to.

This environment was the only one to be used for the semantic segmentation approach. In order to train such a model, the corresponding labels had to be generated, which means a mask indicating which pixels contain an anomaly and which are normal. This is not straightforward using the unity software. To produce the mask the texture colors were changed attending to the object rendered. Using a green texture for the object for the anomaly object and red textures for all the remaining objects the pixels with higher values in the green channel were denoted as anomalous. In Figure 3.5 an example for one stretched texture is provided. Figure 3.6 shows the final mask obtained for training with white pixels indicating anomalous points and black pixels indicating normal points. Theses anomalous pixels were labeled according to the mentioned glitch classes. In total 12.700 samples were obtained from this environment, using 127 different objects.

3.2.2 Stylized Environment

In some videogames generate real-looking images is not the main goal. Instead, providing a stylized environment is quite common in videogames genres

(a) Original Glitch



(b) Mask

Figure 3.5: Glitch sample together with its corresponding image for generating the mask. The tree is seen green while the background is seen red, this means in a pixel level that the image is represented with higher values in the green channel for the tree and higher values in the red channel for the other pixels

Figure 3.6: Mask obtained from the image 3.5b. As can be seen, the darker part of the glitch is not highlighted in this mask. This is due to the construction of the mask as some trade-offs had to be taken. The rendering of the red and green image is far from perfect and also there is some noise in the image. Thus, the red and green channels could not be directly compared, in addition, a gaussian filter was used to reduce noise to soften the edges of the mask, providing a more regular label

such as First Person Shooters like Apex [58]. This environment was considering this other genre of videogames, using the assets provided in [59]. In Figure 3.7 some of the images without glitches from this dataset are presented.

These renderings present their own challenges, most of the time they do resemble things we are familiarized in reality, but since the theme is specific

for every game, the stylizing can be very different between two games. The main implication of this is the lack of generalization between different stylized games, meaning that different themes may need models trained independently in their correspondent scene.

To generate the dataset with this environment the procedure is exactly the same as in the previous one (Natural environment), being the only difference is the environment chosen. The total number of samples generated in this dataset is 10.000 sampled from 100 different objects.

(a) (b)

Figure 3.7: Two images without glitches that show how the stylized environment looks like. It also looks realistic but some elements. such as the plants are not detailed as in the Natural environment.

3.2.3 Object centered Environment

This environment was motivated by the actual way how videogames are developed. There are several objects, *assets*, that are placed in different parts of the game environment, and can be as well moving around the scene. Examples of these are weapons, characters, or rocks. As a result, it may be interesting to obtain a model that can recognize if the quality of an object is adequate in front of a neutral background.

This set is generated in a different way as in with the others. Both a set of objects and locations is defined. To generate the images, the objects are moved to these positions, placed in a random orientation and the camera is placed randomly focusing on the object.

The objective of this environment is to train the network in a more structured with a scalable way to generate data. In this sense, the objects are placed in

a particular location that has been assessed to not encounter occlusion problems. With this setting, objects can just be dropped in the environment and no particular configuration is needed for each object The objects used for generating this dataset were taken from [60]. In Figures 3.8 a sample of the dataset is presented. In total for this dataset 51 different objects were used generating a total of 11.550 samples.

(a) (b)

Figure 3.8: Two images without glitches that show how the setting of the objects environment. The objects are placed in different locations without obstacles, so that occlusions are avoided. As a result, simpler backgrounds are generated but the objective is for the network to get familiarized to how the objects look like when they have a glitch

3.3 Testing data

During the training of the models, we will validate our training by splitting the dataset and evaluating datapoints that were not used in the training phase. This approach ensures that the model trained does not only fit the trained data but that it also works with data from the same distribution that was not trained in. In order for the approach to be useful, we would like to know how does the trained model performs not only to the environment where it was trained on but also in a different environment, these ways we expect to draw conclusions about how it may generalize.

3.3.1 Unity

Since the data used for training was generated in Unity it makes sense to use the same engine to provide a test dataset. This data will contain elements from the previous environments but with some changes. This aims to determine which use cases would be approachable in a production scenario.

Here we will deal with two environments that respond to different use cases. The first one corresponds to the objects scenario, different objects to the ones used on training were placed in different positions but the same environment. This will help to determine if there is a possible application of these methodologies for new objects introduced in an established environment. In that sense, this data will simply be the validation set from the objects environment.

The other environment combines the natural environment with the objects from the object environment. The objective here is to asses whether the model trained separately in both environments can recognize glitches for one in the other. In Figure 3.9 a sample of the environment is shown. On top of that, the data generated has a low proportion of glitch samples, with only 10% of the samples with glitches and being 2.5% for every glitch class.

(a) (b)

Figure 3.9: Images from the test environment combining both natural and object environments

For both use cases, a video has been generated. The video consists of a flying camera that focuses on particular objects in the scene. The objects are modified on the go and here the performance of the model can be assessed in a more intuitive way.

3.3.2 Frostbite

In order to produce extremely realistic graphics videogame companies develop their own high performance graphical engines. As a leading player in the industry EA has Frostbite as its own videogame engine. This project's final goal is to investigate the viability of a glitch detector's use in these high performance engines, thus it would be interesting to test the model with such tools.

The data from Frostbite currently available corresponds only to very particular scenes where a glitch has been manually detected, because of this, the provided dataset is not well structured. Right now, after presenting results to the team, steps are taking place for integrating this into the process, this branch is yet to be explored and corresponds to future work on this line. See Chapter 6

Chapter 4

Methodology

This section presents how the different methods are assembled with the data to explore the main research question: *Can we detect videogame glitches using CNNs?*

The answer to this question lays in understanding how these methods perform in front of graphical anomalies. Whether they are able to recognize them learning from the training data provided, how they fail, and why they fail.

Neural Network models are of high complexity and non-linear, in addition, the representation of images consists of tensors containing high dimensional numerical data. As a result, understanding the processing of the data inside our model is not a trivial task. Through modifying different aspects of how the networks are trained: data, network architecture, batch size, learning rate... A better understanding of the model's behavior in front of the problem.

This Chapter will first present the metrics defined for assessing the performance of the models, the following sections will describe how the models were trained and what were the aspects modified together with their purpose. The main approach taken was the classification approach, for which aspects like data, hyperparameters, or initialization were discussed. Semantic Segmentation was also explored but in a much lower scope.

4.1 Metrics

First of all, we define and motivate the statistics that have been used to understand the performance of the network, this is important to avoid the biases

corresponding to each particular metric, providing the most clear vision of the output of our model [61]. Although there are defined 4 classes of glitches we can simplify the problem to a 2 class problem obtaining the Contingency Table 4.1. The metrics defined will take into account both the two class problems and the number of classes that the model was trained in.

		Predicted Condition	
		Normal	Glitch
True Condition	Normal	True Negative (TN)	False Positive (FP)
	Glitch	False Negative (FN)	True Positive (TP)

Table 4.1: Contingency table 2 class problem

- **Accuracy:** The simplest metric, it states the fraction of datapoints that were correctly predicted.
- **Recall:** Measures how sensitive the model is to the positive instances (glitches). It is the fraction of the true positives with respect to all the relevant elements i.e. Recall $= \frac{TP}{TP+FN}$
- **Precision:** Indicates how many of the elements marked are actually relevant. It is computed as Precision $= \frac{TP}{TP+FP}$
- **False positive rate:** Measures how many of the normal instances are computed as glitches: False Positive Rate $= \frac{FP}{FP+TN}$
- **Confusion Matrix:** The contingency table indicated in Table 4.1 extended to all the classes predicted by the model. All the confusion matrices will be presented adimensionalized by the total number of instances from one class, this will allow to extrapolate the results to different proportions of glitch classes.
- **Cross Entropy Loss:** The loss function used in the training of classification models, already defined in Equation 2.14
- **Intersection over Union (IoU)** This metric (also referred to as Jaccard index) is used in localization tasks, particularly we used it in the Semantic Segmentation approach. Is defined as the intersection divided by the union of the prediction and original label. Since we are dealing with pixel wise classification the metric is computed as:

  $IoU = \frac{\#\text{Intersection Pixels}}{\#\text{Union Pixels}}$

Accounting for this metric we can say an object (glitch) was detected when this metric exceeds a threshold, accounting for that we have established two criteria 0.3 IoU and 0.5 IoU for considering an object detected.

Note that in the case of having no pixels in the original label we would either get IOU 0 or $0/0$ which we identify as 1.

The different measures indicate different properties in which we are interested. The cross-entropy loss and the accuracy give us a general idea of how the model is performing. In a balanced dataset, the accuracy itself is a good first indicator.

Measures like *Recall*, *Precision*, and *False positive rate* give us a better idea of how the model is performing. For example, when having an imbalanced dataset with only a few glitches, predicting everything as normal will give us a high *accuracy* measure but will have zero recall and the precision would directly no exist since none instance would be marked as positive.

In this problem, we are interested in particular in high precision (low false positive rate) although we would like ideally high recall as well. In a real-life application the glitch detector should be a reliable tool and if the precision is too low it may result in unfeasible for use in practice.

The confusion matrix will give us better insight into how the model is working. In particular, it will give us an understanding of which are the classes that the network has more trouble telling apart, and the ones that the model learns to identify right away.

When working with semantic segmentation the concepts of IoU and detection are broadly used to provide a better understanding of the performance of the algorithms, presenting a more understandable metric than the cross entropy loss used while training.

4.2 Classification

This section discusses the methods regarding the analysis of the classification approach. First, in Section 4.2.1 general considerations regarding the implementation are explained. Section 4.2.2 approaches decision of factors like the batch size, learning rate or network complexity. Different data configurations are examined in Section 4.2.3, followed by Class groupings 4.2.4 and

the initialization of the network 4.2.5. To finish the methods to acquire extra information from the network are discussed (Section 4.2.6)

4.2.1 General Implementation

As an industry standard Python was used to implement and train the models, in particular with help of the Neural Network focused framework Pytorch and its library Torchvision (Oriented to computer vision and CNNs) the models were implemented and trained. The classification approach was the main approach followed in this project being ShuffleNet and ResNet were the main architectures used (Section 2.3.1). The models ResNet and ShuffleNet were taken from Torchvisionand trained using a graphics card *GeForce GTX 1080 Ti*. With this setting, ShuffleNetV2 x1.0, which was the main network used, performed at a rate of 60 frames per second, with an input resolution of 800x800.

To train the models one part of the data was separated to use as a validation set. The split was done so no objects have datapoints in both training and testing sets ensuring the results can be extrapolated to new data. The usual setting was to consider 20% of the objects in the validation set.

The models are trained for 100 epochs by default. After every epoch, the metrics mentioned in the previous section are computed. For the training set, the predictions are the ones obtained during training, in contrast, the validation set is evaluated after each epoch. This decision was taken to avoid repeating computations to evaluate the training set at the end of the epoch. In practice, this may produce unexpected results, particularly during the first epochs, such as that the validation loss could be lower than the training loss.

The Training is done using the Adam optimizer (Section 2.2.4). As a default the learning rate was set to $\lambda = 10^{-3}$, while the rest of the parameters of the optimizer were used as the one predetermined in the **Torchvision** implementation. The models were trained using a batch size of 8. Both the batch size and learning rate were chosen arbitrarily, based on commonly used values when training neural networks.

4.2.2 Networks parameter exploration

This Section discusses the exploration done regarding network complexity, hyperparameters, and the loss function used. The different configurations trained are stated as well as the possible outcomes and what these would imply.

Network complexity

The network most used among this project was ShuffleNet V2 x1.0, although several versions of ResNet were tested as well (Section 2.3.1) all in order to asses the performance of different architectures.

In order to experiment with networks of different depth and complexity, different versions of Resnet and ShuffleNet were trained. For Resnet the versions *Resnet18, Resnet34, Resnet50* provide different depths of the same structural blocks, while or ShuffleNet four versions *ShuffleNetV2 0.5x, 1.0x, 1.5x and 2.0x* vary the number of filters for every layer of the network. All the mentioned versions of the networks were already implemented and sourced from the Torchvision library.

Training different model architectures, as well as different complexities from the same structure, will give us an insight on how dependent are the results in the network chosen, which is the most adequate network for our problem and what are the limitations of each architecture.

Hyperparameter tunning

The two main hyperparameters to do an exploration of, after checking the complexity of the network, are the batch size and the learning rate. The exploration was done only for the ShuffleNet V2 x1.0 architecture.

For the learning rate, a logarithmic exploration was considered taking values from 10^{-2} to 10^{-5}. It is important to choose a value that gives us the correct trade off, to provide the model with a fast convergence but to avoid ending up at a suboptimal solution.

The batch size is the number of examples that are fed to the network with every iteration of the optimizer. When the mini-batch is too small, then a high variance is involved when computing the gradient of the function. Nevertheless, this is usually limited in practice by the hardware used for training the network. In our case the maximum batch size admitted for the graphics card to not run out of memory was 32 for ShuffleNetV2 x1.0. The batch sizes explored were powers of two ranged from 2^2 to 2^5.

Loss function

The Loss function used in classification was always the cross-entropy loss which was already described in Section 2.2.2. Although this loss was sometimes modified adding some weighting. This forces the network to avoid miss classifying classes with higher weight. This was used in cases where the dataset became imbalanced such as when grouping classes 4.2.4.

4.2.3 Data

Being data the fundamental part of fitting the Deep CNN models it is very important to choose an adequate dataset to feed to the network during training. In this section, we describe how we analyze the influence of the data on the model performance.

During the project, the development of a dataset was done in parallel to the fitting of the models. In Section 3.2 we described the generation of the training and validation data to be used. Three different environments were implemented. Since the Natural environment was the first implemented the dataset from this environment has been used in most of the trainings.

Size

One of the main factors for the success of Deep Convolutional Neural Networks is using a big amount of diverse data for training the models. In this project, the creation of data has been a relatively manual process where the degree of scalability is limited by the environments used in our graphical engine.

To investigate the potential of the models used we considered training with different sizes for the training set. This allows us to asses how the models react to more data and whether there is room for further improvement with additional data. The performance remaining constant with different sizes of data would mean that more data would likely not influence the performance of the model, and on the opposite case, a bigger and more diverse dataset would likely improve the performance.

Mixing Data

Different combinations of data were fed to the networks to asses how this influences their performance. The assessment of data was done for all the environments, for every environment, the split between training and validation was kept constant and the whole dataset generated from the other environments was added to the training data.

Adding more diverse data was considered in order to investigate the capacity of the network to generalize to different environments. In this sense, the models trained with more diverse datasets would not be expected to perform better with data from the same distribution but with data from a different distribution. If this is the case then it would mean that the models are able to model very different kinds of data.

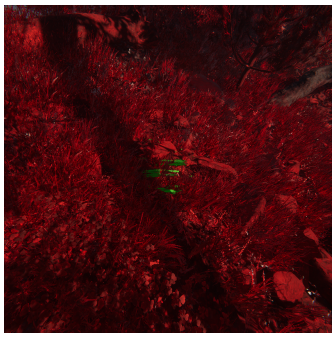
Cleaning Data

One of the main obstacles encountered during this project was the availability of suitable data. In particular, stretched and low resolution textures present a challenge. These were generated by deforming the texture file, although this deformation presents no problem at all, but when rendered into the object it may be the case that the output is not the expected. This is exemplified in Figures 4.1

These glitches are not desirable in our training set because they may induce confusion to the network. In the case of an object disappearing the network should not flag a glitch since there is actually nothing in the image. It would not be adequate either to spot glitches that cannot be recognized by a human since this would not be needed to address in a real use case when testing, the focus overall is to locate the errors that hinder the videogame experience.

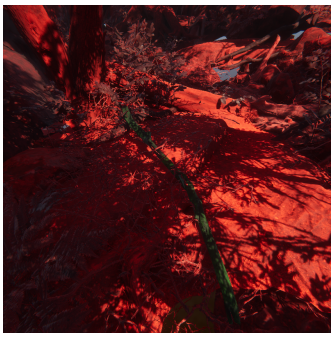
To asses how the quality of the data used to train the models the data has been manually cleaned. To do so, all the classes were visually inspected for every object, discarding all the images of one class for the object at a time if considered that these were not to be recognized by a human. The models were then trained in this dataset and the performance compared with the ones trained using the full dataset.

(a) Stretched



(b) Red and Green Mask from 4.1a

(c) Low Resolution



(d) Red and Green Mask from 4.1c

Figure 4.1: 4.1a and 4.1c show two examples of samples to be filtered. These are hard to spot for a human, to help visualizing the glitches 4.1b 4.1d show in a green tone where the glitches are located. In the case of 4.1a the object modified is grass, in some cases this objects may even disappear altogether when the texture is modified, in contrast what happens in 4.1d is that the texture is quite plain and thus deforming it generates a glitch which is hard to spot to a human eye

Placeholder Texture

As a preliminary approach, a plain white texture was used to simulate the case of a placeholder texture error. In a real use case, placeholder textures are usually made of very recognizable patterns which makes them easier to spot, since these are critical errors that should be acted on.

In a later stage of the project, we had access to a placeholder texture used at DICE. Using this texture a new Natural Environment dataset was generated, and the models trained in this new data. Using a plain white texture has the disadvantage that the model will identify white with the placeholder glitch. In

contrast, a more recognizable pattern should help with this confusion.

4.2.4 Classes

When formulating the problem, although helpful, detecting which kind of glitch is present in the image is not a fundamental part, as long as the network can flag which are the anomalous instances, it will be useful in a real use case. As a result, different settings were presented grouping the glitches according to similarity.

In the main setting throughout the project, the models were trained with the five classes described in Chapter 3. Since the classes could be grouped in corrupted textures (i.e. Stretched and Lower Resolution) and missing textures (i.e. Missing and Placeholder), this training setting was explored, but also the binary setting i.e Normal/Glitch.

To compare the different groupings the models with a finer classification would group the anomalies predicted, and the performance compared referenced to the coarser level. In the case of both approaches performing the same on the broader classification, the finer one would be deemed the best one since more information can be obtained.

4.2.5 Network Initialization

Because of fitting Convolutional Neural Networks is not a convex optimization problem, the initialization of the weights can play a big role when reaching a good local optima. The networks were initialized with the standard layer initialization provided by *torch*, which is a random initialization.

In relation to the similarity of videogame images with natural images, the model was trained using different pretrained weights on ImageNet. This means using weights from an already trained as a starting point. Torchvision provides the networks weights for some versions of ShuffleNet and ResNet.

The models were also trained using as a starting point the weights from a different glitch dataset. This was done in order to asses how easy it is for the network to transfer knowledge from one model trained into a particular environment to a new game or a new scenario.

4.2.6 Confidence Network and Activation Map

Based on the structure of the network we can exploit the model in order to provide extra information.

In order to reduce false positives, the Gauss network was trained to investigate the confidence of the network outputs. After training the network for 100 epochs the centroids were computed and the Gauss network fine tuned for different numbers of epochs.

To visualize the behavior of the network, the Activation map was generated by removing the Adaptive Average pooling layer from the Torchvision implementations of ShuffleNet and ResNet. No performance test was taken since this approach only provides information on which region is triggering most the network classification.

4.2.7 Data Aggregation

One of the main advantages of working with synthetic data is that its generation can be controlled to some degree. When testing whether the objects in the videogame scene are correctly rendered. To better asses the scene several images can be taken focusing on an object providing more input information to the network.

In order to asses how the number of images fed to the network influences the performance on classification, we have used from 1 to 20 images to classify a particular object. The final output considered for the classification consists on the average probability. The data used is the same as in the previous cases, having 20 images per object and class, to obtain one group of data for classification we select one object and class and randomly sample with replacement the corresponding number of images. This is done 100 times for every object and class to obtain a stable result

4.3 Semantic Segmentation

Although not extensively explored as in the classification approach, semantic segmentation was considered for the glitch detection problem. The main advantage of this methodology is that provides extensive information regarding where the graphical anomalies are located. In doing so, it also consumes a

lot of computational resources, being extremely slow to train with long evaluation time would arguably be too demanding and inconvenient for a real use case application.

For this approach the metrics mentioned before can be applied to this problem, we will mainly focus on the mentioned *Intersection Over Union* metric (IOU), together with the is detection criteria based on it.

One of the challenges of this approach also lays in the data provided for the algorithm. To train this it is necessary to provide a segmentation mask as a label. To do this there is not a straightforward approach to generate the mask. This process was explained in detail in section 3.2

4.3.1 Training

The Implementation of the Semantic Segmentation models was done in Python as with the classification approach, the model implementation was taken from the Torchvision library in the case of FCN and DeepLabV3 and from [62] in the case of U-Net. The models were trained with two *GeForce GTX 1080 Ti* graphics card. In this setting, the models evaluate are able to evaluate a dataset at a rate of 10, 7.2, and 6 frames per second for respectively UNet, FCN, and DeepLabV3.

Due to the demanding computational cost of both training and evaluating the models, no evaluation was done during the training. Instead, the only metric provided was the loss function, on the other hand, the models were specifically evaluated at the end of the training. As a default, the models were trained for 10 epochs.

Regarding the trainings, the optimization was done using the Adam optimizer with a learning rate $\lambda = 10^{-3}$. The high complexity of the model limited the batch size used which was fixed as 2 for all the models.

Two settings were explored for Semantic Segmentation, both a binary segmentation, providing classification for the pixels for glitch and normal as output. And a multi-class setting that specified the type of glitch (Stretched, Low Resolution, Missing, or Placeholder texture).

Chapter 5

Results

This section presents the results of the trainings performed. This chapter is divided equivalent to the sections in the previous Methodology chapter.

5.1 Classification

Under this chapter, we present the results for the main approach taken on the project. The results are presented with three structures to visualize the data: **Graphics**, **Confusion Matrices** and **Tables**

The **Graphics** present the evolution of one particular metric through the training process. The dotted line indicates the metric when the model is evaluated in the training set and the continuous line indicates the model is evaluated in the validation set. To facilitate the interpretation of the result an exponential average has been done, which filters the noise and eases seeing which is the graphs trend. Both the filtered and the unfiltered data are presented in the graph with a solid and a light line respectively.

The metrics are presented in **Tables**, which compile the statistics for the last 20 epochs (81 - 100) unless advised otherwise. Two statistics are presented with the structure *MEAN(STD)*. It was a general observation that the networks have already reached convergence with that number of epochs. The mean is presented to asses the average magnitude of the metric and the standard deviation the variability of the metrics in convergence, been able to asses significant differences with different approaches.

To give a more clear insight into how the network is performing **Confusion**

Matrices are presented in very particular cases. Since in some cases the validation sets are not perfectly balanced, and to ease the interpretability of the matrix the values presented are fractions of the total instances in the original class. The matrices are computed averaging the 20 last epochs just as with the information in the Tables. To improve the readability of the matrices higher values were presented with brighter colors and lower values with darker colors.

5.1.1 Hyperparameter Tunning

Batch Size

In order to asses how the batch size influences the training batch sizes from 2^2 to 2^5 were used for training. The metrics for the trainings are compiled in Table 5.1.

Batch Size	4	8	16	32
Accuracy	0.791 (0.012)	0.804 (0.012)	0.805 (0.011)	0.819 (0.006)
False Positive Rate	0.103 (0.034)	0.087 (0.020)	0.097 (0.027)	0.129 (0.032)
Precision	0.971 (0.009)	0.975 (0.005)	0.972 (0.007)	0.966 (0.008)
Recall	0.838 (0.031)	0.845 (0.020)	0.848 (0.024)	0.877 (0.017)

Table 5.1: Metrics table for different Batch Sizes. The models trained were ShuffleNetV2 x1.0 with using the Natural Environment Data and with a learning rate of $\lambda = 10^{-}4$

The outcome of this exploration is that the models increase performance overall when increasing the batch size. Thus, a large batch size is recommended, although, we cannot guarantee that this trend will continue indefinitely. It is likely that the capacity of the hardware used for training is reached before a bigger batch size starts to become inefficient.

Learning Rate

The learning rate parameter λ in the Adam optimizer used was explored to investigate which is the better configuration for the problem given. The metrics obtained in the training are summarized in Table 5.2

The results indicate a clear trend regarding the variance of the metrics. These become more stable as the learning rate becomes smaller. But is also interesting to see that we have an optimal learning rate regarding performance. All the

Learning Rate	10^{-2}	10^{-3}	10^{-4}	10^{-5}
Accuracy	0.715 (0.015)	0.767 (0.014)	0.801 (0.012)	0.791 (0.007)
Loss	1.635 (2.126)	1.408 (0.166)	1.072 (0.122)	0.813 (0.067)
False Positive Rate	0.270 (0.078)	0.116 (0.041)	0.085 (0.028)	0.090 (0.021)
Precision	0.928 (0.017)	0.967 (0.010)	0.975 (0.007)	0.974 (0.005)
Recall	0.859 (0.046)	0.825 (0.036)	0.835 (0.025)	0.834 (0.018)

Table 5.2: Metrics Table for different Learning rates. The models trained were ShuffleNetV2 x1.0 with using the Natural Environment Data and with a Batch Size of 8

metrics are optimal when $\lambda = 10^{-4}$. Although the difference is not significant with respect to a smaller learning rate (10^{-5}) we can see why is of our interest to choose the bigger learning rate when looking at the Graph in Figure 5.1

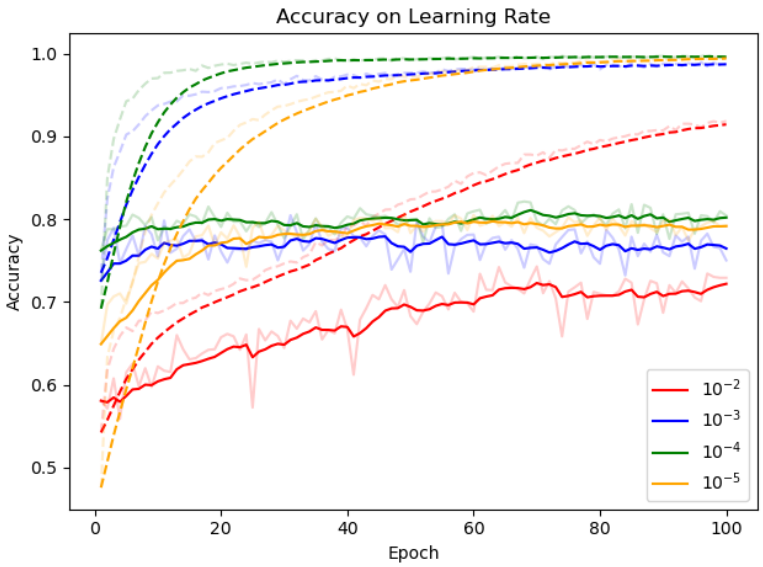


Figure 5.1: Graphics for the accuracy with different values for the learning rate λ during training. All models were trained with ShuffleNetV2, the Natural Environment data, and a batch size of 8

In the graph, we see that both $\lambda = 10^{-5}$ and $\lambda = 10^{-4}$ reach roughly the same accuracy level, but the main difference lays on when this accuracy reached. The higher learning rate reaches this performance almost after the first epoch, in contrast, we have to wait to around epoch 25 to get the same performance with the smaller one. Having a smaller learning rate means reaching the op-

timal weight configuration later thus, and this can be clearly seen with the accuracy in the training set as well.

We can also see how bigger learning rates lead to the network not reaching an optimal minimum and having quite some difficulties doing so, when the learning rate is too high ($\lambda = 10^{-2}$) the convergence is unstable. This can be noticed when looking at Table 5.2 where the variability of the loss is very high compare with the other learning rates.

In conclusion, for our problem, the most adequate learning rate was found to be $\lambda = 10^{-4}$

5.1.2 Complexity Exploration

Another aspect of the network architecture we would like to investigate is the influence of the complexity in the performance. In Figure 5.2 it can be seen how the complexity influences the ShuffleNet V2 performance.

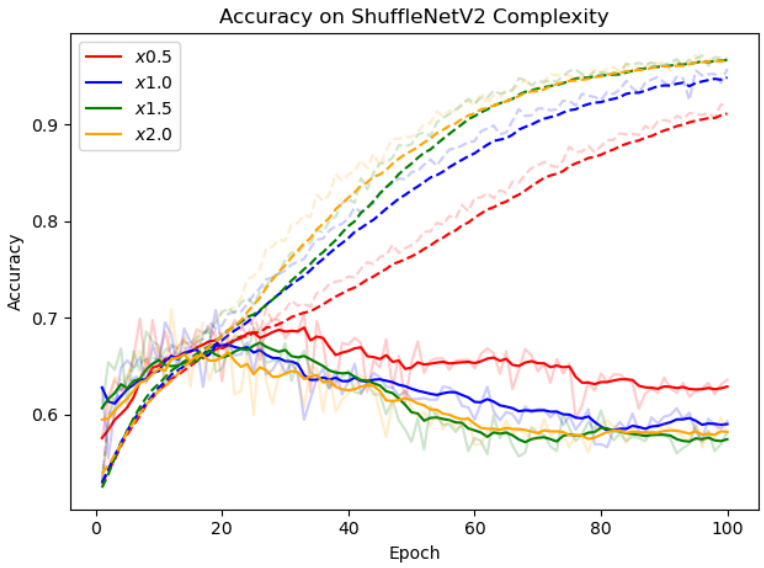


Figure 5.2: ShuffleNet Complexity exploration on not pretrained models

Note that the trainings to which the graph in Figure 5.2 correspond to are initialized randomly, instead of using the weights resulting from training in ImageNet as a starting point. This was due to the lack of pretrained models for higher complexities. More details issue with pretrained models is discussed in depth in the next Section 5.1.5.

When looking at the training curve (dashed line), it is seen that more complex models perform better in the training set. In contrast, when tested in the validation set the more complex models perform worse. This behavior is explained through the concept of overfitting. The more complex models are able to model better the intricacies of the training data reaching very high performance in that dataset. As a result, since the modeled is tailored to one particular set of data it will fail to generalize when new data is provided, the validation set. On the other hand, the less complex the model, the less it can particularize to a given datset, thus having worse performance on the training set. But, since this model does not particularize is able to generalize better, which translates into higher performance in the validation set.

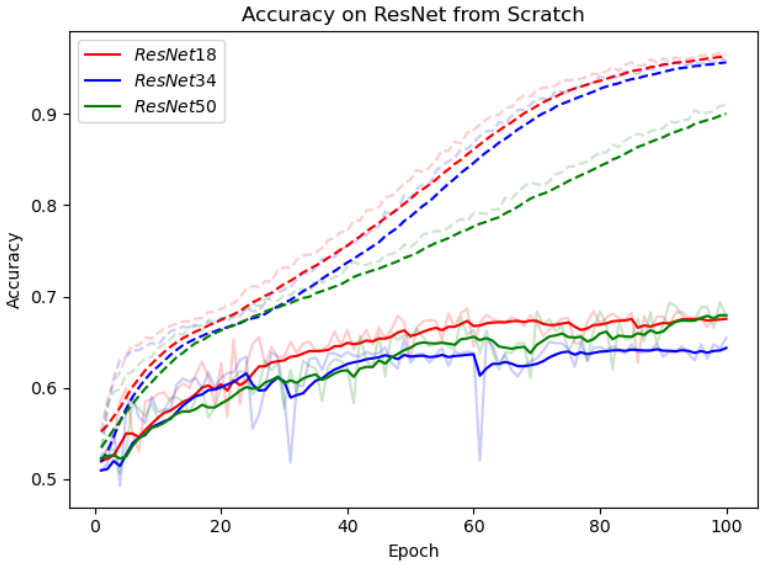


Figure 5.3: ResNet Complexity exploration. All the models were trained on the Natural Environment with batch size 8 and learning rate $\lambda = 10^{-3}$

In the same setting, the different complexities for ResNet were trained from a random initialization, the results for the accuracy are presented in Figure 5.3. The behavior, in this case, is the opposite as in the ShuffleNetV2. It has to be noted that the models have still not converged at epoch 100, and this is particularly visible with *ResNet50* where both the training and validation accuracies have increasing trends. In the graph, we see how the simpler models converge faster in the training set, but we do not see any particular pattern regarding the validation accuracy and the complexity. Slower convergence makes sense since the number of layers is the parameter modified in this case,

in contrast with the width (i.e. the number of filters) which was incremented with ShuffleNetV2.

Since the pretrained models are only available for the versions x0.5 and x1.0 of ShuffleNet, the complexity was not explored for pretrained models in this architecture. Although, ResNet does provide weights for all the models pretrained in ImageNet. In Figure 5.4 the accuracy for different complexities of ResNet is displayed for models trained for 100 epochs. In the graph, the accuracy is plotted with respect to the elapsed time to illustrate also how does the computational time demand increases with the complexity of the network. For ResNet with pretrained models it is shown that the increase of complexity just implies a longer time to fit the model but does not imply an increase in performance. In Table 5.3 it is seen how for the accuracy reached for all the ResNet architectures is the same (around 72%). This accuracy though is still significantly lower than the one reached by ShuffleNet V2.

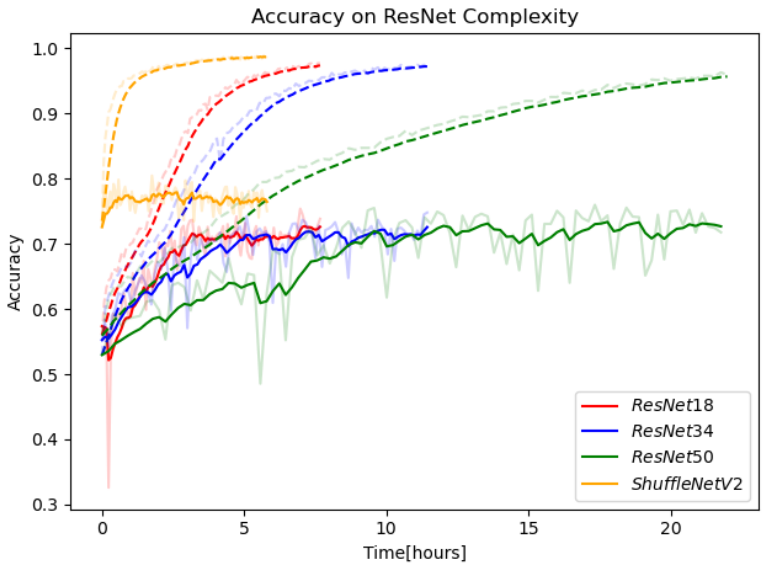


Figure 5.4: ResNet Complexity exploration on pretrained models. Note that in this case the x axis indicates the total time elapsed. All the models were trained on the Natural Environment with batch size 8 and learning rate $\lambda = 10^{-3}$

Network Architecture	ResNet18	ResNet34	ResNet50	ShuffleNetV2
Accuracy	0.721 (0.014)	0.719 (0.017)	0.724 (0.031)	0.767 (0.014)

Table 5.3: Accuracy for the last 20 epochs for the plot in Figure 5.4

The main conclusion from this complexity study is that it does not modify the performance when using pretrained models, but it does when the models are randomly initialized. This could be linked to our dataset having a too small size and diversity. ShuffleNetV2 both performs better and is faster than ResNet, which makes it our network of choice for this problem.

5.1.3 Data

In this section, we show how different characteristics of the data used, such as size, diversity, or quality may influence the results.

Size

Different sizes for training set were used for training the model in the Natural Environment. The performance metrics for this are summarized in Table 5.4. As expected, when increasing the size of the dataset used in training the performance of the model increases, this indicates that there is still a margin of improvement by adding more data to the training set.

Fraction in training	0.9	0.7	0.5	0.3
Accuracy	0.836 (0.012)	0.809 (0.011)	0.784 (0.013)	0.757 (0.012)
Loss	0.676 (0.127)	0.778 (0.117)	1.073 (0.173)	1.162 (0.095)
False Positive Rate	0.100 (0.056)	0.112 (0.051)	0.165 (0.066)	0.211 (0.058)
Precision	0.972 (0.014)	0.969 (0.013)	0.954 (0.016)	0.942 (0.013)
Recall	0.863 (0.027)	0.851 (0.029)	0.839 (0.036)	0.851 (0.032)

Table 5.4: Metrics when varying the fraction of the dataset used for training the model.

The environment used is relatively small in comparison with those used in real videogames. Analyzing the influence of adding extra data allows us to induce that in a real use case the performance reached could be higher than the one reached in this study.

Cleaning data

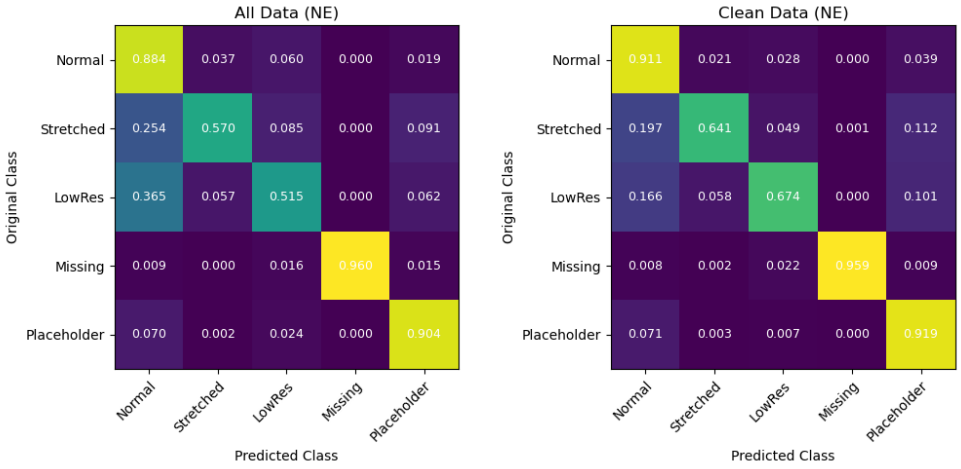
When the model was trained with better quality data, the performance increased significantly. Table 5.5 shows the metrics for the last 20 epochs. All

the metrics improve, which was to expect since the glitches in the images were mean to be easier to recognize. Except for the *False Positive Rate* metric all the other metrics would naturally increase when samples are removed from one particular class, even if the algorithm would perform exactly the same.

Data	All	Clean
Accuracy	0.767 (0.014)	0.847 (0.015)
Loss	1.408 (0.166)	1.343 (0.560)
False Positive Rate	0.116 (0.041)	0.090 (0.028)
Precision	0.967 (0.010)	0.971 (0.008)
Recall	0.825 (0.036)	0.904 (0.024)

Table 5.5: Metrics comparing the models trained with the whole dataset and with the clean data for the last 20 epochs (81-100)

As a result, we display also the average confusion matrix for the last 20 epochs. In Figure 5.5 it can be seen that when filtering the data, the proportions shift to the diagonal which indicates that more samples are classified correctly. This is noticeable in both stretched and low resolution textures since were the ones filtered. In the other cases, the performance stays the same.



(a) (b)

Figure 5.5: Cleaning data has a clear positive impact in the performance of the classification. The values showed indicate which fraction of the number of the real example falls into which spot.

From this, we can interpret that better quality data may further increase the performance of the model. Since the filtering done either removes the whole

class for one object or keeps it, much finer filtering could be done. In a real use case, it is recommended to generate good quality data, with clearly noticeable glitches to a human eye.

Environments

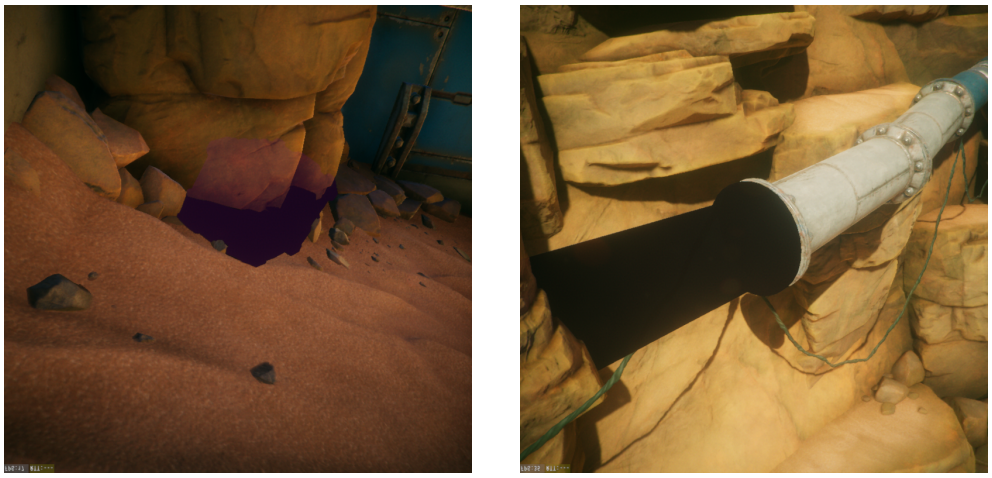
Validating on Natural, environment the model was trained in using data from different environments. The results from this training are displayed in Table 5.6. Regarding most of the metrics, we cannot see much influence of adding more data. The only exception being the false positive rate, which increases as the data gets more diverse. This is not desirable, especially in an imbalanced dataset, which will be the ultimate application. The interpretation from this is that more data implies more patterns of failure are recognized, and, since these may be specific to one environment, in other environments may trigger false alarms. One thing to highlight is that these correspond mostly to samples classified as corrupted textures instead of normal.

Datasets Used	Natural, Object, Stylized	Natural, Object	Natural
Accuracy	0.769 (0.008)	0.777 (0.009)	0.767 (0.014)
Loss	1.358 (0.137)	1.249 (0.139)	1.408 (0.166)
False Positive Rate	0.210 (0.061)	0.186 (0.051)	0.116 (0.041)
Precision	0.943 (0.014)	0.950 (0.012)	0.967 (0.010)
Recall	0.859 (0.032)	0.868 (0.028)	0.825 (0.036)

Table 5.6: Metrics table for different Environments. The metrics in this table result from evaluating the models in the validation set from the Natural data

This analysis was also done with the objects environment, whose results are presented in Table 5.7. In this case, we observe small changes that may not be completely significant but that indicate that the addition of more data slightly improves the performance. This could be thought of with the diversity of the objects data, since there are a lot of different objects seeing how new objects fail may help the model recognize what does one kind of glitch imply. In contrast in the Natural Environment, we had not a diverse set of objects, with a few main categories (tree, bushes, rocks. and logs) which had similar ways of crashing.

The last dataset we tried was the Stylized environment. The results from training are compiled in Table 5.8. When dealing with the stylized data we clearly see a big decrease in performance. This can be in part because the environ-



(a) (b)

Figure 5.6: Two different missing textures in the Stylized environment

Datasets Used	Object, Natural, Stylized	Object, Natural	Object
Accuracy	0.776 (0.009)	0.788 (0.007)	0.764 (0.013)
Loss	1.078 (0.106)	1.057 (0.074)	1.300 (0.118)
False Positive Rate	0.350 (0.064)	0.331 (0.060)	0.385 (0.055)
Precision	0.902 (0.014)	0.907 (0.014)	0.892 (0.012)
Recall	0.876 (0.023)	0.884 (0.019)	0.870 (0.019)

Table 5.7: Metrics table for different Environments. The metrics in this table result from evaluating the models in the validation set from the Object data

ments are quite different, but, if this was the case, we should observe a similar phenomenon with the previous environments. One hypothesis to explain this asymmetry can be explained by observing the dataset from the stylized environment. In Figure 5.6 we see two examples of a missing texture glitch in the Stylized environment. In contrast with the other datasets, where the images with a missing texture always looked a certain way. Is this diversity that could be responsible for this asymmetry observed.

Placeholder texture

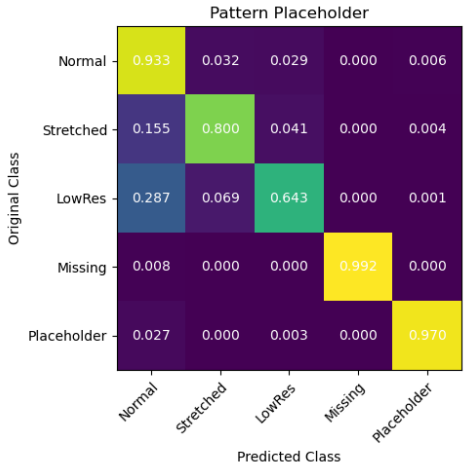
Aiming for a more realistic use case another dataset for the Natural Environment was generated using the same procedure described in Chapter 3, but using a placeholder texture with a recognizable pattern. One sample can be seen

Datasets Used	Stylized, Natural, Objects	Stylized, Natural	Stylized
Accuracy	0.776 (0.016)	0.781 (0.014)	0.828 (0.020)
Loss	1.286 (0.172)	1.104 (0.127)	0.815 (0.111)
False Positive Rate	0.117 (0.025)	0.148 (0.033)	0.070 (0.024)
Precision	0.970 (0.006)	0.963 (0.008)	0.982 (0.006)
Recall	0.939 (0.027)	0.965 (0.016)	0.983 (0.006)

Table 5.8: Metrics table for different Environments. The metrics in this table result from evaluating the models in the validation set from the Stylized data

in Figure 5.7b.

In Figure 5.7a we see the performance of ShuffleNetV2 in this dataset, here we used the best hyper parameters of obtained in Section 5.1.1. With this setting, we obtain the best performance in the report. The main effect of using the placeholder texture has been the improvement regarding images misclassified as placeholder glitches. Using a plain white placeholder texture made the model recognize white elements as glitches, providing false positives with objects that used white color. In average at convergence, the accuracy with this configuration was 86.8% with a false positive rate of 6.7% and a recall of 88.1%



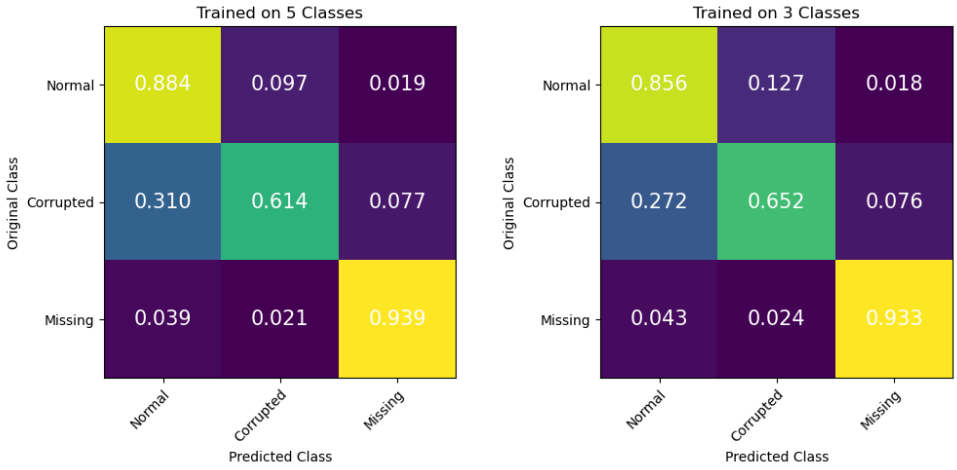
(a)

(b) Placeholder texture used

Figure 5.7

5.1.4 Classes

The assessment of how the grouping of classes influence the performance was done using the Natural Environment data, and with the standard training configuration. In Figure 5.8 the confusion matrices are presented when using both 5 and 3 classes for training. No difference is observed between the matrices which allows us to say that is preferable to train with the 5 original classes since more information is provided. Since we are using a very complex neural network this result seems natural, especially with a small dataset as it is this case.



(a) (b)

Figure 5.8: Performance training with different groupings.

The binary problem (normal or glitch) was also considered and for discussing the results all the class groupings are compiled in Table 5.9. This table also shows the performance when the classes were weighted to correct for class imbalance when grouping. We observe that no significant differences are present in the metrics of False positive Rate Precision and Recall, note that the Accuracy metrics improve with coarser grouping but this is misleading since they cannot be directly compared because of the different number of classes.

5.1.5 Network Initialization

In the previous section 5.1.2, we viewed how pretrained models performed. We justified using these models due to the similarity of the videogames images

Classes in training	5	3	3 (Weighted)	2	2 (Weighted)
Accuracy	0.767 (0.014)	0.805 (0.015)	0.810 (0.016)	0.856 (0.016)	0.834 (0.022)
False Positive Rate	0.116 (0.041)	0.144 (0.047)	0.143 (0.065)	0.213 (0.044)	0.165 (0.047)
Precision	0.967 (0.010)	0.959 (0.011)	0.960 (0.015)	0.943 (0.010)	0.953 (0.011)
Recall	0.825 (0.036)	0.842 (0.030)	0.845 (0.038)	0.873 (0.027)	0.834 (0.036)

Table 5.9: Metrics for the different groupings in training. In this table we do not include the loss metric

with natural images, and we could see that the difference between pretrained models and the ones trained from scratch was a significant better performance in the former setting.

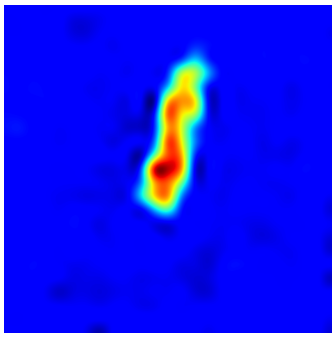
In addition to using models pretrained in ImageNet we used as initialization the weights of models trained in other environments. In particular, we used a model trained in the Natural Environment and used it as a starting point for training with the other datasets, Objects, and Stylized environment. The results are seen in Table 5.10. These results have a similar trend as in the previous section when we trained mixing different datasets. The stylized environment experiments a negative effect from using a model pretrained in the natural environment in comparison with the objects environment that does not see any difference. This could be interpreted as the pretrained guiding the weights to a specific region in the parameter space far from the local optima that is reached otherwise with the other initialization.

Environment (Pretraining)	Object (IN)	Object (NE)	Stylized (IN)	Stylized (NE)
Accuracy	0.764 (0.013)	0.765 (0.008)	0.828 (0.020)	0.771 (0.020)
Loss	1.300 (0.118)	1.363 (0.134)	0.815 (0.111)	1.570 (0.228)
False Positive Rate	0.385 (0.055)	0.325 (0.044)	0.070 (0.024)	0.132 (0.026)
Precision	0.826 (0.022)	0.837 (0.017)	0.802 (0.025)	0.752 (0.020)
Recall	0.806 (0.016)	0.790 (0.011)	0.802 (0.026)	0.747 (0.026)

Table 5.10: Table of metrics for different pretrained configurations: IN (Model pretrained on ImageNet), NE (Model pretrained on Natural Environment)

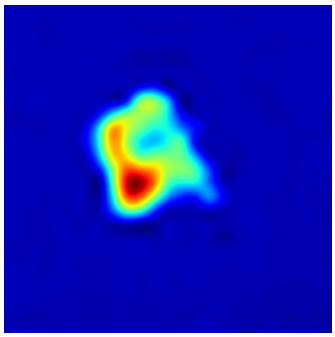
One fact that is observed when looking at the convergence using the pretrained weights in the Natural Environment is that for the case of the Objects Environment the convergence is faster. This makes sense since both have similar glitches and scenes, in contrast with the stylized environment. For a real application, this may come in useful when a model will be transferred to work

(a) Glitch: Missing



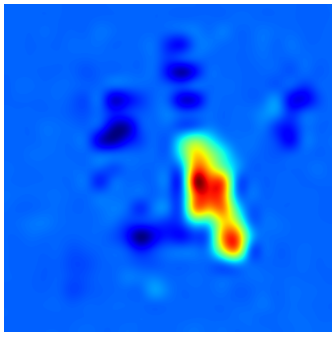
(b) Activation map from 5.9a

(c) Glitch: Placeholder



(d) Activation map from 5.9c

(e) Glitch: Stretched Texture



(f) Activation map from 5.9e

Figure 5.9: Images with glitches and their correspondent activation map.

with similar data, using the training from a previous model would not increase the performance but it would be trained faster, although, defining how similar must the datasets be could be a bit complicated.

5.1.6 Activation Maps

In this section, we present some results from the activation layer visualization. We do not present a detailed analysis of how this method can help to locate the anomalies. In Figure 5.9 we see several examples where the activation map highlights regions in the image that correspond to the region where the glitch is in the image.

5.1.7 Confidence estimation

The Gauss Network aims to model the confidence of the network, to do so trains it was trained in a loop using the training scheme provided in the original paper [46]. In Figure 5.10 the metrics considered most relevant confidence and accuracy are presented.

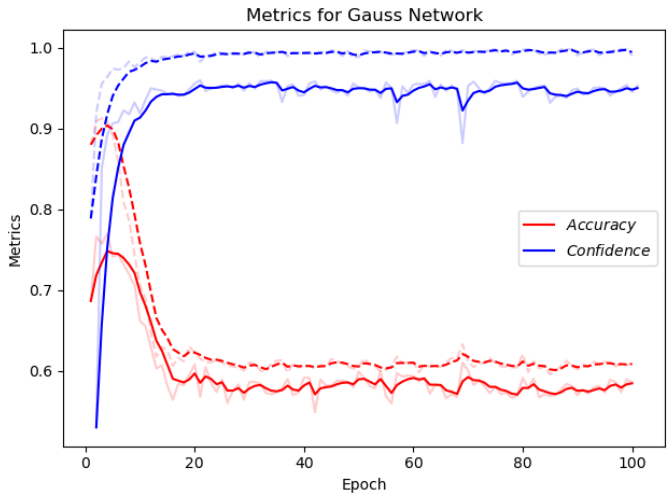


Figure 5.10: Mean Confidence and Accuracy on training and validation sets

Since the original work, where this method is presented, does not provide specific guidelines on how to apply the approach or what to expect during the training, we have decided to take the state of maximum accuracy, which corresponds to the first epochs. It also makes sense to take a state early on since otherwise, it seems we are overfitting the network, towards always having confidence close to one, which seems to indicate that the network degenerates to bring all the samples to one single point.

5.1.8 Data Aggregation

In Figure 5.11 it can be seen how the accuracy is influenced by the number of images provided for object and class. We can see a clear increasing trend, having the biggest increase in the smaller numbers.

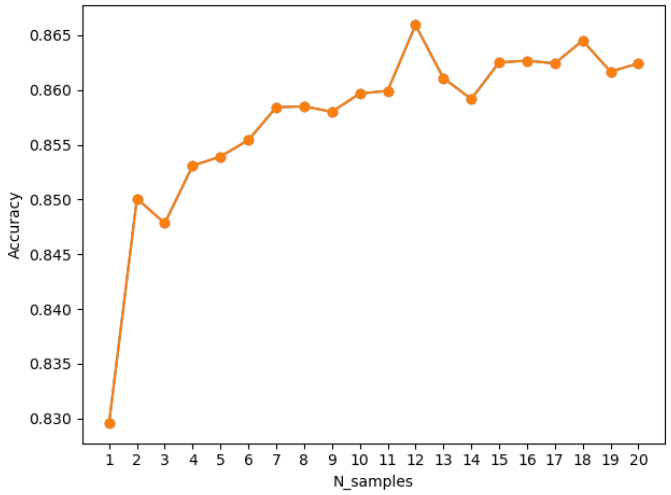


Figure 5.11: Accuracy for different number of images provided to the network

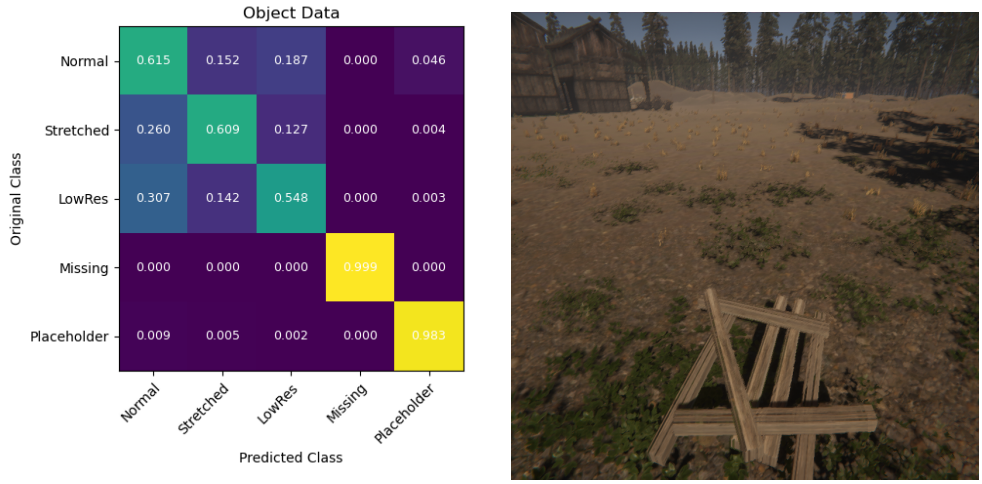
This indicates that in a real use case a limited increase of performance can be achieved by increasing the number of images fed to the network. Although, it has to be considered the considerable computation overhead since the number of computations will scale linearly with the number of images provided to the network.

5.1.9 Testing Data

New Objects

The performance of the model when looking into new objects can be understood through the confusion matrix, shown in Figure 5.12a. Here it can be seen that as in the Natural environment 5.5a the main confusion is within the corrupted textures: Stretched and Lower resolution.

When looking into some examples such as the one shown in Figure 5.12b We see that the textures are really similar to the original texture, thus it makes sense for the model to get confused between the three classes. In fact, the



(a) Confusion matrix for new objects

(b) Sample of stretched textures

Figure 5.12

actual stretched textures should also deform the object, since this was not accomplished in Unity, with our limited knowledge, we could not use samples as the one in Figure 3.1a

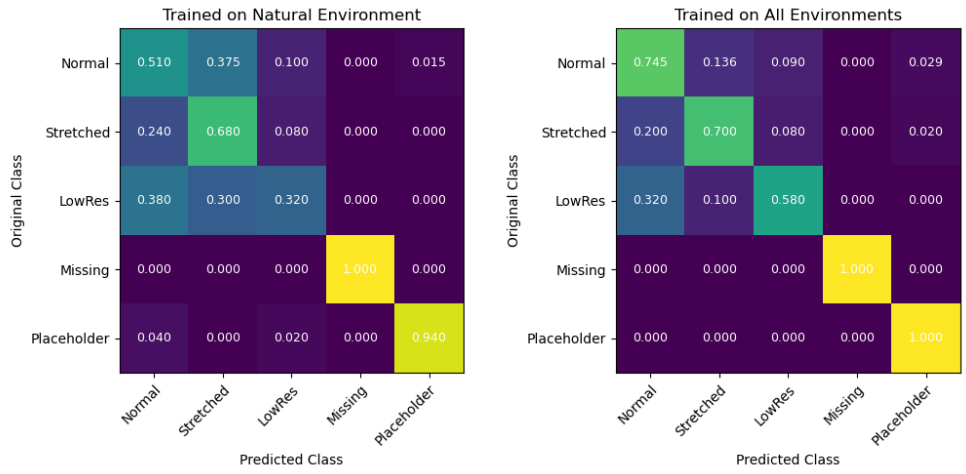
Natural environment + Objects

The models from the previous sections were applied to the testing data obtained from combining both the Natural and objects environment. As mentioned in Section 3.2 the proportion for glitches in the data is low, to simulate a real use case.

Both the model trained only on the Natural Environment and the one Trained in all environments were used with this data. Figure 5.13 shows the confusion matrices for both cases.

As it was to expect the model trained in all the environments performed better. It is important also to consider that the objects were input for training the networks in the objects environment. This means that even if they were fed in a different environment the model "learns" the way how the objects fail thus increasing the performance with the test dataset.

When evaluated with the model trained in all the environments, the model reaches a precision of 24 % which means that in a real use case when reviewing the samples marked as faulty one in every four does contain a glitch, which is



(a) (b)

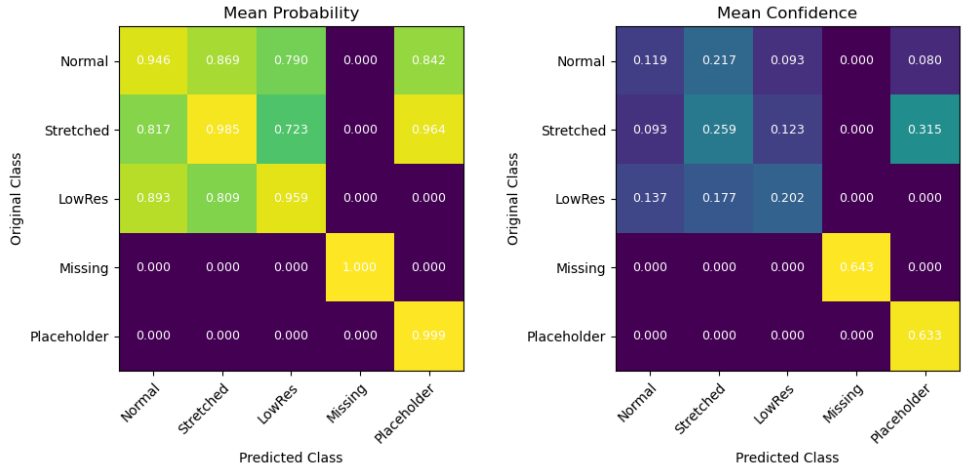
Figure 5.13: Confusion matrices relative to the total number of samples from one class.

a reasonable proportion if these were to be reviewed manually.

To further reduce the number of false positives the confidence measure for the network becomes quite useful. In Figure 5.14 the mean probability (direct output from the network) and confidence for every cell of the confusion matrix is compiled, zero is displayed when there are no samples in the correspondent cell. When we look at these measures, the mean probability is distributed with high values through the whole matrix, while the confidence shows a clear distinction between the upper tree rows, with lower confidences. And we can see how the confidence does correspond to the intuition built in the project that the network does have trouble classifying corrupted textures. Potentially the confidence would be useful to filter out false positives, although this would come at a cost of a lower recall.

5.2 Semantic Segmentation

This approach achieves pixel-level classification, providing a label for each of the pixels in the image. Three models were tested for this approach although many shortcomings were encountered starting from the definition of the targets. The Semantic Segmentation used two settings, binary segmentation and multiple label segmentation. The first one only provides information about



(a) Mean Probability (b) Mean Confidence

Figure 5.14: Confusion matrices relative to the total number of samples from one class.

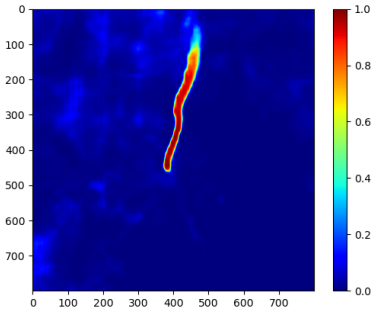
whether the pixel is part of a glitch or not, and the second states which kind of glitch does it correspond to.

	Training		**Validation**	
	IoU (Mean/Median)	Detection (0.3 IoU / 0.5 IoU)	IoU (Mean/Median)	Detection (0.3 IoU / 0.5 IoU)
UNet	0.316/0.002	0.281/0.269	0.323/0.002	0.278/0.266
FCN	0.499/0.502	0.525/0.431	0.503/0.523	0.531/0.441
DeepLabV3	0.486/0.535	0.651/0.557	0.483/0.533	0.659/0.559

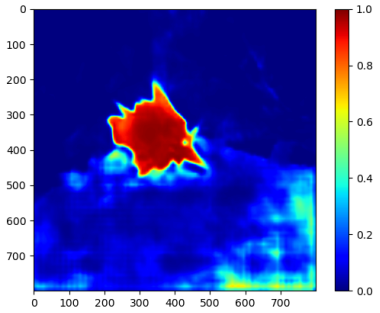
Table 5.11: Metrics for semantic segmentation, with one class classification configuration

In Table 5.11 the performance metrics for the binary segmentation are presented. The metrics correspond to the anomalous class and are stated for both the training set and validation set. In both networks no overfitting is observed, the performance is better in the case of DeepLabV3, which we also should mention is slower that FCN. In the case of UNet the network is faster due to its simpler structure but the performance is greatly reduced. When looking at the mean and Median IoU it can be induced that this metric takes extreme values either close to 0 or to 1, explaining the disparity between these metrics.

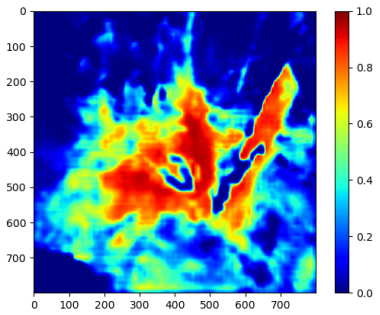
In Figure 5.15 we see the output for the samples seen in Figure 5.9. We can



(a) Anomaly Heatmap from 5.9a



(b) Anomaly Heatmap from 5.9c



(c) Anomaly Heatmap from 5.9e

Figure 5.15: Anomaly probability maps form the samples in 5.9

using DeepLabV3. It can be seen that the performance is good in the classes Placeholder and Missing, but does a bad job with the image with a stretched texture. This can be understood greatly due to the quality of the target generated for training, in the case of the stretched and Low Resolution textures, the masks obtained could vary in quality depending on the object. Also, another factor is that missing texture glitches seem easier to recognize as seen in the classification approach. When analyzed, most of the glitches detected correspond to these two glitch types.

This ease for the network to detect certain kind of glitches is clearly seen in Tables 5.12 and 5.13. Here it is interesting to see how misleading can actually be the mean IoU. In this case, since most of the images are negative examples (there is no anomalous pixel in the sample for 80% of the images) the value of the metric is quite high but, in contrast, the detection rate is low for both networks regarding the stretched and low resolution glitches. It is also inter-

esting to observe that neither method excels in the task of detecting missing textures. Both UNet and FCN detect almost all the missing textures but ignore completely the placeholder glitches, while in the case of DeepLabV3 detects both kinds of glitches, but certainly not most of them. The surprising part is how the setting of the problem made UNet perform relatively better compared to the one class classification problem.

Mean IoU	Stretched	Low Resolution	Missing	Placehholder
UNet	0.825	0.829	0.973	0.785
FCN	0.581	0.697	0.975	0.789
DeepLabV3	0.735	0.788	0.900	0.787

Table 5.12: Mean IoU metric for each of the anomaly classes on the validation set. Note that as in the case of binary segmentation the results did not present signs of overfitting.

Detection (0.5 IoU)	Stretched	Low Resolution	Missing	Placehholder
UNet	0.106	0.081	0.967	0.097
FCN	0.060	0.081	0.980	0.097
DeepLabV3	0.143	0.083	0.576	0.611

Table 5.13: Detection rate with the 0.5 IoU criteria for each of the anomaly classes on the validation set. Note that as in the case of binary segmentation the results did not present signs of overfitting.

We have not performed an extensive analysis of the possibilities of Semantic Segmentation for this problem, but we do not discard the idea of this being an interesting approach when some difficulties are overcome. The main obstacle was regarding data, certainly, the targets regarding the corrupted texture glitches were not adequate, and this may have driven down the performance in these classes. On the other hand, the data is not diverse enough, it corresponds to object centered images, where the glitches are always placed in the center. This makes the Semantic Segmentation approach a waste of computational resources since the information regarding where the glitch is located can be deemed as redundant. In an environment were glitches are placed uniformly across the image this approach may be better made use of.

Chapter 6

Future Work

In this chapter, we discuss the main limitations of the project and how these could be addressed in future projects.

6.1 Data

One of the main limitations of this project was the data provided. Unity provided us with data good enough to do a proof of concept on using CNNs to detect graphical glitches. But more diverse and better quality data would be needed for obtaining a better performing model or to deal with different use cases.

6.1.1 Glitch quality

The generated glitches like stretched textures or lower resolution textures were deemed hard to identify even for a human eye in some of the samples used in this project. On top of that we had seen that the stretched textures would have a very different appearance in a real scenario (Compare Figures 3.1a and 3.4a). Thus, generating a more accurate dataset in relation to what is seen in real life is required for a better model.

The setting used refers to a very diverse set of objects where the glitch takes place. When developing videogames there are particular objects that may be of interest to model. For instance, if provided a model specific to detect glitches in weapons or vehicles. Exploring generating and training a particular class

of objects may improve the results, obtaining a very specific model but with much higher performance

6.1.2 Other use cases

The current setting for capturing glitches is limited by the unity framework we used. In order to generate glitches in a systematic way, the object was purposely centered in the image, thus guaranteeing that the samples generated as glitch would always present a glitch in the image.

As a result, the use case for the models trained with this data is limited to captures where the object is centered in the image. When looking for graphical malfunctioning in videogames this is one of the settings, but other configurations like flying cameras or character cameras are also used. The data generated would then have to account for glitches not being centered, and even the possibility of several types of glitches in the same image

6.2 Methods

Both classification and Semantic Segmentation were used in this work. Although classification was the way to go due to the high computation requirements of Semantic Segmentation, other approaches could also be realizable for detecting glitches.

6.2.1 Object Detection

Linked to detecting glitches in different settings object detection is presented as an interesting solution when we consider the possibility of a glitch not centered in the image, or several glitches in the same image.

The object detection problem deals with identifying and locating objects of certain classes in an image, by creating a bounding box around the object and indicating which class this belongs to. From the first real-time human face detector [63] during the last 20 years extensive research has been done in this area [64].

In the current state of the art, methods are divided in one stage detectors that directly output both bounding boxes and a label, and two stage, which first

provide box proposals and then a classification for the object. In both approaches real-time state of the art methods are currently developed, highlighting YOLOV3 in one stage [65] and Faster RCNN in two stage [66]

6.2.2 Non Supervised

The current approach focuses on using a supervised configuration, accounting for very common glitches whose mechanisms are well known and that are critical enough to require actions from the developing teams. This set of glitches though does not comprise many other common glitches that occur on videogames.

Accounting for the glitches being unexpected and diverse, another kind of approach is necessary to address the problem, one that is able to detect unexpected failures that the model has not seen before. In this setting Section 2.1.2 and 2.1.3 already discussed the availability of several methods that work with image and are able to detect unexpected samples. Applying unsupervised and semisupervised approaches to this problem, although not a straightforward task could be an interesting line of research.

Chapter 7

Conclusions

This study presents a simple yet effective approach for detecting graphical anomalies in videogames images. Making use of a supervised approach with a CNN we were able to detect 88.1% of the glitches with a false positive rate of 6.7% and a classification accuracy of 86.8%. Such results allow us to answer affirmatively the research question stated in the beginning: *Can we detect videogame glitches using CNNs?*

On top of that several considerations regarding data, architecture, hyperparameters and network initialization are discussed providing general guidelines for future methods addressing the problem. Additionally, different ways to exploit the internal architecture of the networks were presented allowing to better understand the behavior of the model (Activation maps) and obtaining a measure of confidence for the networks, these having potential uses in localizing the glitch and filtering out false positives respectively. Taking advantage of the particularities of the problem we describe how controlling the camera allows us to provide more information to the network, thus increasing the performance on classification.

The fundamental implication of this study is that CNNs can be used in a production environment to improve the testing process in videogames, partially automating a currently complete manual process. The architecture used was able to evaluate images at a rate of 60 frames per second, being computationally light enough to be running in parallel with the videogame rendering allowing for real-time assessment on the video quality.

Furthermore, since the methods presented are trained directly in the images from the game, glitch detection is presented as an independent block, provid-

ing high versatility when used in the actual testing pipeline. This will allow for diverse uses like processing all the images during gameplay, focusing only on particular objects or running completely separated from the game using recorded images.

Nevertheless, one of the main limitations of the method presented is being supervised which means that specific data exemplifying both normal and anomalous images have to be provided. Although some level of extrapolation was displayed by the model, this approach would be useless during the first stages of game development, since no data is available. These methods prove useful only later stages of the development when operations like migrating to a new version (another console platform / graphical engine updates) or adding new objects to the game, both cases in which sample data is available.

On the other hand, in this work, we have only explored one of many possible approaches to the problem, by restricting ourselves to a very particular subset of glitches. Further research is needed to asses whether other problems in testing could also be addressed, focusing on other kinds of glitches and not only limited to static images but also including video sequences.

Bibliography

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[2] H. Wang, M. J. Bah, and M. Hammad, "Progress in outlier detection techniques: A survey," *IEEE Access*, vol. 7, pp. 107 964–108 000, 2019.

[3] W. Sultani, C. Chen, and M. Shah, "Real-world anomaly detection in surveillance videos," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 6479–6488.

[4] A. Khaleghi and M. S. Moin, "Improved anomaly detection in surveillance videos based on a deep learning method," in *2018 8th Conference of AI & Robotics and 10th RoboCup Iranopen International Symposium (IRANOPEN)*. IEEE, 2018, pp. 73–81.

[5] C. Baur, B. Wiestler, S. Albarqouni, and N. Navab, "Deep autoencoding models for unsupervised anomaly segmentation in brain mr images," in *International MICCAI Brainlesion Workshop*. Springer, 2018, pp. 161–169.

[6] T. Schlegl, P. Seeböck, S. M. Waldstein, U. Schmidt-Erfurth, and G. Langs, "Unsupervised anomaly detection with generative adversarial networks to guide marker discovery," in *International conference on information processing in medical imaging*. Springer, 2017, pp. 146–157.

[7] T. Schlegl, P. Seeböck, S. M. Waldstein, G. Langs, and U. Schmidt-Erfurth, "f-anogan: Fast unsupervised anomaly detection with generative adversarial networks," *Medical image analysis*, vol. 54, pp. 30–44, 2019.

[8] M. Ahmed, A. N. Mahmood, and M. R. Islam, "A survey of anomaly detection techniques in financial domain," *Future Generation Computer Systems*, vol. 55, pp. 278–288, 2016.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[10] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[11] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 116–131.

[12] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: A survey," *arXiv preprint arXiv:1901.03407*, 2019.

[13] G. Georgoulas, T. Loutas, C. D. Stylios, and V. Kostopoulos, "Bearing fault detection based on hybrid ensemble detector and empirical mode decomposition," *Mechanical Systems and Signal Processing*, vol. 41, no. 1-2, pp. 510–525, 2013.

[14] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.

[15] D. Pokrajac, A. Lazarevic, and L. J. Latecki, "Incremental local outlier detection for data streams," in *2007 IEEE symposium on computational intelligence and data mining*. IEEE, 2007, pp. 504–515.

[16] G. G. Hazel, "Multivariate gaussian mrf for multispectral scene segmentation and anomaly detection," *IEEE transactions on geoscience and remote sensing*, vol. 38, no. 3, pp. 1199–1211, 2000.

[17] S. M. Anwar, M. Majid, A. Qayyum, M. Awais, M. Alnowami, and M. K. Khan, "Medical image analysis using convolutional neural networks: a review," *Journal of medical systems*, vol. 42, no. 11, p. 226, 2018.

[18] M. S. Minhas and J. Zelek, "Anomaly detection in images," *arXiv preprint arXiv:1905.13147*, 2019.

[19] N. Görnitz, M. Kloft, K. Rieck, and U. Brefeld, "Toward supervised anomaly detection," *Journal of Artificial Intelligence Research*, vol. 46, pp. 235–262, 2013.

[20] L. Ruff, R. A. Vandermeulen, N. Görnitz, A. Binder, E. Müller, K.-R. Müller, and M. Kloft, "Deep semi-supervised anomaly detection," *arXiv preprint arXiv:1906.02694*, 2019.

[21] D. Wulsin, J. Blanco, R. Mani, and B. Litt, "Semi-supervised anomaly detection for eeg waveforms using deep belief nets," in *2010 Ninth International Conference on Machine Learning and Applications*. IEEE, 2010, pp. 436–441.

[22] M. Nadeem, O. Marshall, S. Singh, X. Fang, and X. Yuan, "Semi-supervised deep neural network for network intrusion detection," 2016.

[23] H. Estiri and S. N. Murphy, "Semi-supervised encoding for outlier detection in clinical observation data," *Computer methods and programs in biomedicine*, vol. 181, p. 104830, 2019.

[24] J. An and S. Cho, "Variational autoencoder based anomaly detection using reconstruction probability," *Special Lecture on IE*, vol. 2, no. 1, 2015.

[25] H. Zenati, C. S. Foo, B. Lecouat, G. Manek, and V. R. Chandrasekhar, "Efficient gan-based anomaly detection," *arXiv preprint arXiv:1802.06222*, 2018.

[26] S. Akcay, A. Atapour-Abarghouei, and T. P. Breckon, "Ganomaly: Semi-supervised anomaly detection via adversarial training," in *Asian Conference on Computer Vision*. Springer, 2018, pp. 622–637.

[27] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[28] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[29] F. T. Liu, K. M. Ting, and Z.-H. Zhou, "Isolation forest," in *2008 Eighth IEEE International Conference on Data Mining*. IEEE, 2008, pp. 413–422.

[30] C. Liu, S. Ghosal, Z. Jiang, and S. Sarkar, "An unsupervised spatiotemporal graphical modeling approach to anomaly detection in distributed cps," in *2016 ACM/IEEE 7th International Conference on Cyber-Physical Systems (ICCPS)*. IEEE, 2016, pp. 1–10.

[31] T. Ergen, A. H. Mirza, and S. S. Kozat, "Unsupervised and semi-supervised anomaly detection with lstm neural networks," *arXiv preprint arXiv:1710.09207*, 2017.

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*, 2016.

[33] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[34] J. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of machine learning research*, vol. 12, no. Jul, pp. 2121–2159, 2011.

[35] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[36] B. C. Csáji *et al.*, "Approximation with artificial neural networks," *Faculty of Sciences, Etvs Lornd University, Hungary*, vol. 24, no. 48, p. 7, 2001.

[37] Z. Lu, H. Pu, F. Wang, Z. Hu, and L. Wang, "The expressive power of neural networks: A view from the width," in *Advances in neural information processing systems*, 2017, pp. 6231–6239.

[38] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[39] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *arXiv preprint arXiv:1704.04861*, 2017.

[40] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 6848–6856.

[41] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500.

[42] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2921–2929.

[43] T. DeVries and G. W. Taylor, "Learning confidence for out-of-distribution detection in neural networks," *arXiv preprint arXiv:1802.04865*, 2018.

[44] A. Subramanya, S. Srinivas, and R. V. Babu, "Confidence estimation in deep neural networks via density modelling," *arXiv preprint arXiv:1707.07013*, 2017.

[45] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[46] S. Hess, W. Duivesteijn, and D. Mocanu, "Softmax-based classification is k-means clustering: Formal proof, consequences for adversarial attacks, and improvement through centroid based tailoring," *arXiv preprint arXiv:2001.01987*, 2020.

[47] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results," http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

[48] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, and J. Garcia-Rodriguez, "A review on deep learning techniques applied to semantic segmentation," *arXiv preprint arXiv:1704.06857*, 2017.

[49] M. Cordts, M. Omran, S. Ramos, T. Rehfeld, M. Enzweiler, R. Benenson, U. Franke, S. Roth, and B. Schiele, "The cityscapes dataset for semantic urban scene understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 3213–3223.

[50] B. Kayalibay, G. Jensen, and P. van der Smagt, "Cnn-based segmentation of medical imaging data," *arXiv preprint arXiv:1701.03056*, 2017.

[51] J. M. Wong, V. Kee, T. Le, S. Wagner, G.-L. Mariottini, A. Schneider, L. Hamilton, R. Chipalkatty, M. Hebert, D. M. Johnson *et al.*, "Segicp: Integrated deep semantic segmentation and pose estimation," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 5784–5789.

[52] M. Wurm, T. Stark, X. X. Zhu, M. Weigand, and H. Taubenböck, "Semantic segmentation of slums in satellite images using transfer learning on fully convolutional neural networks," *ISPRS journal of photogrammetry and remote sensing*, vol. 150, pp. 59–69, 2019.

[53] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in *International Conference on Medical image computing and computer-assisted intervention*. Springer, 2015, pp. 234–241.

[54] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *arXiv preprint arXiv:1706.05587*, 2017.

[55] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3431–3440.

[56] "Battlefield V — DICE." [Online]. Available: https://www.dice.se/game/battlefield-v

[57] New Textures & Standard Pipeline Conversion for Book of the Dead | 3D Landscapes | Unity Asset Store. [Online]. Available: https://assetstore.unity.com/packages/3d/environments/landscapes/new-textures-standard-pipeline-conversion-for-book-of-the-dead-113784

[58] "Apex Legends | Respawn Entertainment." [Online]. Available: https://www.respawn.com/games/apex-legends

[59] "FPS Sample - A multiplayer shooter game project | Unity." [Online]. Available: https://unity.com/fps-sample

[60] "NVIDIA® Viking Village | Tutorials | Unity Asset Store." [Online]. Available: https://assetstore.unity.com/packages/templates/tutorials/nvidia-viking-village-88651

[61] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation," *Journal of Machine Learning Technologies*, vol. 2, no. 1, pp. 37–63, 2011. [Online]. Available: http://dspace.flinders.edu.au/dspace/http://www.bioinfo.in/contents.php?id=51

[62] "GitHub - milesial/Pytorch-UNet: PyTorch implementation of the U-Net for image semantic segmentation with high quality images." [Online]. Available: https://github.com/milesial/Pytorch-UNet

[63] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1. IEEE, 2001, pp. I–I.

[64] Z. Zou, Z. Shi, Y. Guo, and J. Ye, "Object detection in 20 years: A survey," *arXiv preprint arXiv:1905.05055*, 2019.

[65] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[66] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

TRITA -SCI-GRU 2020:090