G-EOT - Generative network for black-box adversarial attacks for 3D objects

Alexandru Dascălu
965337

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Master of Science

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

September 30, 2022

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

I would like to dedicate this work to my family, for raising me and supporting me to be where I am now.

Abstract

Ever since the publication of the AlexNet architecture [1] in 2012, deep neural networks have seen an explosion in popularity, and are used in a wide variety of applications, like malware detection, biometric security, self-driving cars, etc. However, it has been proven that they are vulnerable to adversarial attacks, where subtle perturbations are added to the normal input to fool the victim model into giving a specific wrong output. Neural networks used in safety-critical applications can therefore be a cybersecurity risk.

Existing attack methods that work in physical settings, not just on 2D images, are white-box, they require full access to the victim model. Another line of research created attack methods against black-box victim models. In this project, I present G-EOT, the first generative model that attempts to create physical 3D adversarial objects that fool a black-box victim. It is a combination of the EOT framework [2] for creating 3D adversarial objects and a generator-simulator model [3] for black-box attacks.

The experiments show that the model is unable to learn much throughout its training history. They suggest that the main issue is the large variation of the gradient between optimisation steps, due to the rendering of the object in random poses, which is necessary for creating robust adversarial noise for 3D objects. Further work is needed to see if the issue can be overcome.

Finally, the experiments suggest show that a black-box version of EOT would be far more convenient for an attacker, and the thesis discusses how that could be achieved. On the other hand, a successful version of G-EOT would be very useful for augmenting datasets to make neural networks impervious to adversarial attacks.

Acknowledgements

First of all, I would like to express my deep gratitude to my mother for all the emotional support and encouragement that she has given me over the past academic year and for being there when I needed her.

Furthermore, I would like to thank Dr Jingjing Deng for being a great supervisor and for guiding me towards realistic project goals. In addition, Mr Hanchi Ren and Dr Mike Edwards greatly helped me with technical questions related to Tensorflow.

Moreover, I want to thank Mr Szabolcs Dombi, the developer of ModernGL, for his quick replies to my emails in regards to technical issues related to ModernGL. I am also grateful to user "einarf" from the ModernGL Discord for his help.

Finally, I am thankful to my former coursemates Avi Varma and Radu Bucurescu. Avi gave me support and suggestions and shared with me things he learnt when doing his project a year ago, while Radu taught me the basics of OpenGL rendering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the past 10 years, deep neural networks have advanced tremendously, and have been used for safety and security-critical tasks, such as face recognition [6], self-driving cars [7] and malware detection [8]. However, neural networks are vulnerable to a variety of attacks which enable malicious agents to manipulate their outputs or leak information [9–12]. Adversarial attacks are one such class of attacks. First discovered in 2014 by Szegedy *et al.* [12], they involve crafting special perturbations that are added to the original input to make the victim model predict a wrong label. Moreover, these perturbations are often imperceptible to humans [12] and can fool a model into outputting specific wrong labels which are very different to the ground truth.

Since then, researchers have proven that attacks in the physical world [13] and in cyberspace [14] are possible. For example, Eykholt *et al.* [13] successfully performed an adversarial attack that fooled a road sign recognition model in real life. Athalye *et al.* [2] created physical 3D objects that consistently fool neural networks for object recognition, regardless of the camera angle. This was in contrast to traditional attacks that create 2D adversarial images, as those are much less effective when printed and physically held in front of a camera [15]. Consequently, the existence of adversarial examples undermines the safety of neural networks, and thus their applicability in safety or security-critical systems.

One disadvantage of the physical world attacks created in [2] and [13] is that they are **white-box** attacks, they require access to the victim model. This is necessary because they rely on gradient-descent optimisation and thus need to differentiate through the victim model. Another line of research looked at **black-box** attacks, which do not have this requirement [16] and are thus more practical for the attackers. However, the black-box attacks in the literature only

create 2D adversarial examples in the lab setting and are less effective in a physical scenario.

An adversarial attack that is both black-box and applicable to the physical domain would be especially practical for attackers. Therefore, my project will try to combine the framework used in Athalye *et al.* [2] and the generative model presented in [3] to create a black-box adversarial attack. It which produces 3D rendered objects that can consistently fool object classifier neural networks. The findings may be useful for understanding adversarial attacks further and for creating better defence methods.

1.1 Motivation

Adversarial attacks are often highly effective at fooling a victim model [16–20]. Hendrik Metzen *et al.* [21] attacked an image segmentation neural network used in a self-driving car. They managed to make the victim model not perceive pedestrians, as 84.5% of pixels making up pedestrians were misclassified. If an attacker were to implement this in real life, the self-driving car could have run over pedestrians, because it would not see them, as figure 1.1 on page 3 shows. Therefore, adversarial examples are a threat to neural networks used in safety-critical systems.

Most attacks, including the one in figure 1.1, directly add the image perturbations to the whole input image. In a realistic setting, the attacker would not be able to directly manipulate the input to a neural network, especially when the neural network takes its input from sensors. Examples include robots and self-driving cars which use cameras and voice command systems. On the other hand, Eykholt *et al.* [13] created adversarial patches which were stuck over a real STOP road sign. They then took pictures of the sign from a moving car and found that they were misclassified 84.8 and 87.5% of the time by two different neural networks, respectively. Their attack would have made a self-driving car not stop at an intersection, and potentially cause a car crash.

The attacks against cyber-physical systems presented in [2] and [13] are **white-box**, they require access to the victim model's architecture and parameters. Traditional security measures such as encryption and network access control could secure the model and prevent attacks. But **black-box** attacks do not need access to the victim, thus bypassing those security measures. Consequently, they are more likely to be used by malicious agents.

Therefore, black-box attacks which also work in physical settings are the attacks most likely to be used by malicious agents. However, to my knowledge, no attack in the literature has both of these characteristics. The black-box attacks in [3, 5, 22] only create adversarial

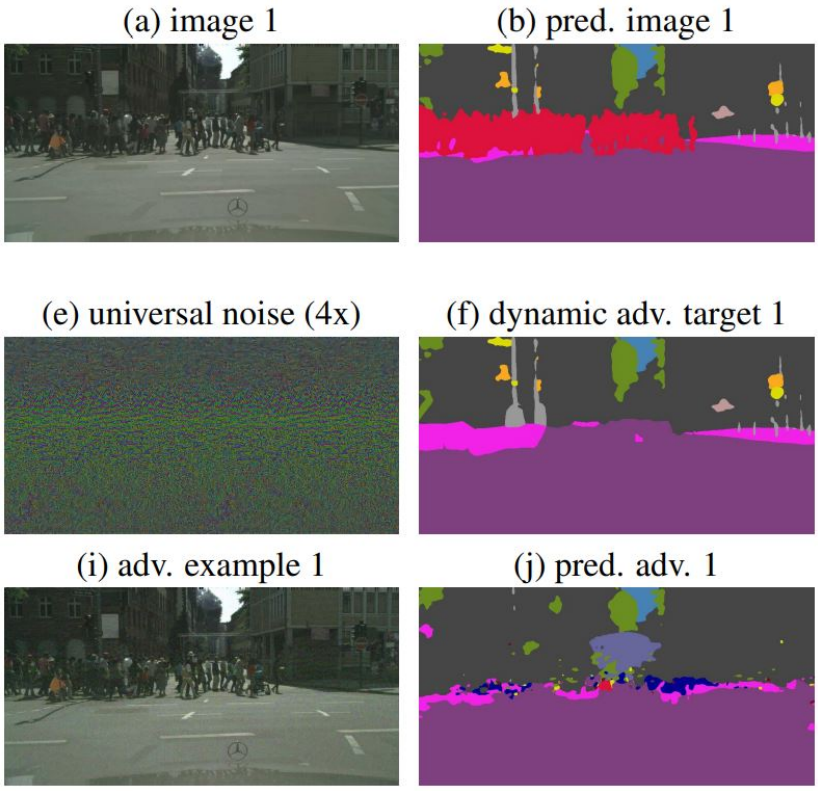

Figure 1.1: Example of an adversarial attack which removes pedestrians from a scene, as perceived by a neural network. The top row contains the original image and the correct image segmentation. The middle row shows the adversarial noise and the target segmentation, which has the pedestrians removed from the scene. The last row shows the adversarial example and the predicted segmentation. Taken from [21].

noise for 2D images. The purpose of this project is to create a generative model for creating 3D adversarial objects [2] and see if it is effective. I specifically plan to use generative models for creating black-box attacks because they can produce adversarial examples almost instantly [5] once trained, rather than the hours it would take with an optimisation method. Moreover, they have high fooling rates [3, 5, 22].

Creating new, powerful and practical attack methods has several benefits. Firstly, the experimental results may reveal further insight into the nature of adversarial attacks and how to defend against them. Secondly, it may raise awareness of the risks involved with using ML models in physical systems. Furthermore, it is important to assess the robustness of neural networks against adversarial attacks by testing them with a variety of strong attacks, such as the proposed attack method. The proposed attack method may be used in future benchmarks

to test the robustness of models. Finally, the proposed attack could be useful for improving the robustness of models via adversarial training. It is one of the more effective defence methods [18], where adversarial examples are labelled with the ground truth label and are included in the training set. It is important to augment the training set with adversarial examples made with powerful attack methods.

1.2 Scope and limitations

Although the motivation behind the project is to create a black-box attack method applicable to physical world attacks, the proposed method will only be evaluated on 3D rendered objects. EOT [2] proved to be highly effective for creating both 3D rendered adversarial objects and also 3D printed physical adversarial objects. However, I do not have access to the kind of high-quality 3D printer that Athalye *et al.* [2] had. I anticipate that simulating the 3D printing error as Athalye *et al.* [2] did would make EOT-GAN effective for 3D adversarial objects.

1.3 Aims and objectives

The project aims to establish the feasibility of using a generative network to create black-box adversarial perturbations for a 2D texture, which when rendered into a 3D object in various poses, manages to fool the target network.

Therefore, the objectives of the project are as follows:

- Implement the generative network and repeat some of the experimental results shown in [3]. This is necessary because the proposed model is a modified version of the model in [3].
- Implement the method shown in [2] and repeat the experimental results for 3D rendered objects in section 3.3 of [2]. This is necessary because the rendering pipeline from figure 3.1 in section 3.4 should work like the one in [2].
- Create a generative network that is capable of making adversarial perturbations that consistently fool a neural network classifier, when those perturbations are applied to the texture of a 3D rendered object, regardless of the rotation or position that the object is rendered in.

- Evaluate the new attack method by measuring how well the attack fools a neural network, on the dataset of 3D models with textures described in section 4.1.

1.4 Contributions

The main contributions of this work are as follows:

- **A dataset of 15 labelled 3D models, along with associated textures, representing classes from the Imaganet dataset**
- **A complete, clear and well-documented implementation of the EOT framework [2] for 3D rendered objects:** The existing implementation does not implement all aspects of EOT seen in [2], uses deprecated libraries and lacks documentation.
- **Analysis of the experimental results of using EOT to create adversarial attacks for 3D objects:** The shape and colour of the 3D models impact the performance and runtime of EOT, and the adversarial noise has semantic meaning.
- **G-EOT, a generative model for creating adversarial 3D rendered objects for a black-box victim model**
- **Analysis of the experiment results of the G-EOT model:** Further work is needed to see if the issue of partially random gradients caused by rendering objects in random poses can be overcome.

1.5 Thesis Overview

The rest of the thesis is organised in the following way: chapter 2 provides the literature background, chapter 3 describes the research methodology and proposed architecture, chapter 4 contains the experiment design and results along with a discussion of said results, and chapter 5 summarises the contributions of this project and mentions future work to build upon it.

Chapter 2

Background Literature and Related Work

This chapter will cover the background literature that is necessary to understand the project, including an overview of deep neural networks and general information on adversarial examples. Following that, published works on physical adversarial examples and black-box attack methods will be discussed.

2.1 Background Literature

2.1.1 Generating adversarial examples

Generating adversarial perturbations in the white-box setting is an optimisation problem, and there is a wide variety of formulations of the problem in the literature. Szegedy *et al.* [12] first defined it in the targeted setting as finding the minimum perturbation δ such that the victim model f computes the desired wrong label y':

$$\begin{aligned} \min_{\delta} \quad & c\|\delta\|_2 \\ \text{s.t.} \quad & f(x+\delta) = y' \\ & x+\delta \in [0,1]^m, \text{ m is the number of pixels in image} \end{aligned} \tag{2.1}$$

Because equation 2.1 is a very hard optimisation problem, Szegedy *et al.* [12] use a relaxed form of the problem, where the constraint regarding the desired output is included in the objective function:

$$\begin{aligned} \min_{\delta} \quad & c\|\delta\|_2 + L(f(x+\delta), y') \\ \text{s.t.} \quad & x+\delta \in [0,1]^m, \text{ m is the number of pixels in image} \end{aligned} \tag{2.2}$$

In equation 2.2, $L(\cdot)$ represents the loss function of the victim model. We want to minimise the loss in regards to the desired wrong label to increase the chance that the victim model will produce that label when the adversarial input is given.

Another paper [17] defines the problem for untargeted adversarial attacks as:

$$\begin{aligned} \max_{\delta} \quad & L(f(x+\delta), y) \\ \text{s.t.} \quad & \|\delta\|_2 \leq \varepsilon \end{aligned} \tag{2.3}$$

Here, maximising the loss regarding the true label increases the chance that the model will not give that output. For targeted attacks, they define the problem as:

$$\begin{aligned} \max_{\delta} \quad & L(f(x+\delta), y) - L(f(x+\delta), y') \\ \text{s.t.} \quad & \|\delta\|_2 \leq \varepsilon \end{aligned} \tag{2.4}$$

In equation 2.4, the second term ensures that the loss function in regards to the desired wrong label is minimised.

In most papers on the subject, the optimisation problem constrains the size of the perturbation vector δ [16, 17, 23]. It does so by making sure that the perturbation vector's ℓ_2 or ℓ_∞ norm is smaller than the maximum threshold ε. This is to ensure that the perturbation is not too noticeable to the human eye. ε is often called **perturbation budget** in the literature. The ℓ_0 norm is used by some attacks which only want to change as few pixels as possible [16] .

The optimisation problem is usually solved through gradient descent. Szegedy *et al.* [12] used a box constrained L-BFGS optimiser. Goodfellow *et al.* [20] developed FGSM, a fast one-step attack method that is very popular in the literature. It calculates the gradient of the loss function of the target classifier at the given input and multiplies its sign with a constant scalar to obtain a perturbation that will change the output label. Basic Iterative Method is an iterative version of FGSM [24]. On the other hand, generative networks can also be used instead of gradient descent optimisation to create adversarial perturbations [3, 5, 22]. These methods will be presented in more detail in subsection 2.3.

You can see two examples of adversarial examples in figures 2.1 and 2.2. The example in figure 2.2 is classified incorrectly as a mask with a confidence value of 81.8%, while the normal image is classified correctly with a confidence value of 99.9%.

Figure 2.1: Example of an adversarial example made by L-BFGS [12]. The left image is the original image, which is classified correctly. You can see the adversarial noise in the centre, and on the right you can see the adversarial image, which is classified as an ostrich. Image taken from [12].

Figure 2.2: Example of an adversarial example created by the method in [4]. The left image is the original image of a jay, while the adversarial example on the right is misclassified as a mask. Images taken from [16].

2.1.2 Applicability of adversarial attacks

Ever since the publication of [12], most of the research has been on CNNs for object recognition tasks [16]. However, adversarial attacks are possible on other architectures, such as GANs [25], autoencoders [26], recurrent neural networks [27] and neural networks for semantic segmentation of an image [21]. Moreover, autoencoders are much more resilient against adversarial attacks [26], and the experiments done by Tang *et al.* [19] show that Vision Trans-

formers are more robust against adversarial attacks than CNNs.

2.1.3 Transferability of adversarial examples

Transferability-based black-box attacks succeed due to a property of adversarial examples, first discovered in [12], that they often manage to fool victim models that are different than the model for which the adversarial example was originally made. This can happen even if the two models have different architectures. Therefore, an attacker can create adversarial perturbations for a substitute model that they control and then use them to mount an attack against another model with inaccessible parameters and loss function gradient. Dong *et al.* [28] use an optimisation method with momentum to create transferability-based attacks. Meanwhile, ANGRI [22] is a generative model which learns to create adversarial perturbations by using 3-4 substitute models. Similarly, Zheng *et al.* [3] train their generator against a substitute model as well.

2.1.4 Taxonomy of adversarial attacks

Silva and Najafirad [17] categorises attacks based on several criteria. Depending on the **information** the attacker knows, an attack may be **white-box** or **black-box**. In the former, the attacker has full access to the architecture and parameters of the victim model, while in the latter the attacker does not have this knowledge. Dong *et al.* [18] further divides black-box attacks into three categories: transferability-based attacks, score-based attacks, where the attacker can query the victim model for soft-label predictions on chosen input, and then estimate the gradient, and decision-based attacks, where the attacker can only query for hard-label predictions, for which it is harder to estimate the gradient.

Attacks can also be classified depending on the **goals** of the attacker. **Untargeted** attacks only seek to make the victim model output a wrong result, whatever it may be. Meanwhile, **targeted** attacks have the goal of making the victim compute a specific wrong label. Furthermore, most attacks produce **image-specific** perturbations, which are designed to work for one specific image only. However, Moosavi-Dezfooli *et al.* [4] discovered a method of creating **universal** adversarial noise, which induces misclassification when applied to the vast majority of images in the dataset. Finally, attacks can be categorised based on the **attack frequency**, whether the adversarial noise is generated in one step, such as in FGSM [20], or in an iterative process, like in [29].

2.1.5 Why do adversarial examples exist?

There are varied and sometimes contradictory viewpoints in regards to why adversarial examples exist and there is no consensus. The hypothesis in Goodfellow *et al.* [20] appears to be the more popular explanation in the literature [16]. This hypothesis says adversarial attacks are caused by the fact that most neural networks are too linear, which is supported by the effectiveness of the FGSM [20] attack method. This excessive linearity could be caused by the popular ReLU activation function, which is piece-wise linear [20]. In Krotov and Hopfield [30], experiments show that neural networks with highly non-linear activation functions can not be fooled by adversarial examples generated by models equivalent to DNNs with ReLU activation, therefore supporting the linearity hypothesis.

On the other hand, Tanay and Griffin [31] contradict this and show that some linear image classifiers are not vulnerable to adversarial attacks. They hypothesize that adversarial examples exist because naturally occurring data samples exist on a subspace of the total input space and that adversarial examples exist when the classification boundary lies too close to this submanifold. Therefore, small perturbations manage to move the input across the decision boundary. On the other hand, their experiments show that if the decision boundary is perpendicular to this sub-manifold, the model is more robust against attacks. It is worth noting that their explanation does not totally contradict the linearity hypothesis, as the behaviour they describe is still quite linear.

On a different line of thought, Ilyas *et al.* [32] hypothesize that images contain subtle features that are useful for maximising the classification accuracy but are semantically meaningless in regards to the correct label. Adversarial perturbations that resemble these "non-robust" features make the victim model misclassify the adversarial examples with very high confidence.

2.1.6 Defences against adversarial attacks

Defences against adversarial attacks can be classified in the following categories [17]: gradient masking, used so that the attacker can not use the loss function gradient to create adversarial examples, adversarial example detection and robust optimisation. The latter includes adversarial training, where adversarial examples labelled with the correct label are added to the training set, certified defences, where the model is proven formally to be resistant against perturbations up to a certain limit, and regularisation. The authors of the literature survey in [17] put a lot of emphasis on certified defences and seem to disregard the effectiveness of adversarial training.

However, Dong *et al.* [18] performed extensive experiments that show that adversarial training often leads to more robust models than other defences based on regularisation or certified defence.

2.2 Adversarial attacks in the physical world

Most attacks, including the one in figure 1.1 on page 3, directly add the perturbations to the whole image. In a realistic setting, the attacker can not directly manipulate the pixels of the input, especially when the model takes its input from sensors. Furthermore, physical adversarial examples, whether 2D printed images or 3D objects, can have their fooling rate diminished by the angle they are viewed from, their distance from the camera, lighting conditions, the inability of the sensor to pick up the subtle adversarial perturbations, and perhaps colour printing errors [2, 13, 24].

Kurakin *et al.* [24] were among the first who investigated adversarial attacks in the physical space. They created adversarial images using FGSM [20], printed them and took a picture of the printed photos using a smartphone. They then used automatic perspective transformation and cropping to transform the picture, then ran the classifier on it and compared the attack success with the original adversarial example. Over 66.67% of adversarial examples still fooled the victim model after being printed and photographed. Therefore, it was proven that adversarial examples apply to the physical domain.

However, the photographs taken in [24] only had slight variations in terms of the camera lighting, camera distance and camera angle. Meanwhile, Lu *et al.* [15] did similar experiments to the ones in [24], except they tried a wide variety of camera angles and distances, and they found that the adversarial examples were successful only when the camera was in certain positions. For example, the accuracy of the model doubled or almost tripled when the camera distance increased from 0.5m to 1.5m. Therefore, they concluded that adversarial examples may not be a serious threat to neural networks used in cyber-physical systems.

2.2.1 Expectation over Transformation

To rectify the shortcomings of adversarial attacks mentioned above, Athalye *et al.* [2] proposed the Expectation over Transformation (EOT) framework. It is designed to create adversarial examples that are effective in physical settings, and can work on 2D images, 3D rendered objects and 3D printed objects. Their key innovation is that they model transformations such

as rotation, translation, perspective projection, 3D rendering or lighting in the optimisation search for the adversarial noise.

2.2.1.1 Technique

The authors optimise the adversarial texture x' in order to maximise the following objective function:

$$\begin{aligned} \max_{x'} \quad & \mathrm{E}_{t\sim T}[logP(y_t|t(x'))] - \lambda \mathrm{E}_{t\sim T}[d(t(x'),t(x))] \\ \text{s.t.} \quad & x \in [0,1]^d \end{aligned} \tag{2.5}$$

where $t(\cdot)$ is a function which simulates the various transformations, T is a distribution of transformation functions, λ is a chosen penalty constant, and $d(\cdot,\cdot)$ measures the perceptual difference between x and x'.

While x' is the input that the attacker directly controls, $t(x)$ is the input seen by the neural network. In the 3D case, x is a 2D texture and $t(x)$ is a 2D image of a 3D rendered object with that texture, seen from a random angle and distance. You can see an example of this in figure 2.3.

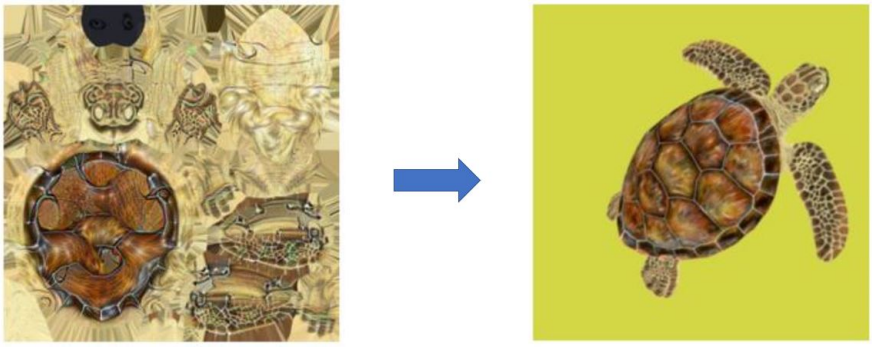

Figure 2.3: The transformation function $t(\cdot)$ can model the rendering of a 2D texture into an image of the 3D object with a specific pose. Images are taken from [2].

The 3D rendering process is formally considered as a linear transformation such that $t(x) = Mx + b$. This is crucial, because EOT uses gradient descent optimisation, and therefore needs equation 2.5 to be differentiable, including the transformation function $t(\cdot)$. The authors of [2] claim that the coordinate map M and background b can be calculated for a given object pose by computing the texture-space coordinate for each corresponding on-screen pixel. M and b must be calculated for each new pose of the 3D model. The authors claim that they modified

an existing renderer to return this information. However, they do not provide further details on how they compute M and b, what they are exactly, nor do they mention which renderer they used. Furthermore, they do not give a link to their source code.

The first term of equation 2.5 maximises the log probability that the neural network will classify the adversarial example as the desired target label y_t. Meanwhile, the second term minimises the distance between the adversarial example and the original input. The authors use the distance function $d(\cdot,\cdot)$ instead of $x' - x$ to focus on the actual perceptual distance between the two, as the classifier sees $t(x)$ rather than x.

The framework leaves the choice of the distribution of transformation functions T and the distance function $d(\cdot,\cdot)$ up to the user. According to the supplementary material of Athalye *et al.* [2], the parameters for the camera distance, translation on the x and y axes, rotation of the object, lighting, and 3D printer colour error are each sampled from independent truncated uniform distributions. Camera noise is simulated by using noise drawn from a gaussian distribution.

Meanwhile, the distance metric that they used is the Euclidean distance between the projections in LAB space of $t(x)$ and $t(x')$. LAB [33] is a colour space where the Euclidean distance between two colour vectors is roughly equivalent to how different the human eye perceives those colours. The authors chose this metric so that the adversarial noise is less obvious to humans.

In each optimisation step, the authors use the mean loss function value over a mini-batch of 40 images, each being a different transformation of the same original image. 32 of those are re-used from the previous mini-batch, while the other 8 are new images created with 8 new transformation functions drawn from the distribution T. The authors reuse renders as it is computationally expensive to create a new one. By using this process, they ensure that the adversarial object remains adversarial even though it is viewed from a wide variety of perspectives.

2.2.1.2 Results and discussion

To evaluate the EOT framework for generating 3D adversarial objects, the authors attack an InceptionV3 classifier [34] with 78% accuracy on the Imagenet dataset. For the 3D rendered objects scenario, they used a dataset of 10 3D models, each representing a different class of the Imaganet dataset. For each of these models, they randomly chose 20 random target labels and then created an adversarial texture for that model and target label.

When evaluating the 200 adversarial examples, they sample 100 random transformations from the distribution T for each example and render 100 images using the adversarial texture and 100 using the original texture. The images with the normal texture had a classification accuracy of 68.8% and were classified as the adversarial target label 1.1% of the time. Meanwhile, the adversarial images were classified as the target label 83.4% of the time, while only 0.01% of these images were correctly classified.

Following that, Athalye *et al.* created two 3D-printed adversarial objects, a baseball and a turtle, and found that they were classified as the target label 59% and 82% of the time, respectively. You can see some photos of the turtle in figure 2.4. Consequently, we can conclude that the EOT framework can synthesize physical 3D adversarial objects that remain highly effective when viewed from a variety of viewpoints.

Athalye *et al.* [2] provides a good high-level description of their proposed framework, and their experiment design is sound. Furthermore, their discussions offer insight into the limitations of their method. However, it lacks some critical information needed to re-create their work. As mentioned in subsection 2.2.1.1, they do not explain exactly how to get the necessary parameters to represent $t(\cdot)$ as a linear transformation, nor do they provide a link to their source code or even the data set they used. Furthermore, they do not mention what optimiser learning rate or penalty constant λ they used or for how many steps they ran their algorithm. Finally, a step-by-step algorithm would have been useful for implementing EOT and re-creating their results.

2.2.2 Robust Physical Perturbations

Concurrently to Athalye *et al.* [2], Eykholt *et al.* [13] developed an attack for 2D physical objects, such as traffic signs, called Robust Physical Perturbations (RP_2).

2.2.2.1 Technique

In contrast with EOT [2], where the transformation functions from T synthetically augmented the dataset by rendering the 3D object in various poses, here the authors augment it by manually taking photos of the original object from various angles and distances and in varying lighting conditions. They further use some synthetic transformations in the form of cropping and changing the image brightness to add new transformed images to the dataset. The input to RP_2 is a photo of an object taken from its front, and during the optimisation process, it

Figure 2.4: A selection of photographs of the 3D adversarial turtle object. The adversarial perturbation was made for the "rifle" target label. Taken from [2].

draws a new sample from a distribution X of images of that same object, seen under various transformations.

Moreover, because the input to RP_2 is an image of the physical 3D object rather than a 2D texture of an object, the authors use a mask M_x to make sure that the adversarial perturbation δ is only applied to object itself and not to the image background. The mask is a matrix the size of the image, with a 0 for each pixel where no perturbation should be added, and 1 otherwise.

With these changes in mind, the authors of [13] define the optimisation problem in RP_2 as:

$$\min_{\delta} \quad \lambda \|M_x \cdot \delta\|_p + NPS + \mathrm{E}_{x_i \sim X} J(f_\theta(x_i + T_i(M_x \cdot \delta)), y_t) \tag{2.6}$$

where NPS is a term for correcting printing colour errors, $J(\cdot,\cdot)$ is the loss function of the classifier represented by f_θ, y_t is the adversarial target label and T_i is a function that transforms

the perturbation to match the object in the image. If the object is seen from a 30 angle, the perturbation is rotated by 30 in 3D. The first term is for minimising the size of the perturbation, and the third term ensures that the adversarial example is misclassified.

2.2.2.2 Results and discussion

Eykholt *et al.* used RP_2 to create a poster of a road STOP sign, with perturbations applied to the whole surface of the sign. They manually took photos of the poster from a variety of distances and angles and fed them into a CNN trained on the LISA traffic sign dataset. 100% of those photos were misclassified as the target label by the network, with an average confidence of 80.51%. In another experiment, they created adversarial patches in the form of stickers and placed them over a STOP sign. Just like in the previous experiment, they took photos of the sign and then fed them into the LISA CNN. The attack with regular stickers had a 100% success rate, and an attack with a sticker which looked like graffiti succeeded 66.67% of the time. Finally, they put adversarial stickers over a real STOP sign on a street and took photos of it from a moving car. The attack success rate was 92.4%, once more proving the effectiveness of their algorithm.

RP_2 is an interesting attack method which is capable of producing robust adversarial examples for road signs, thus endangering traffic safety. However, it is quite inconvenient. It requires the user to manually take a lot of photos and manually create the perturbation mask M_x. Furthermore, the authors of [13] do not offer any detailed information on how the perturbation alignment function T_i works, not even in the supplementary material. Finally, their approach is only applicable to flat physical surfaces, like road signs. On the other hand, EOT works for any 3D object [2], and could also use a mask to create adversarial patches like [13] did.

2.3 Generative Models for black-box attacks

2.3.1 Generator-simulator model

Zheng *et al.* [3] proposed a generative model for creating targeted black-box adversarial examples for 2D images. Its architecture consists of two components, a generator and a simulator. The purpose of the simulator is to act as a substitute for the black-box victim model, while the generator creates adversarial noise for a given image and target label.

2.3.1.1 Technique

The simulator is trained so that it provides the same output as the victim model for a given adversarial example created by the generator:

$$\begin{aligned} \min_{\theta_s} \quad & L(S(\theta_s, x+G(x,z)), V(\theta_v, x+G(x,z))) \\ \text{s.t.} \quad & x \in X, \|G(x,z)\|_\infty < \delta \end{aligned} \tag{2.7}$$

where the outputs predicted by the simulator and victim model are denoted by $S(\cdot,\cdot)$ and $V(\cdot,\cdot)$, respectively. θ_s and θ_v are the parameters of the simulator and victim model, respectively. The adversarial noise created by the generator for a given image x and a given target label z is written as $G(x,z)$. X is the training set and δ is the perturbation budget. $L(\cdot,\cdot)$ represents the cross-entropy loss function. Equation 2.7 is visualised in figure 2.5. It is essentially a simplified version of distillation [35].

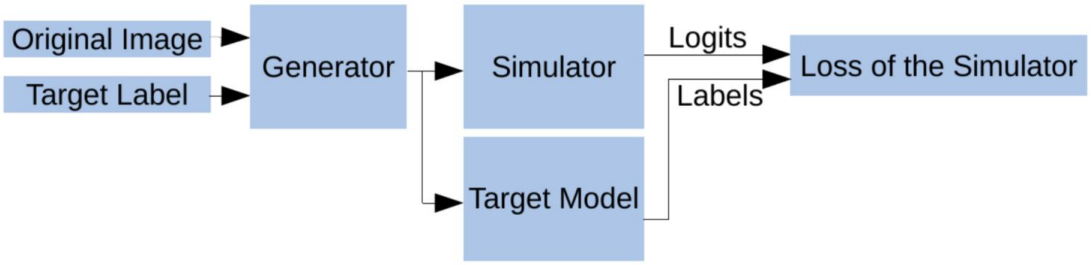

Figure 2.5: Computational flow of the loss function for training the simulator. Diagram taken from [3].

Meanwhile, the generator is trained to fool the simulator such that the latter will classify the generated adversarial image with the desired wrong label z. Training the generator takes the following form:

$$\begin{aligned} \min_{\theta_g} \quad & L(S(\theta_s, x+G(x,z)), z) + \beta \|G(x,z)\|_2 \\ \text{s.t.} \quad & x \in X, z \in Y \setminus \{y\} \end{aligned} \tag{2.8}$$

where θ_g is the parameter vector of the generator, Y is the set of labels and y is the correct label of x. β is a constant for the perturbation size penalty. The first term of equation 2.8 maximises the probability that the simulator will classify the generator output as the target label, while the second term constrains the ℓ_2 norm of the noise image vector. Figure 2.6 visualises the generator training process.

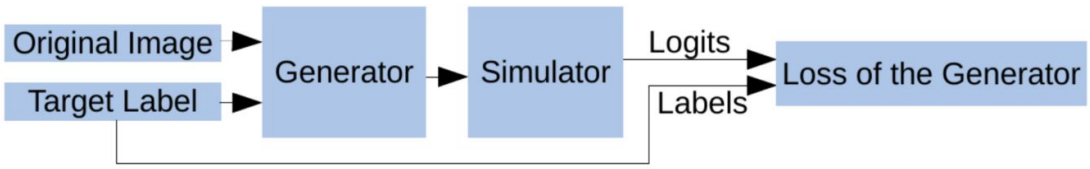

Figure 2.6: Computational flow of the loss function for training the generator. Diagram taken from [3].

The simulator in the paper is a CNN. The proposed generator is an auto-encoder, as you can see in figure 2.7. The encoder takes the form of a CNN without fully connected layers or global average pooling, and its job is to encode an image as a vector in higher-dimensional latent space.

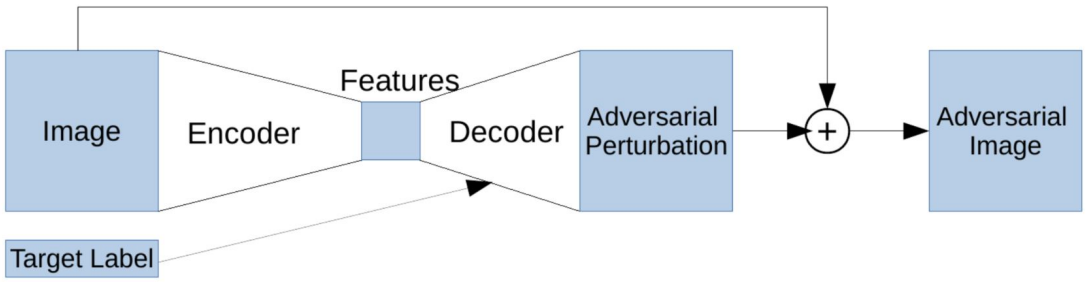

Figure 2.7: The autoencoder architecture of the generator in [3]. Diagram taken from [3].

The decoder is an ensemble of neural networks made out of transposed convolution layers, as you can see in figure 2.8. It uses an ensemble because the perturbation must be designed for a given target label. The decoder has a gating sub-network inspired by Shazeer *et al.* [36] which learns to create a sparse weights vector based on the target label. This sub-network is just one fully connected layer with as many output neurons as there are experts. The output of the decoder is a weighted sum of the outputs of each expert model, using the weights from the gating network. Different combinations of expert models are used for different target labels, and each expert learns to focus on one or a couple of target labels.

The authors train the simulator and generator in alternative steps. Moreover, the authors used a number of techniques to improve results and speed-up training. Before the main training loop, they warm-up the simulator by training it on normal images. During the main loop, they also train the simulator on images with random noise, and not just with adversarial noise. Furthermore, they clip the gradients of the generator to values between -1 and 1.

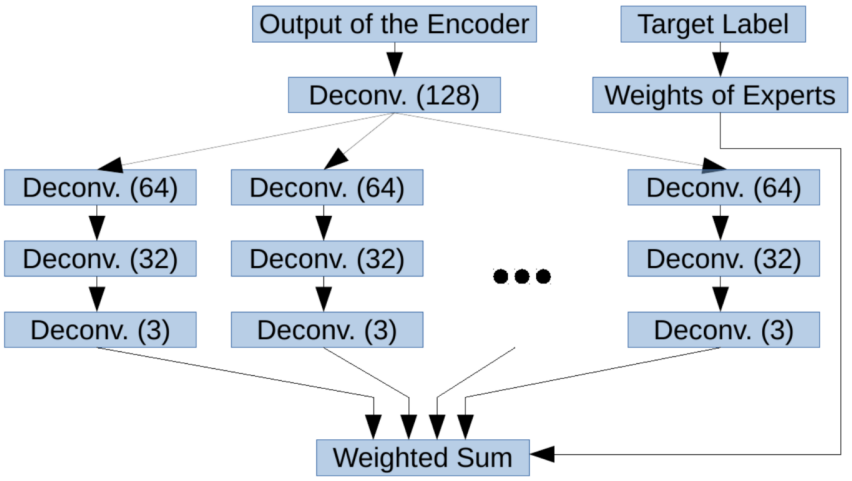

Figure 2.8: The decoder architecture of the generator in [3]. Diagram taken from [3].

2.3.1.2 Results and discussion

Zheng *et al.* [3] evaluate their model on the CIFAR-10 dataset and experiment with 4 different CNNs as simulators. These include 3 small novel CNNs, SmallNet, SimpleNet and ConcatNet, with architectures inspired by ResNet [37], Xception [38] and DenseNet [39] respectively. The fourth is the Xception model [38] itself. They try all 16 possible combinations of the 4 CNNs as both simulator and victim model, with an average success rate of over 80% across the 16 experiments and with a minimum of 63% and a maximum of 87.3%. Considering that all 4 CNNs had an accuracy of over 90% on normal images, this demonstrates that the proposed technique can effectively create black-box attacks on CIFAR-10 CNNs. Moreover, it outperforms the attack success rate of the earlier generative model made by Sarkar *et al.* [22]. The authors repeated their experiments on CIFAR-100, where they had an attack success rate of 72.1%.

The paper demonstrates an effective method to create a generator for black-box adversarial attacks on small images and provides plenty of details to enable someone to re-create their work. The diagrams of the model architecture are very clear, and the authors provide values for most hyper-parameters. However, they do not mention how long they trained their model, nor what learning rate, optimiser and batch size they used, and the paper does not have supplementary material. They do not provide a link to their source code either.

2.3.2 AdvGAN

Concurrently with Zheng *et al.* [3], Xiao *et al.* [5] created a generative adversarial network (GAN) [40] to create adversarial images in the black-box setting. It is similar to the former, and therefore this subsection will concentrate on the differences between the two.

2.3.2.1 Technique

AdvGAN has a generator network which produces adversarial noise and a simulator which is trained to behave like the black-box victim model, just like Zheng *et al.* does. However, it also uses a discriminator, as you can see in figure 2.9, where the simulator is called a "distilled black-box".

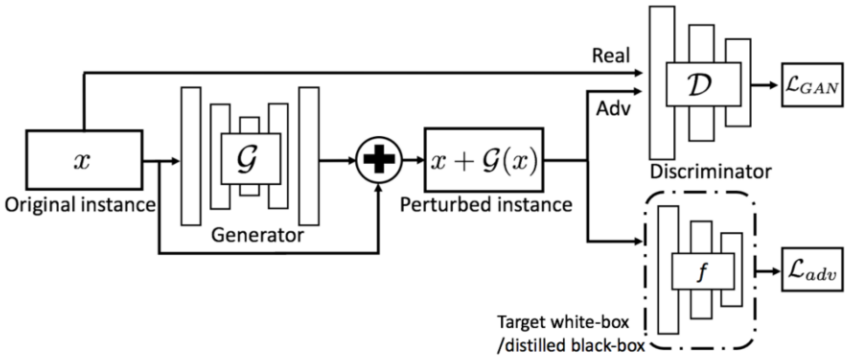

Figure 2.9: High-level overview of AdvGAN. Diagram taken from [5].

Unlike in [3], the generator trains to fool the discriminator into thinking adversarial images are normal images, while also training to fool the simulator into classifying the adversarial images as the desired target label. Meanwhile, the discriminator trains to tell if an image contains noise made by the generator. The generator training against the discriminator ensures that the adversarial noise is hard to distinguish. The noise is further constrained by a hinge loss:

$$\mathrm{L}_{\text{hinge}} = \mathrm{E}_x max(0, \|G(x)\|_2 - c) \tag{2.9}$$

which adds a penalty if the Euclidean length of the noise vector is under c, while Zheng *et al.* simply used an ℓ_2 penalty term.

The simulator is trained to behave like the victim model using distillation [35]. After every step of training the generator and discriminator, the simulator performs a training step with

both normal and adversarial images as input. On the other hand, Zheng *et al.* [3] only do so with adversarial images and trains the simulator on normal images only during the initial warm-up phase.

The discriminator of AdvGAN is a CNN which performs binary classification on patches of the input image and then averages the patch predictions to determine if the whole image is adversarial. The generator is an auto-encoder where the encoder is made up of convolutional layers, while the decoder is made of transposed convolutions. Unlike in the generator of Zheng *et al.*, AdvGAN has pairs of encoder and decoder layers with residual connections between them, as U-Net does [41]. Finally, the generator of AdvGAN does not make use of a mixture-of-experts in its decoder, and therefore it can not generate perturbations for all target labels and must be re-trained for each target label. This is in contrast with the generator in subsection 2.3.1, which could generate adversarial examples for any target label.

2.3.2.2 Results and discussion

The authors evaluate AdvGAN against ResNet-32 and Wide ResNet-34 [37] models trained on CIFAR-10, and in the black-box setting the targeted attack has a success rate of 78.5% and 81.8%, respectively. Moreover, it achieved an attack success rate of 92.76% in the black-box setting of the MadryLab challenge [42]. Finally, the authors claim that they managed to create adversarial noise for 100 high-resolution ImageNet-like images against an InceptionV3 model [34] with a 100% attack success rate.

Xiao *et al.* created a powerful generative network for targeted black-box attacks, and the paper provides a lot of detail on the overview of the model and on how it is trained. Moreover, it performs a wide range of experiments, with both white-box and black-box settings, as well as attacking models with defences against adversarial attacks. However, they do not provide detailed information on the exact architecture they used for the discriminator and generator, they just say "adopt similar architectures for generator and discriminator with image-to-image translation literature" then cite two papers. They do not provide source code or supplementary material either. However, the largest downside of their model is that it must be trained for each individual target label.

Furthermore, their claim of a 100% attack success rate on high-resolution images is odd, as they do not mention how they adapted their architecture for larger input, and it makes no sense how the attack was even more successful than against simpler models on CIFAR-10. The example Imagenet adversarial images in the paper look identical to the normal images, too.

Chapter 3

Methodology

In this chapter, I will be discussing the research methodology of the project, and then explain the implementation of the proposed model and the metrics which will be used in experiments.

As you will see later in section 3.4, the proposed model is a combination of the methods of [2] and [3], and therefore it is necessary to implement those papers before merging the two for my model. Therefore, these implementations will be covered in sections 3.2 and 3.3.

3.1 Method overview

The EOT framework presented in subsection 2.2.1 is highly effective at creating robust adversarial noise for 3D-rendered objects. This is despite the fact it is training on renders of the object with a random position and angle. In EOT, the transformation from the texture to the rendered image is represented as a simple linear transformation. At the same time, generative models have proven to be capable of learning to generate large 256x256 images [43], especially benefiting from having significantly more parameters. Therefore, my intuition is that with the proper training, hyper-parameters and architecture, a generative model should be able to learn to create adversarial textures. Since the transformation representing 3D rendering is linear, it should not be an insurmountable obstacle for training the generator.

Moreover, the black-box generative model in subsection 2.3.1 can be easily combined with the EOT framework for making adversarial objects in subsection 2.2.1, by placing a differentiable rendering pipeline between the generator and the simulator.

The research hypothesis of this project is the following:

It is possible to use a generative neural network to create 3D-rendered adversarial objects in the targeted black-box setting that have a targeted fooling rate of at least 50% on a victim CNN classifier for object recognition.

The targeted fooling rate metric is explained later in section 3.5.

This is a quantitative experimental research project. The model proposed in section 3.4 creates 3D-rendered objects, and pictures of those objects will be given to various neural network classifiers. The predicted label will be compared against the correct label and the adversarial target label to evaluate the effectiveness of the proposed method. We will mainly be using a circulatory research process. Further details on the experiments are found in subsection 4.4.1.

3.2 Implementation of EOT

Like subsection 2.2.1 mentioned, the authors of [2] fail to include detailed information on how they represented 3D rendering as a differentiable linear transformation. One of the authors provides a step-by-step guide with source code [44], but only for attacks for 2D images, not for textures which are then rendered as 3D objects. Therefore, I searched Github.com for repositories related to [2], and found one implementation of the paper for the 3D rendered objects case [45].

However, the implementation lacked documentation and was written in Tensorflow 1, which is obsolete, less intuitive and harder to debug. I forked the repository and almost completely re-wrote the code to run with Tensorflow 2 [1], documented it and refactored it according to software engineering principles, although the algorithm is broadly the same as [45] implemented it.

3.2.1 UV mapping

3D models are made up of a series of vertices, points in 3D space which define the shape of an object. One method to apply a texture to a model is to use UV coordinates, which are pairs of coordinates which tell you which pixel from the texture image should be used to colour a certain vertex. These coordinates are defined when the 3D model is created and come included with the vertex coordinates in the 3D model file. After rendering a 3D model, an OpenGL renderer will look up the UV coordinates for each vertex, and for each triangle it will copy the

[1] Source code at https://github.com/Alexandru-Dascalu/adversarial-3d

fragment of the texture that is within the three texture coordinates of the triangle, thus creating a textured object. A UV map is what Athalye *et al.* [2] likely referred to when talking about texture-space coordinates.

A normal renderer will take the array of vertices, the UV map and the object's texture as inputs. It will first render the object by translating and rotating the position vector of each vertex using matrix multiplication [46]. Following that, it will use the UV coordinates of each vertex to sample a colour from the texture, either by using nearest-neighbour, bilinear interpolation or anisotropic filtering [47]. However, the renderer implemented by [45] and used in this project simply returns the UV coordinates instead of a colour. Therefore, the output of the renderer is a 299x299 image of the object rendered in the desired pose, but instead of an RGB colour, each pixel has a U and V coordinate pointing to a pixel in the texture. This output image will be called a "UV map" from now on. The rotation angle, X/Y position and distance from the camera are sampled from truncated uniform distributions, and used by the renderer to create a UV map for a certain pose of the 3D model.

The renderer is implemented using ModernGL [2], a Python wrapper around OpenGL which allows you to render .obj files with textures.

3.2.2 Transformation from texture to image

Algorithm 1 Pseudo-code representation of the transformation function $t(\cdot)$

$std\_textures \leftarrow repeat(std\_texture)$
$adv\_textures \leftarrow repeat(adv\_texture)$
if print_error **then**
    $std\_textures, adv\_textures \leftarrow apply\_print\_error(std\_textures, adv\_textures)$
end if
$std\_image \leftarrow resample(std\_textures, uv\_maps)$
$adv\_images \leftarrow resample(adv\_textures, uv\_maps)$
$std\_images, adv\_images \leftarrow add\_background(std\_textures, adv\_textures)$
if photo_error **then**
    $std\_images, adv\_images \leftarrow apply\_photo\_error(std\_images, adv\_images)$
end if
$std\_images, adv\_images \leftarrow normalise(std\_images, adv\_images)$
return std_images, adv_images

The renderer produces a UV map for each image in the mini-batch, which together with the texture is used as input for the algorithm seen in algorithm 1. This algorithm is a step-by-step

[2] https://github.com/moderngl/moderngl

representation of the $t(x) = Mx + b$ function mentioned in section 2.2.1.1. As you will see, this algorithm is indeed just a linear transformation. The reason that this algorithm creates images with both the original and adversarial paper is that we need to calculate the difference between them for the penalty term in equation 2.5 on page 12. Since this penalty seeks to measure the perceptual difference caused by the adversarial noise, all other parameters such as the pose of the object or the background colour must be identical for each pair made up of a normal and an adversarial image.

The $repeat(\cdot)$ function in lines 1 and 2 of the pseudocode simply replicates the 3D texture tensor as many times as the size of the batch, as you can see in listing 3.1.

```
1 import tensorflow as tf
2 import config
3
4 def repeat(x):
5     """
6     Args: x: A 3-D tensor with shape [height, width, 3]
7     Returns: A 4-D tensor with shape [batch_size, height, size, 3]
8     """
9     return tf.tile(tf.expand_dims(x, axis=0), [config.batch_size, 1, 1, 1])
```

Listing 3.1: An implementation of the repeat function from algorithm 1

```
1  import tensorflow as tf
2  import config
3
4  def apply_print_error(std_textures, adv_textures):
5      multiplier = tf.random.uniform(
6          [config.batch_size, 1, 1, 3],
7          config.channel_mult_min,
8          config.channel_mult_max
9      )
10     addend = tf.random.uniform(
11         [config.batch_size, 1, 1, 3],
12         config.channel_add_min,
13         config.channel_add_max
14     )
15     std_textures = std_textures * multiplier + addend
16     adv_textures = adv_textures * multiplier + addend
17
18     return std_textures, adv_textures
```

Listing 3.2: Simulating 3D printer errors.

Athalye *et al.* [2] modelled per-colour channel 3D printer errors in their experiments with 3D-printed adversarial objects. The *apply_print_error* function implements this by randomly generating a scalar multiplier and addend for each colour channel in each texture in the batch, and using those to linearly transform the texture, as you can see in listing 3.2.

After applying the print error, we can create a batch of 299x299 images of the rendered object. The $resample(\cdot,\cdot)$ function from algorithm 1 is implemented as the $resampler(\cdot,\cdot)$ method from the tensorflow_addons library [48]. For each pixel in each image, it samples a colour from that image's texture based on the UV coordinates of the corresponding pixel in that image's UV map. tensorflow_addons.resampler uses bilinear interpolation to mix the colours of neighbouring texture pixels to get the colour it will return [48]. Therefore, it is a linear transformation.

```
1  import tensorflow as tf
2  import config
3
4  def add_background(std_images, adv_images, uv_maps):
5      # compute a mask with True values for each pixel which represents the object,
           and False for background pixels.
6      mask = tf.reduce_all(tf.not_equal(uv_maps, 0.0), axis=3, keepdims=True)
7      # generate random background colour for each image in batch
8      color = tf.random.uniform([config.batch_size, 1, 1, 3], config.background_min,
            config.background_max)
9
10     new_std_images = set_background(std_images, mask, color)
11     new_adv_images = set_background(adv_images, mask, color)
12     return std_images, adv_images
13
14 def set_background(images, mask, colours):
15     """
16     images: A 4-D tensor with shape [batch_size, height, size, 3].
17     mask: boolean mask with shape [batch_size, height, width, 1]
18     colours: tensor with shape [batch_size, 1, 1, 3].
19     """
20     # repeat mask for each colour channel
21     mask = tf.tile(mask, [1, 1, 1, 3])
22     inverse_mask = tf.logical_not(mask)
23
24     background = tf.cast(inverse_mask, tf.float32) * colours
25     object_cutout = tf.cast(mask, tf.float32) * image
26     return object_cutout + background
```

Listing 3.3: Code for adding colour background to the renders of the 3D object.

In [2], the authors used random colours for the background of each render. Therefore, for each image, a mask with true boolean values for each pixel that makes up the object is computed. The UV maps have a value of 0 for each pixel that makes up the background of the scene rather than the object, so I just check for UV coordinates different from 0. An RGB colour is then sampled from a truncated uniform distribution. I obtain the background by performing element-wise multiplication between the colour vector and a logical inverse of the mask, which has true values for the background and false for the object. I then add that with a multiplication between each image with its mask, which essentially "cuts out" the rendered object and adds it on top of the coloured background. This process can be seen in code listing 3.3.

Since a 3D-printed adversarial object would be photographed with a camera connected to a neural network, Athalye *et al.* [2] also modelled random lighting conditions and camera noise in their experiments with physical objects. The implementation, represented by the $apply\_photo\_error(\cdot,\cdot)$ function in algorithm 1, linearly scales each image with a random scalar multiplier and addend to lighten or darken the image. It then simulates camera noise by adding Gaussian noise.

```
1  import tensorflow as tf
2
3  def normalize(std_images, adv_images):
4      std_images_minimums = tf.reduce_min(std_images, axis=[1, 2, 3], keepdims=True)
5      adv_images_minimums = tf.reduce_min(adv_images, axis=[1, 2, 3], keepdims=True)
6
7      std_images_maximums = tf.reduce_max(std_images, axis=[1, 2, 3], keepdims=True)
8      adv_images_maximums = tf.reduce_max(adv_images, axis=[1, 2, 3], keepdims=True)
9
10     minimum = tf.minimum(std_images_minimums, adv_images_minimums)
11     maximum = tf.maximum(std_images_maximums, adv_images_maximums)
12
13     minimum = tf.minimum(minimum, 0)
14     maximum = tf.maximum(maximum, 1)
15
16     return (std_images - minimum) / (maximum - minimum), (adv_images - minimum) /
           (maximum - minimum)
```

Listing 3.4: Source code for normalising the rendered images

Finally, the $normalise(\cdot,\cdot)$ function from algorithm 1 linearly normalises the images, as the random scaling done when applying the photo or print error may result in values outside $[0,1]$. Firstly, the minimum and maximum pixel values across all colour channels for each

image are calculated. I will then pick the minimum and maximum values for each pair of normal and adversarial images. This is important, as the adversarial noise may fall outside $[0, 1]$, and because it is semantically meaningful, the normal image must use the same scale as its adversarial counterpart. Finally, if a minimum or maximum value is inside $[0, 1]$, we set it to 0 or 1, respectively. This is because scaling is done only to bring invalid values inside the valid range, so if values are already inside that range, they do not need to be scaled. This normalisation process can be seen in code listing 3.4.

3.2.3 Creating the adversarial texture

The rest of the implementation works just how subsection 2.2.1.1 described the EOT framework [2]. Rendered images of the 3D model are created using the algorithm described in subsection 3.2.2. As seen in the code listings in subsection 3.2.2, these renders are dependent on parameters drawn from uniform distributions for the rotation, camera distance, X/Y translation, print error, lighting, camera noise and background colour.

The rendered images are given to the victim model for inference. The resulting logits, together with the target labels, are used to calculate the average cross-entropy loss across all images in the mini-batch. This is the first term of the objective function in equation 2.5 on page 12. Furthermore, the second term of that equation is calculated by projecting the normal and adversarial images into LAB space [33] using the scikit-image library [3], then calculating the average ℓ_2 norm of the difference between each pair of normal and adversarial images. An Adam optimiser is used to update the adversarial texture such that the objective function in equation 2.5 is minimised.

At each optimisation step, a certain percentage of rendered images from the previous mini-batch are re-used, as Athalye *et al.* also did. This is implemented by shifting the previous tensor of images to the left, thus discarding some of the images from the previous mini-batch, and then adding the new renders at the end. Therefore, old renders are gradually replaced. For example, with a re-use percentage of 80%, there is an entirely new batch of renders after every 5 optimisation steps. For added efficiency, the logits of the victim model for old renders are memorised rather than re-calculated at each stop, logits are computed only for the new renders.

[3] https://scikit-image.org/docs/dev/api/skimage.color.html#skimage.color.rgb2lab

3.3 Implementation of Zheng et al.'s model

I searched Github.com for an existing implementation of Zheng *et al.* [3] and found a Tensorflow implementation made by the authors of the paper themselves [4]. Unfortunately, it was also written in Tensorflow 1, lacked documentation and had a lot of duplicate or unused code. On the other hand, it implemented all models and training techniques described by Zheng *et al.* [3].

Due to time constraints, I chose not to completely rewrite this implementation in Tensorflow 2. I forked the repository [5] and only added some comments, some minor refactoring to improve code readability and some code to plot the training history.

3.4 Proposed model for G-EOT

3.4.1 Overview

The proposed G-EOT model is at a high level the same generative model presented in subsection 2.3.1, except that it has elements of the EOT framework [2] added to it. The key difference is that the generator produces adversarial perturbations for a 2D texture rather than an image of an object, as you can see in figure 3.1. The perturbations are then added to the original texture to create the adversarial texture. The latter is then rendered as a 3D object, and an image of the rendered object is then given to the simulator and to the victim model.

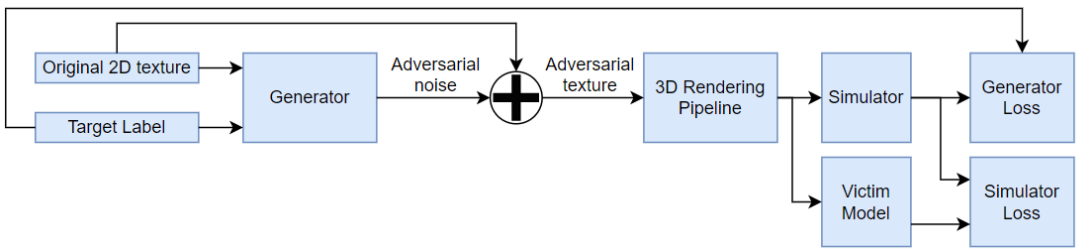

Figure 3.1: The architecture of G-EOT.

The 3D rendering pipeline in figure 3.1 is essentially identical to the one presented in section 3.2. It simulates 3D rotation, translation, varying camera distance, different background colours and lighting conditions, camera noise, and 3D printer error. The parameters for 3D

[4] https://github.com/StanleyZheng-FDU/targeted-black-box-attack
[5] https://github.com/Alexandru-Dascalu/targeted-black-box-attack

rendering are sampled from truncated uniform distributions, similarly to the distributions presented in the supplementary material of [2].

All textures for 3D models that I could find had at least 1024x1024 pixels, with most having a resolution of 2048x2048 pixels. Since generative models struggle to generate output larger than 256x256 pixels [43], the generator first performs average pooling on the texture to reduce its resolution. The adversarial noise created by the generator has a resolution of 256x256. This is then upsampled using bicubic interpolation to 2048x2048 pixels so that it can be added to the original texture to obtain the adversarial texture.

EOT [2] was chosen rather than RP_2 [13] because it is a lot more convenient, for reasons named in subsection 2.2.2.2. Furthermore, the synthetic transformation functions for 3D rendered objects can accurately simulate real-world transformations [2]. On the other hand, the method presented in Zheng *et al.* [3] was chosen for the generator-simulator model rather than the one from Xiao *et al.* [5] because it is simpler and it can generate attacks for all target labels, as mentioned in subsection 2.3.2.

The implementation[6] is written using the Tensorflow 2 library.

3.4.2 Generator

The generator is an auto-encoder with a similar structure as the one in figure 2.7 in subsection 2.3.1. Its encoder is a CNN with an architecture inspired by the SimpleNet CNN from Zheng *et al.* [3], which is itself inspired by Xception [38]. Its structure can be seen in figure 3.2. The numbers after the type of the convolutional layer are the number of output channels of that layer, followed by the kernel size.

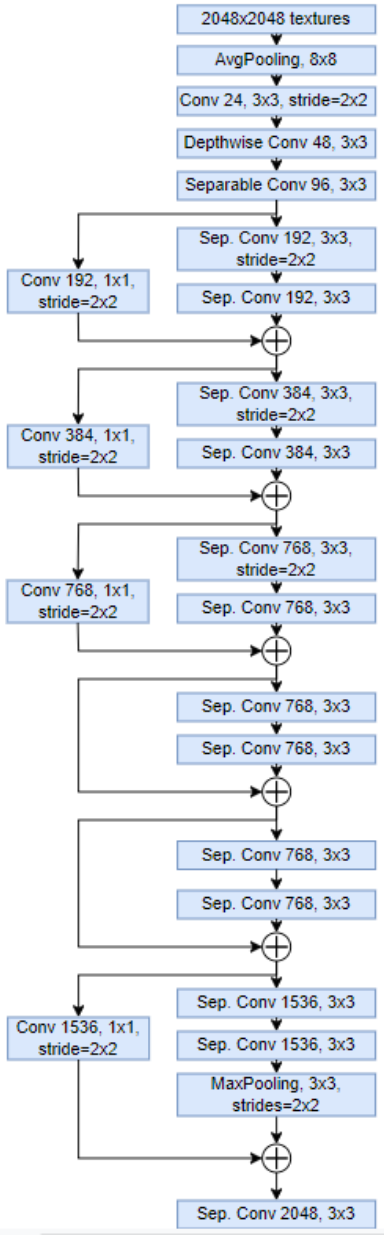

Figure 3.2: The structure of the encoder in the G-EOT generator.

[6] https://github.com/Alexandru-Dascalu/G-EOT

Although not seen in the diagram, each convolutional layer is followed by a batch normalisation layer [49].

Meanwhile, the structure of the decoder is similar to the one in Zheng *et al.* [3], but it is deeper, has more kernels in its first layer, and has an upsampling layer at the end. Figure 3.3 has a diagram of the structure of the decoder. A more detailed diagram of the generator is in appendix A.

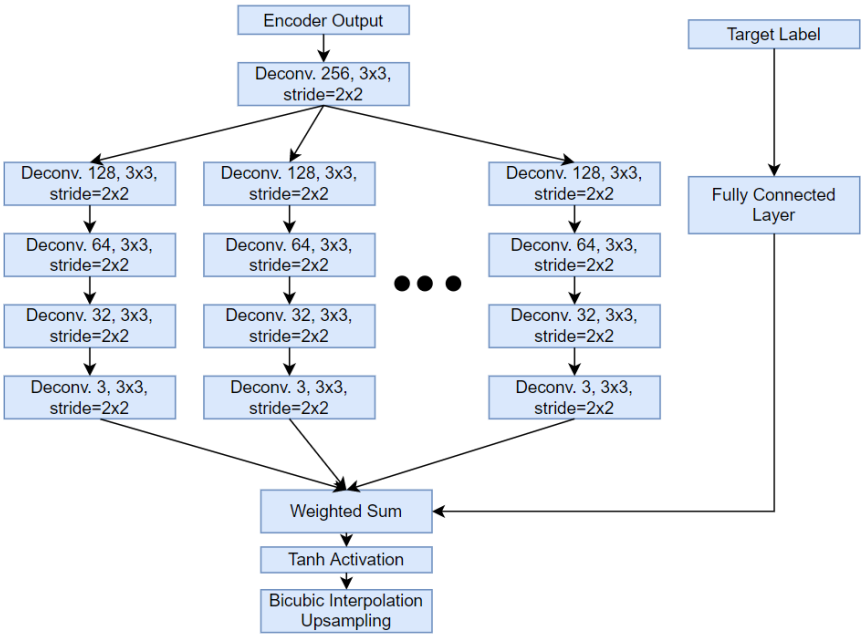

Figure 3.3: The structure of the decoder in the G-EOT generator.

The loss function used to train the generator is the same as equation 2.8 on page 17 describes, except for two things. Firstly, the adversarial texture $x + G(x,z)$ is used to render an image, represented by $t(x + G(x,z))$, and that image is given to the simulator. Secondly, it is not enough for the penalty term to be the ℓ_2 norm of the adversarial noise, as the victim model does not perceive the texture directly, but rendered images of the 3D model instead. Therefore, the penalty term used is:

$$\beta \| t(x + G(x,z)) - t(x) \|_2 \tag{3.1}$$

where x is the original texture, z is the target label, $t(\cdot)$ is the transformation function represented by the 3D rendering pipeline, and $G(\cdot,\cdot)$ is the adversarial noise created by the generator. The image with the adversarial texture and the image with the normal texture use the

same function $t(\cdot)$ so that the images will have the same pose, and the only difference is caused by the adversarial noise on the texture. I chose not to project the images into LAB space, as Athalye *et al.* did, to simplify the training task of the generator.

3.4.3 Simulator

The simulator is a CNN inspired by the SimpleNet architecture from Zheng *et al.* [3], and it is almost identical to the architecture of the encoder, though its last three convolutional layers have fewer kernels, and it has a global average pooling and a fully connected layer at the end, and it does not have the average pooling layer at the beginning. It can be seen in figure 3.4, and a more detailed diagram is in appendix A.

Moreover, the loss function for training the simulator is the same as equation 2.7 from subsection 2.3.1.

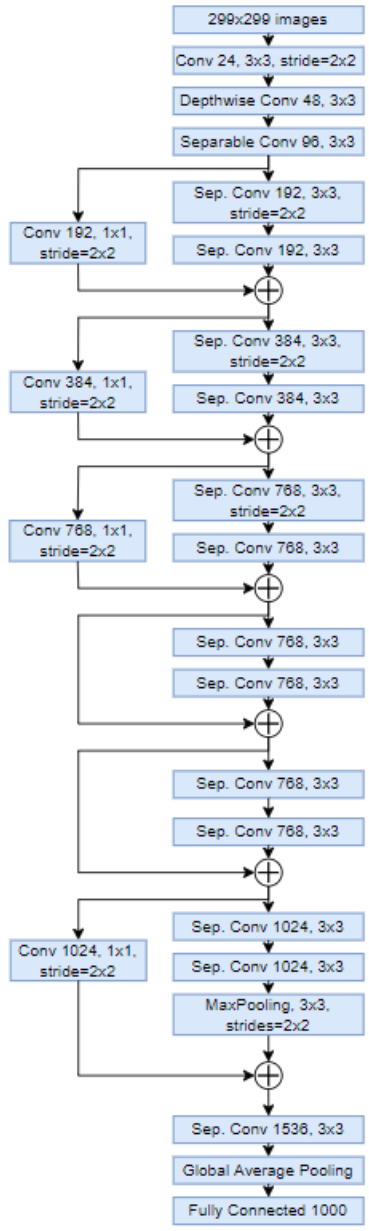

Figure 3.4: The structure of the G-EOT simulator.

3.4.4 Training techniques

Firstly, the simulator is warmed up by training it to predict the same labels as the victim model predicts on images rendered using normal textures. This is done to speed-up training, as Zheng *et al.* [3] explained.

Following that, the simulator and generator are trained in alternative steps. In each simulator training step, two steps are actually performed to train the simulator to imitate the victim model. In the first one, the simulator and victim model are given as input images rendered with normal textures. In the second one, they are given images with textures perturbed by the generator. This is done to ensure that throughout training, the simulator further learns to be an accurate imitation of the black-box victim model.

Then a training step is performed for the generator. It trains to generate adversarial noise for textures such that images rendered with those textures fool the simulator, as

described in subsection 3.4.2. The gradients of the generator loss function are clipped before they are applied, just as Zheng *et al.* [3] did for their model.

Unlike in Athalye *et al.* [2], no renders are re-used between mini-batches. This is done because it has been observed that the maximum batch size for G-EOT that can fit in 8GB of memory is only 5. Generating 5 new renders for each training step can be done fast enough, and the model trains better the more new unique samples it sees.

The source code of the implementation is available at https://github.com/Alexandru-Dascalu/G-EOT.

3.5 Analytic techniques

Each experiment will use the following metrics to evaluate the proposed attack's success:

- **Targeted Fooling Rate (TFR):** the percentage of adversarial examples that are classified by the victim model as the desired target label. Please note that TFR is the same thing as the adversariality metric used in Athalye *et al.* [2]. It is also frequently called "attack success rate" in the literature.
- **Untargeted Fooling Rate (UFR):** the percentage of adversarial examples that are classified by the victim model as any incorrect label. It is relevant because if the attack induces a misclassification, even if it is not the desired target label, then it is still potentially dangerous. It is always equal to $1 - accuracy$.
- **Classification Accuracy:** the percentage of adversarial examples classified with the correct label. It is always equal to $1 - UFR$.

Chapter 4

Results and discussion

4.1 Dataset

The data set is made up of 15 3D models, each one representing a different object which has a class in the ImageNet dataset. Each model consists of a 2D texture, a .obj file, and a text file. The .obj file contains the vertices making up the 3D object and each vertex's texture coordinates, while the text file contains a list of correct labels.

All 3D models and their textures were obtained by manually searching 3D asset sites for suitable models, and were not created by me. All credits for the models go to their respective creators, seen in Appendix B. Each model was manually re-scaled using the Blender 3D graphics tool [1] such that all 15 models are of comparable size. This was done to ensure that each model would not need different rendering parameter distributions to properly fit into the rendered image.

Out of the 15 models, 10 represent the same classes that the dataset of 10 3D models used by Athalye *et al.* [2] also represented. This was done in order to properly reproduce the experimental results of Athalye *et al.*. Please keep in mind that these 10 models are not the same ones that Athalye *et al.* used, as they did not publish their dataset. The other 5 models represent other ImageNet classes: a crocodile, a killer whale, a jeep, a running shoe and a rugby ball.

You can view the correct labels for each model in table 4.1, along with the classification accuracy of InceptionV3 on rendered images of each model.

The choice of suitable models was constrained by the following requirements:

[1] https://www.blender.org/

- The model has to be free.
- It has to have a texture, not a solid colour.
- It has to have only one texture, not multiple ones for different parts of the object, as the renderer only applies one texture to the object.
- Rendered images of the model must be classified accurately by a pre-trained InceptionV3 classifier at least 30% of the time.
- The model must not have transparent surfaces such as car windows, as the implemented renderer can not calculate UV coordinates for that surface.

Due to the above requirements, I could not find a very good 3D model for a taxi. The one I settled for has a different texture for its wheels and license plates, and the renderer simply applies parts of the main texture of the body of the taxi to the wheels, making them look yellow, as you can see in figure 4.1. Similarly, a suitable model for a sofa, as used by Athalye *et al.* [2], could not be found. Several models that I tried were classified correctly by an InceptionV3 classifier less than 10% of the time. Therefore, a 3D model of a purse is used instead.

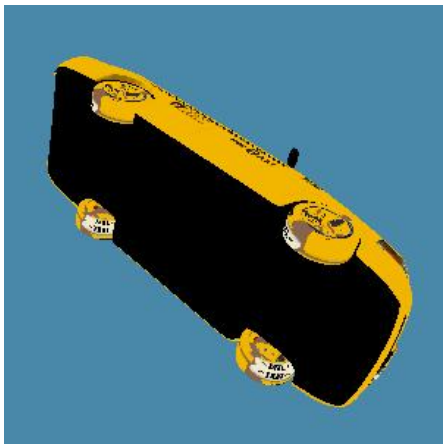

Figure 4.1: A render of the taxi model with its normal texture. As you can see, the renderer applies parts of the car body texture to the wheels.

The original textures of the model had a resolution of 1024x1024, 2048x2048 or 4096x496 pixels. Since the generator requires each input texture to have the same size, the 4096x4096 textures were downsampled to 2048x2048 using OpenCV's resize function with INTER_AREA interpolation [2]. The 1024x1024 pixels textures were upsampled to 2048x2048

[2]https://learnopencv.com/image-resizing-with-opencv/

Table 4.1: The 3D model dataset, with the correct labels of each model, and the classification accuracy of InceptionV3 on 100 rendered images of those models.

Model Name	Labels	Classification Accuracy (%)
barrel	barrel/cask (427)	92.8
baseball	baseball (429)	100
camaro	station wagon (436), race car (751), sports car (817)	10.8
clownfish	anemone fish (393)	37.2
crocodile	African crocodile (49), American alligator (50)	44
german_shepherd	All 118 dog breeds (labels 151 to 268), as well as grey wolf, white wolf, red wolf, coyote, dingo, dhole, hyena dog (labels 269 to 275)	73.2
jeep	amphibious vehicle (408), half-track (586), jeep (609)	19
orange	orange (950)	59
orca	orca (148)	69
purse	purse (748), wallet (893)	71
rugby_ball	rugby ball (768)	95
running_shoe	running shoe (770)	35
sea_turtle	loggerhead turtle (33), leatherback turtle (34), mud turtle (35), terrapin (36), box turtle (37)	89.8
taxi	taxi (468)	15
teddy	teddy bear (850)	54

pixels using OpenCV's resize function with bicubic interpolation, which is more suitable for upsampling.

Some models have multiple correct labels for two reasons. The first one is that a pre-trained InceptionV3 [34] model consistently misclassifies some rendered objects with the same wrong labels, but those labels are semantically related to the correct class. For example, the purse model is often confused with a wallet, but since both a purse and a wallet are made of the same material and have similar colours, this is considered to be meaningful. Similarly, the jeep model is frequently misclassified as a "half-track" and as an "amphibious vehicle", both of those being somewhat similar to jeeps. Secondly, some 3D models can fit in a variety of Imagenet classes. The model of a muscle car can be reasonably described as both a "sports car" and a "race car".

4.2 EOT for rendererd 3D objects

4.2.1 Experiment Design

To validate that my implementation of EOT is correct, I evaluate it by checking if the attack success rate is the same as Athalye *et al.* [2] reported on their experiments with adversarial examples for 3D rendered objects. For each 3D model, Athalye *et al.* chose 20 random target labels and used EOT to create adversarial textures. Then for each of these 200 textures, they sampled 100 different random poses and rendered images of the 3D model in that pose and with the adversarial texture. These images were fed into Tensorflow's pre-trained InceptionV3 classifier neural network, and the authors measured how often they were classified with the correct label versus the adversarial label.

The procedure used by me is similar. I use 10 out of the 15 models in the dataset from section 4.1, the ones matching the models used by Athalye *et al.*, and for each one 5 target labels are sampled from a uniform distribution, ensuring that they are different to the correct labels. Only 5 target labels are used instead of 20 due to computational and time constraints. The EOT implementation described in section 3.2 is used to create 50 adversarial textures. Tensorflow Keras's pre-trained InceptionV3 neural network is used again to evaluate rendered images of the adversarial objects. The algorithm is run until the average loss of the past 400 steps is below 0.5, or for 10000 steps at most. A constant learning rate of 0.003 is used.

Just as it was done in [2], 80% of samples from the current mini-batch are re-used in the next mini-batch. For models whose original texture was 1024x1024, I chose to use that one rather than the texture upscaled to 2048x2048 pixels, because a smaller texture allows for larger batch sizes, which gives better results. The batch size is 40 for models with 1024x1024 textures, the same batch size used in [2]. For textures with 2048x2048 pixels, the batch size is 30, the maximum that 8GB of memory can fit.

The authors of [2] say that for each model/target label pair they tried four values for the λ hyper-parameter from equation 2.5 on page 12, used for constraining perceptual difference between normal and adversarial images. They then chose the adversarial example that had the best attack success rate. They do not say what value they used for each adversarial example. Due to time and computation constraints, it was unfeasible to try 4 different λ values for each of the 50 adversarial examples. Some ad-hoc experiments done with the crocodile 3D model indicate that a value of 0.025 ensures that the adversarial texture looks similar to the original one, while still having a high attack success rate. Therefore, this was used for this experiment.

The rendering parameters are sampled from uniform distributions identical to those used by Athalye *et al.* [2], which you can see in table 2 of their supplementary material. Translation on the X/Y axes is quite small, between -0.05 and 0.05. The rotation angle on all three axes is drawn from an unbounded uniform distribution. The camera distance is between 1.8 and 2.3, while in [2] it was between 2.5 and 3. The reason behind this is that the 3D models in my dataset appear to be smaller than those used in [2], and so the camera distance was reduced to make them appear to the camera as large as they did in Athalye *et al.*. You can see in figure 4.2 that with the adjusted camera distance, the barrel model appears roughly as large as the model used in [2]. Finally, because this experiment is on evaluating EOT on 3D rendered objects and not physical 3D printed adversarial objects, printer and camera errors are not modelled, and the apply_print_error and apply_photo_error functions from algorithm 1 on page 24 are not called.

A. Model used in [2]. B. Model used by this project.

Figure 4.2: A comparison of an image of the barrel model used in Athalye *et al.* versus an image of the barrel model used by this project.

After creating the 50 adversarial textures, the same renderer used during the EOT optimisation process is used to create the evaluation images. For each adversarial texture, 100 images of the rendered object with that texture are created. For each image with the adversarial texture, an image of the object in the same pose and with the same background colour, but with the normal texture, is rendered. You can see an example of this in figure 4.3. This is done to observe the effect of using the adversarial texture rather than the original one, as all other factors are the same in the two images.

A. Adversarial image. B. Normal image.

Figure 4.3: Comparison between an evaluation image of the dog model with the adversarial texture versus its normal counterpart. The adversarial texture is for the "black grouse" target label.

4.2.2 Experiment Results

On the 5000 images with normal textures, the average classification accuracy was 60.28%, while only 0.08% of them were classified as the adversarial target label. By comparison, the 20000 normal images in Athalye *et al.* [2] had an accuracy of 68.8% and only 0.01% were classified as the target label. These figures are similar, although my images had a significantly lower classification accuracy. This is caused by the classification accuracies for normal images of the taxi and camaro models being only 15 and 10.8, as you can see in table 4.1. Without these two models, the average accuracy would be 72.125%.

But on the adversarial images, the average classification accuracy was just 0.86%, very close to the EOT paper's result of 1.1% [2]. Meanwhile, the average TFR was 80.08%, slightly under the TFR of 83.4% seen in the EOT paper. The mean TFR and the target labels for each individual model can be seen in table4.2. The slightly worse performance may be caused by a multitude of factors. Even though the dataset described in section 4.1 is designed to mimic the one used in [2], it is not identical. Furthermore, different values for the λ penalty constant, learning rate, and the number of optimisation steps were used compared to [2], as the authors did not specify what values they used.

The fact that the adversarial textures brought down the classification accuracy from 60.28%

to 0.86%, and increased the TFR from 0.08% to 80.08%, demonstrates that this implementation of EOT [3] is highly effective at creating robust adversarial attacks for 3D rendered objects. Moreover, the results are very similar to those in [2], demonstrating that it is an accurate re-creation of the implementation used by the authors of that paper.

In figure 4.4 you can see a histogram of the TFR for each of the 50 adversarial textures. You can see that 27 out of 50 adversarial textures had a mean TFR of over 90%. Meanwhile, just 2 adversarial textures are failures, with a mean TFR under 20%. This demonstrates that the vast majority of adversarial textures for 3D objects are highly successful.

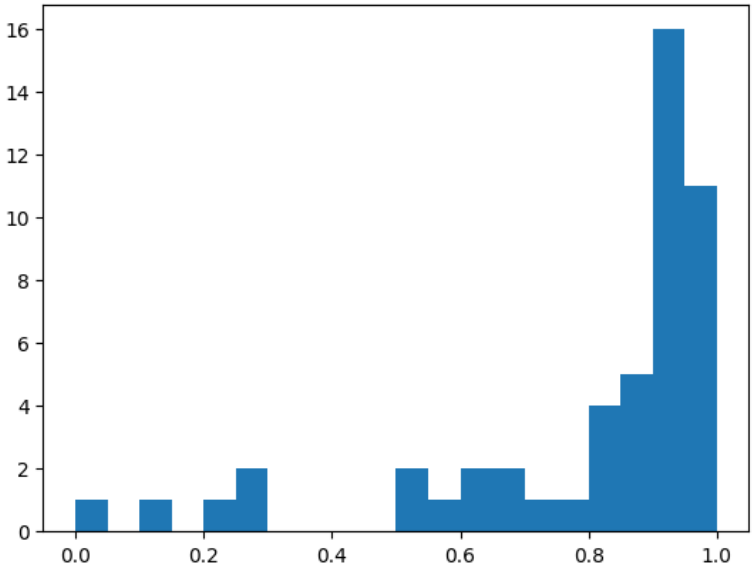

Figure 4.4: Histogram of TFR of the 50 adversarial examples.

It took 82 hours to create the 50 adversarial textures on an RTX 3070 with 8GB of VRAM. That GPU can do around 2900 EOT optimisation steps in an hour, and therefore the creation of a single adversarial texture can take up to around 3.4 hours.

Links to all 50 adversarial textures created by and used in this experiment can be seen in appendix C. Moreover, that appendix has links to the Jupyter notebooks that contain proof of the experiment and the full results, including optimisation loss history, TFR and classification accuracy for each adversarial texture.

[3] https://github.com/Alexandru-Dascalu/adversarial-3d

Table 4.2: Results of the EOT experiment on rendered 3D objects.

Model Name	Target Labels	Mean TFR (%)	Mean number of iterations
barrel	file cabinet, hourglass, hautboy, purse, typewriter keyboard	95.59	1428.4
baseball	Australian terrier, dingo, indian elephant, radiator, warplane	80.6	6349.2
camaro	kite, mongoose, indri, trombone, red-breasted merganser	93.39	2585.8
clownfish	swimming cap, grand piano, shopping basket, wall clock, bridegroom	48.2	10000
german_shepherd	bald eagle, barrel, bikini, black grouse, hornbill	85.6	5243
orange	whippet, goblet, lighter, missile, plate	97.4	1163.2
purse	pit bull terrier, chickadee, dhole, weevil, sandbar	42.8	9090.6
sea_turtle	Australian terrier, leopard, mantis, cradle, plunger	80.39	5123
taxi	groenendael, marmot, acoustic guitar, cradle, radiator grille	87.8	4229.6
teddy	soft-coated wheaten terrier, leatherback turtle, pool table, hot pot, pineapple	89	2782.8

4.2.3 Discussion

4.2.3.1 Classification accuracy on normal images

The very low InceptionV3 classification accuracies for normal texture images of the taxi and camaro models of 15 and 10.8 per cent, respectively, might be caused by the fact that the objects are rendered with completely random rotations on the X, Y and Z axes. Therefore, these cars are often seen from below or above. The images of taxis or sports cars in the Imagenet dataset are of cars as they appear on the street, and therefore seen from the side, front or back, not from above or underneath.

Then again, the classification accuracy on images with the normal texture does not seem to impact the performance of EOT. The adversarial textures for taxi and camaro models had a mean TFR of 87.8% and 93.39%, respectively.

4.2.3.2 EOT performance depending on the model

Out of the two adversarial textures with a mean TFR of under 20%, one is the adversarial texture for the clownfish model and the grand piano target label, the other is the one for the purse model and the pit bull terrier label. Even without those two, the mean TFRs of the clownfish and purse adversarial textures are around only 50%. The ability of EOT to create adversarial examples seems to be influenced by the perceptual difference between the model and the target label. A clownfish has a vastly different shape and colour compared to a piano. In particular, the colour pattern of a clownfish is very distinct and different from a piano, and likely prevents the renders of the adversarial clownfish to be classified as a piano.

Similarly, the brown purse model has a very different colour from a pit bull terrier, resulting in a TFR of 1% for that model. However, the purse adversarial texture for the Asian wild dog target label had a mean TFR of 62%. The contrast may be explained by the fact that the wild dog is brown in colour, and thus already perceptually closer to the brown purse.

Models with a simple shape, similar to a sphere, and/or a colour pattern dominated by one simple colour, seem to be a better "canvas" for EOT to apply adversarial noise on. The barrel and orange 3D models have a mean TFR of over 90%, as you can see in table 4.2. The camaro and taxi models mostly have a solid yellow colour and they have a mean TFR of over 85%. On the other hand, models with more complicated shapes like the sea turtle and the german shepherd have a lower mean TFR.

A simple colour ensures that EOT can gradually develop adversarial patterns more easily, while a simpler shape means that the model looks similar regardless of the angle it is viewed from, which makes the gradient fluctuate less despite the random poses the model is rendered in.

4.2.3.3 Number of iterations

As subsection 4.2.1 mentioned, the algorithm stopped when the average loss of the past 400 steps was below 0.5 or ran for at most 10000 steps. Table 4.2 displays the average number of iterations it took to create the five adversarial examples for each model. It is likely that the threshold for stopping the algorithm could have been set to 0.8 instead of 0.5 to reduce runtime while obtaining similar TFRs.

Unsurprisingly, the models with the lowest mean TFR, the clownfish and the purse, also took the longest to create. Meanwhile, the models with a simpler texture and colour were easier

to optimise. For example, it took EOT on average just 1163.2 iterations to run for the orange model.

The number of necessary optimisation steps is influenced by the somewhat random nature of the gradient, caused by the fact that each rendered image in the batch is random. This perceptual randomness is smaller for models with a simple shape, as they look similar regardless of the angle they are seen from. As you can see in figure 4.5, the loss value fluctuates very little for the orange model, leading to much faster convergence. At the other extreme, the loss value for the clownfish model and shopping basket target label fluctuates wildly, sometimes from a value of 1 to 3 or even 4, as you can see in figure 4.6. The "Main loss" in these loss history plots refers to the value of the first term in equation 2.5 in subsection 2.2.1.1, while "L2 Loss" refers to the penalty constraining the size of the adversarial perturbation in that equation. "Total Loss" refers to the sum of these two terms.

Figure 4.5: Optimisation history for the orange model and target label goblet. L2 loss refers to the penalty term in equation 2.5, main loss refers to the first term. Its evaluation TFR is 100%.

Another interesting observation is that the longer the EOT algorithm has to run, the more noisy the created adversarial textures will be. You can see this in the textures in figures 4.7 and 4.8, both of which took the maximum number of 10000 steps to create.

Figure 4.6: Optimisation history for the clownfish model and target label shopping basket. Its TFR is 71%.

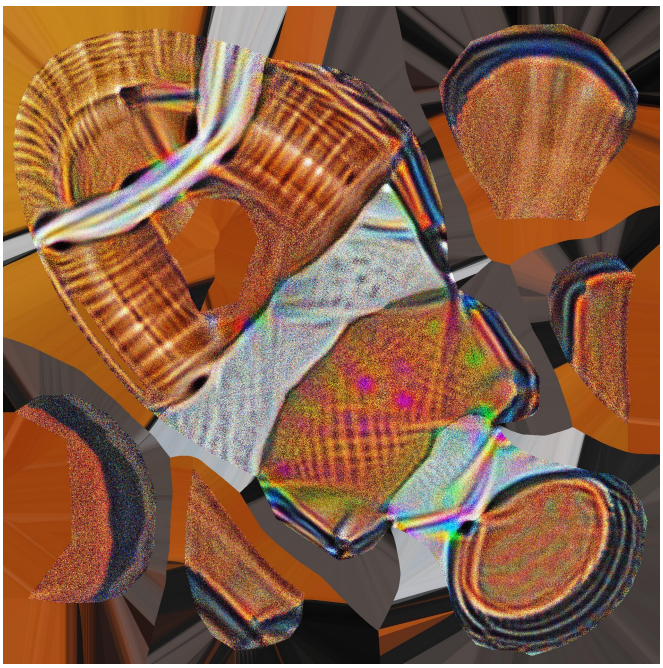

Figure 4.7: Adversarial texture for the clownfish model and target label shopping basket. Its TFR is 71%.

Figure 4.8: Adversarial texture for the german shepherd model and target label barrel. Its TFR is 76%.

4.2.3.4 Reusing renders between optimisation steps

Some ad-hoc experiments done with the barrel model, before the experiment in subsection 4.2.1, show that re-using 80% of renders versus re-using none has little effect in terms of the TFR of adversarial texture. However, it greatly speeds up the EOT algorithm, as it can do the same number of optimisation steps in less than half the time. It shows that one of the most computationally intensive parts of EOT is the computation of UV maps and using the latter to render 2D images of the objects, as Athalye *et al.* [2] also suggested.

4.2.3.5 Semantic adversarial perturbations

A key characteristic of the adversarial noise made for textures of 3D objects by EOT is that they show semantically meaningful patterns related to the target class. A different experiment from the one described in subsection 4.2.1, on the crocodile 3D model and target label compass, resulted in the texture in figure 4.9. EOT added noise on the belly and back of the crocodile with an oval or circular shape, the same shape a compass has. Similarly, you can see in figure 4.10 that EOT added green spots, reminiscent of the green seeds on a strawberry, for an adversarial texture of the running shoe model for the strawberry target label.

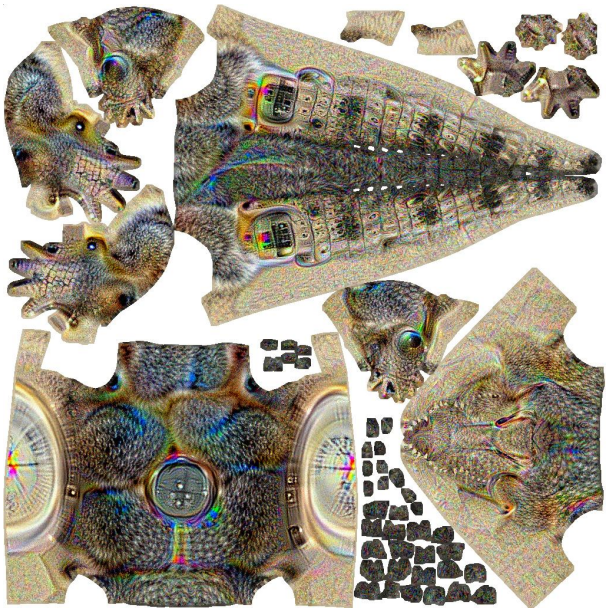

Figure 4.9: Adversarial texture for the crocodile model and target label compass. It has a TFR of 60%.

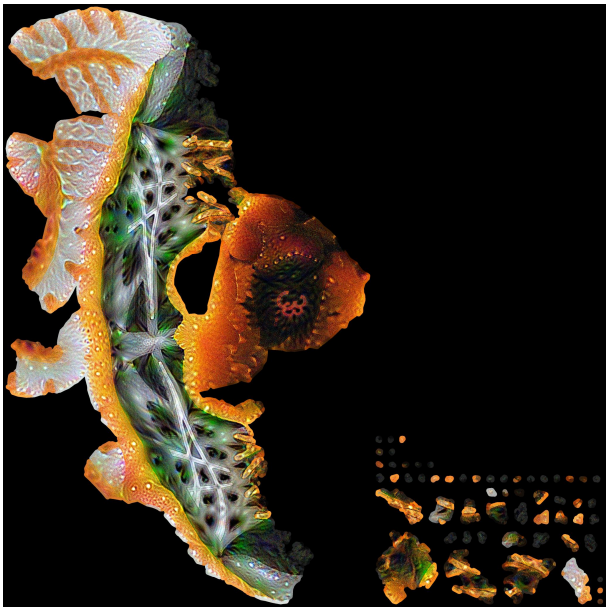

Figure 4.10: Adversarial texture for the running shoe model and target label strawberry. It has a TFR of over 80%.

Another example is in figure 4.11, where the adversarial texture for the running shoe and target label rifle exhibits metallic patterns in the shape of a gun barrel. Given that these three adversarial textures have a significant TFR, they highlight features that the InceptionV3 victim

model learned for recognising a compass, strawberry and rifle, respectively. Therefore, EOT can be used to study the explainability of a neural network, as it highlights class features that make the neural network classify a completely different object as that class.

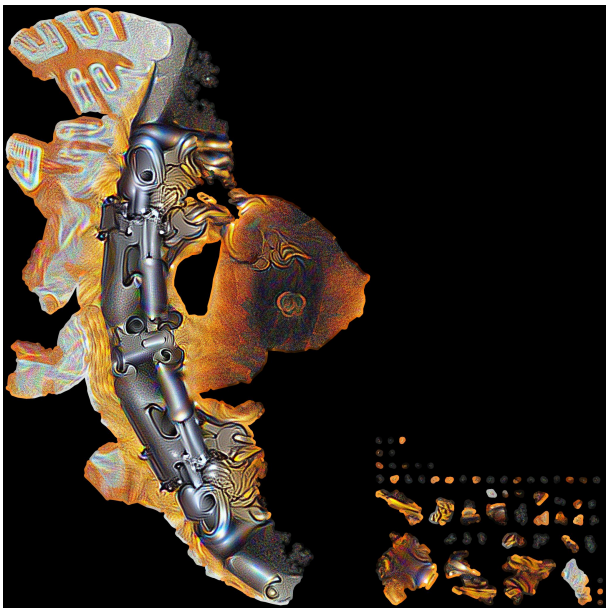

Figure 4.11: Adversarial texture for the running shoe model and target label rifle. It has a TFR of 45-50%.

4.3 Generator-simulator model for black-box attacks

4.3.1 Experiment Design

The implementation[4] of Zheng *et al.* [3] made by the authors of that paper was used for this experiment, whose purpose is to see if the implementation produces the same results as in the paper.

The model is trained to create adversarial noise for images from the CIFAR10 dataset. Both the simulator and generator use an exponentially decaying learning rate whose initial value is 0.001. The decay rate is 0.95, and it decays every 300 steps. However, it stops decreasing when it hits the minimum value of $2 * 10^{-5}$.

These experiments used a victim model with the SmallNet architecture, trained on CIFAR-10. The batch size is 200 for the experiment that uses SmallNet and SimpleNet as simulators,

[4]https://github.com/Alexandru-Dascalu/targeted-black-box-attack

Table 4.3: TFR results of experiment evaluation the implementation of the generator-simulator attack model on CIFAR-10.

Simulator Architecture	TFR	TFR in Zheng *et al.* [3]
SmallNet	85.63%	85%
SimpleNet	84.15%	84.3%
ConcatNet	79.69%	87.3%
Xception	86.62%	86.9%

and 100 for the ConcatNet and Xception simulators, the maximum that can fit in 8 GB of VRAM. This experiment ran on an RTX 3070 GPU.

4.3.2 Experiment Results

You can see in table 4.3 the TFR for the experiment with each of the 4 simulators, along with the TFR result of the equivalent experiment from Zheng *et al.* [3]. Moreover, the UFRs for each simulator architecture were over 90%.

The results obtained in this experiment are very close to those reported in [3], demonstrating that the implementation is successful at creating CIFAR-10 adversarial images against black-box victim models. Since my fork of the implementation of the authors of [3] did not change the code for the main algorithm and only refactored the code slightly, it was decided that repeating this experiment for other victim models was not necessary and that the implementation is a good starting point for G-EOT.

4.4 G-EOT

4.4.1 Experiment Design

The purpose of this experiment is to see if G-EOT is able to learn to create adversarial noise for textures. The model is evaluated by training it using the process described in subsection 3.4.4 to attack Tensorflow Keras' pre-trained Xception neural network classifier [38]. This target was chosen because the simulator's architecture is inspired by Xception, and therefore training the simulator to distil that model should be easier compared to a different architecture.

Firstly, the simulator is warmed-up for 2000 training steps. Following that, the G-EOT model is trained for 40000 steps, where each one consists of one simulator training step and one generator training step. This number was chosen because training the model made by

Zheng *et al.* [3] took around 30000 steps to achieve the results seen in their paper, and G-EOT has a harder generation task. Every 500 training steps, the generator is evaluated on 200 batches of new data samples that the generator has not seen before.

The model is trained on the dataset of 15 3D models presented in section 4.1. At each training step, the batch is made up of randomly picked models out of this dataset. For each model in the batch, random parameters are sampled for translation, rotation and distance from the camera, and a UV map is computed based on those. Moreover, a random target label is sampled from a uniform distribution. It can be any of the 1000 Imagenet labels, except the correct labels for that model. Therefore, each element in the batch is a randomly picked model, with a UV map for a random pose and a random target label. In this experiment, the batch size is 5, as that is the maximum that will fit in 8 GB of VRAM. An Nvidia RTX 3070 graphics card is used for this experiment.

The bounds for the uniform distributions used for sampling the camera distance, X/Y translation, additive and multiplicative lighting, and the standard deviation of the gaussian noise, are the same as those used for the EOT experiment in subsection 4.2.1. This experiment does not scale the texture before rendering in order to simulate 3D printer errors, as the adversarial 3D models will not be physically printed.

The number of experts in the decoder is 50, as having one expert for each of the 1000 Imagenet labels would be too much, and different combinations of expert outputs can be used for each target label. The simulator uses an Adam optimiser with an exponentially decaying learning rate. The initial learning rate is 0.001, with a decay rate of 0.98, and it decays after every 300 steps. The generator uses a constant learning rate of 0.004. The reason is that the random renders can lead to strong fluctuations in the loss function value, as you have seen in subsection 4.2.2, and a large constant learning rate mitigated that and improved performance for the EOT experiment in 4.2.2. The gradients of the generator loss function are clipped to a range between -1 and 1, as Zheng *et al.* [3] also did.

The noise produced by the G-EOT generator has individual values for each pixel between -25 and 25, to be applied to images with pixel values between 0 and 255. The reason behind this limitation is that when training starts, the randomly initialised weights are very small, and therefore the values of the generated noise tend toward 0. When they are scaled to values between -1 and 1, these values are mostly -1. When added to a texture with pixel values between 0 and 1, this resulted in a completely black image. The simulator would not provide a useful gradient to the generator when given a black image, and therefore I introduced this

limitation to avoid this. Further work should look into initialisation techniques to avoid this issue.

The weight β from equation 3.1 on page 31 has a value of 0.001, chosen to relax the constraint on the size of the adversarial noise and make the learning task for the generator easier. Finally, the weight for L2 regularisation of the weights of the simulator and generator is $0.5 * 10^{-4}$, and the simulator and generator's weights are initialised with Orthogonal Initialisation, which was used in BigGAN [43].

4.4.2 Experiment Results

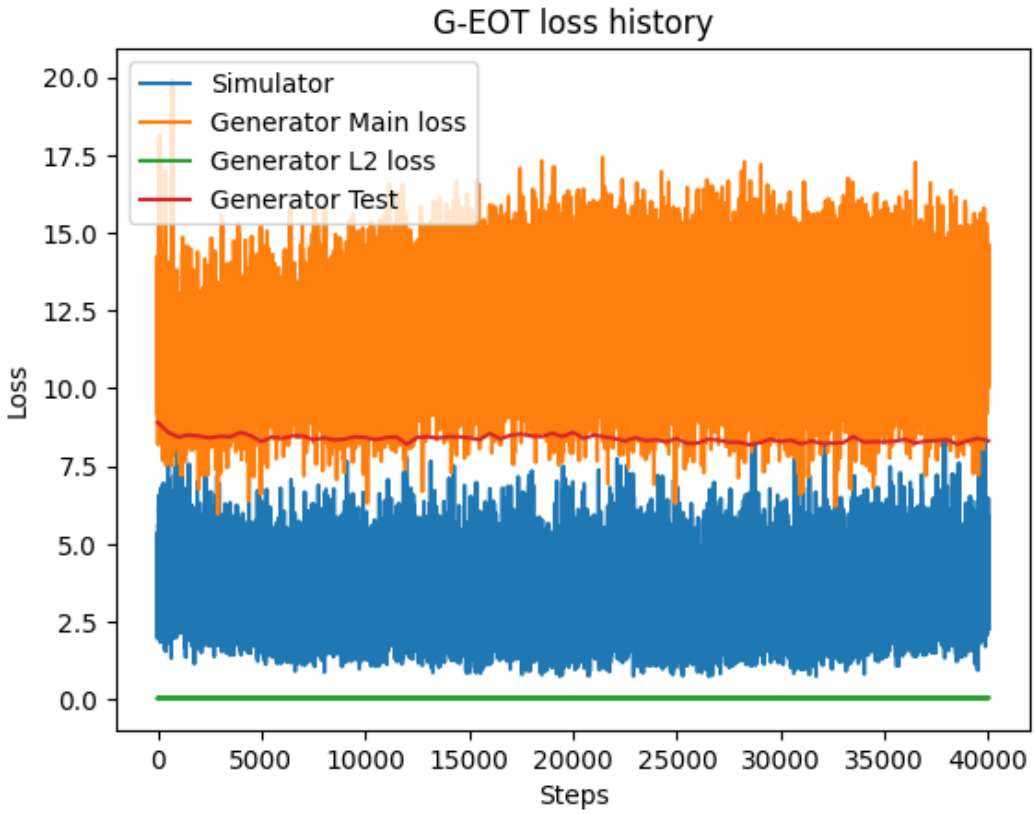

Figure 4.12: Loss history during training for the G-EOT model.

You can see a history of the loss of the model during training in figure 4.12. The orange plot denotes the value of the first term of equation 2.8 on page 17, which measures how close is the generator to fooling the simulator. The green plot denotes the value of the term seen in 3.1 on page 31, which constrains the size of the adversarial noise. The red line represents the

average generator loss at each evaluation step, which is done once every 500 training steps, as mentioned in the previous subsection. Moreover, this test loss is calculated by using the victim model rather than the simulator, as fooling the victim model is the intended objective.

The figure shows that the gradient for the generator fluctuates wildly between training steps. Moreover, despite a small decrease in the generator main loss and the generator test loss at the very beginning, they do not decrease further during training. Therefore, the generator failed to learn almost anything, having a TFR of almost 0% for new random renders sampled during evaluation steps. Interestingly, the loss of the simulator also varies wildly between training steps and does not have a downward trend throughout training, despite its training task being a lot simpler. The loss of the simulator is a lot lower than the generator's, likely due to the simulator warm-up before training, at the end of which the simulator had an accuracy of 28.4% on never before seen test samples.

The simulator has 9.75 million trainable parameters, almost half of the 22.8 million trainable parameters of Xception [38]. Given that distillation allows models to learn to imitate much larger models [35], and that during warmup the simulator's accuracy went from 0 to 28.4%, the training technique of the simulator should not be a cause of the wild fluctuations of the simulator loss seen in figure 4.12. Instead, the intuition is that the generator keeps changing its pattern of adversarial noise, and therefore the simulator has a high loss when trying to predict the same labels that the victim model predicts on the adversarial images.

The training task for the generator is significantly more difficult than that for the generator in Zheng *et al.* [3]. Its task is to learn to generate adversarial textures for 1000 target labels for each of the 15 3D models, such that the textures remain adversarial regardless of the random rendering of the model. You can see in figure 4.6 in page 44 that using EOT to optimise just one texture for one target label can have large variations in the gradient between consecutive steps, and it takes thousands of optimisation steps with a large batch size to produce good results. A hypothesis for why the generator does not learn much is that the gradient changes too much between training steps, and therefore does not provide a useful training direction.

To test this hypothesis, I simplified the training task so that the generator only learns to create perturbations for 10 target labels. These are Imagenet labels 281 to 290, which represent 5 different cat breeds, puma, lynx, leopard, snow leopard and jaguar. These felines have specific fur colour patterns that should be easy to learn to generate and apply to textures. Consequently, the number of experts in the decoder was reduced from 50 to 10, and the smaller model allowed the batch size to be increased to 9. All other hyper-parameters are the same.

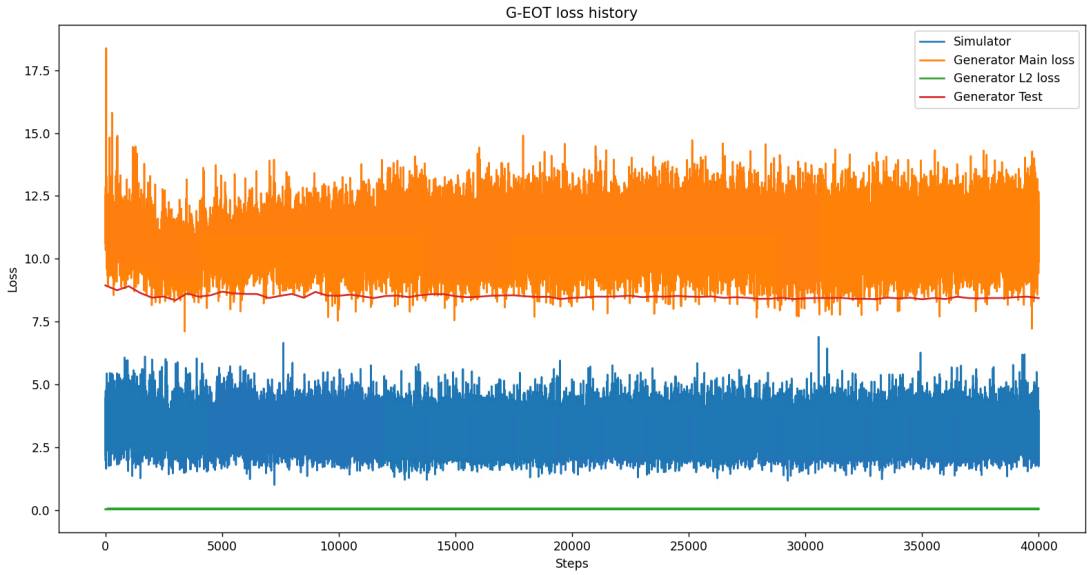

Figure 4.13: Loss history during training for the G-EOT model for 10 target labels.

You can see the result in figure 4.13. Unfortunately, the generator and simulator loss exhibit the same behaviours. While the generator's main loss and test loss decreased during the first 2000 steps, they made no improvements after. The value of the main term of the generator loss during training still has wild variations. However, while in figure 4.12 the generator main loss varies between 7.5 and 15, in figure 4.13 it varies roughly between 8.5 and 13.5. This shows that decreasing the number of target labels made the task easier, but it was not enough.

4.4.3 Discussion

As mentioned in the previous subsection, the intuition is that the generator failed to learn much due to the random image renders not giving a stable enough gradient. This issue was seen in the EOT experiment in subsection 4.2.2 for just 1 texture and one target label, yet the Adam optimiser could overcome that. However, trying to learn a distribution of adversarial noise for 15 textures and 1000 or 10 target models was too much. Therefore, more background reading needs to be done on initialisation, regularisation, training techniques, etc. for generative models that would mitigate the issue of the random gradient.

With 50 experts in the decoder, the generator has 36.1 million trainable parameters, and with 10 experts it has 20.5 million trainable parameters. Brock *et al.* [43] used models with between 71.3 and 173.5 million parameters to generate images with sizes between 128x128 and 512x512. The G-EOT model may be able to learn better if it has a lot more parameters.

However, such a large model would probably not be able to use a batch size larger than 1 with just 8 GB of VRAM. Therefore, a lot more computational resources might be needed to train a model such as G-EOT.

Moreover, even if a modified G-EOT would be able to complete its learning task, it might not be that useful for an attacker. In the experiment in subsection 4.2.2, EOT took on average 1.64 hours to create an adversarial texture, with 3.3 hours being the maximum. A malicious agent would attack a neural network with a specific goal in mind, and therefore with a certain adversarial object and with a specific target label. They might be able to choose a 3D model of their choosing. For example, if they want to fool a facial recognition model in the real world, they may choose to add adversarial noise to a face mask, a pair of glasses, or something else worn by a person. The target label would be a different person, who has access to the room or space to which access is restricted by facial recognition. In another scenario, the attacker may want to trick a self-driving car into doing a particular action when seeing an adversarial road sign. In this case, the type of road sign to which adversarial noise is applied is not controlled by the attacker. But in both scenarios, the attacker only needs adversarial noise for one object and one target label, not for any of the 15 3D models and any arbitrary target label. And training G-EOT for 40000 steps took around 34.6 hours, more than 10 times more than it takes to use EOT to create an adversarial texture for a 3D model. Therefore, EOT would be more convenient for an attacker and is suitable for their task. Although EOT, as seen in Athalye *et al.* [2], is a white-box attack method, it can become a black-box method by using distillation [35] to train a simulator which behaves just like the victim model and then differentiate through the simulator to create the adversarial texture. Training the simulator should not take more than a couple of hours.

On the other hand, G-EOT could still be useful for adversarial training, which entails including adversarial examples in the training set of machine learning models so that they learn to classify them with the correct label, thus making the model secure against adversarial attacks. Once trained, G-EOT would be able to create adversarial textures for rendered 3D objects near instantly [5], and would therefore be excellent for augmenting the dataset of natural data samples. By contrast, EOT would take 1025 days on an RTX 3070 to create an adversarial texture for every Imagenet target label for each of the 15 3D models, assuming it would take an average of 1.64 hours to create one. This would take too long, and if there is a method to successfully train a modified G-EOT model, it would likely be quicker.

Chapter 5

Conclusions and Future Work

Deep neural networks are used in a variety of safety-critical systems, yet they are vulnerable to adversarial attacks, which create subtle noise that makes the victim model produce the wrong output. These attacks are applicable to physical objects and can be highly successful. Moreover, there are attack methods that can work with a black-box victim model.

In this project, I created G-EOT, the first generative model for creating 3D adversarial objects to fool a black-box victim model, that remain adversarial regardless of the position they are seen from. The experiments done so far show that the model is unable to learn much. Moreover, the experiments show that the issue is not caused by the number of target labels for which it needs to learn to create adversarial noise. Therefore, I hypothesize that the main issue is that the random poses of the object make the gradient vary too much. More work is needed to see if this issue can be overcome.

Furthermore, the runtime of the experiments for the evaluation of EOT [2] and G-EOT shows that for attackers, it would be far more convenient to train a simulator to behave like the black-box model and then use that simulator with EOT, rather than using G-EOT. The latter would take longer to train than it takes EOT to create a single adversarial 3D object. However, a generator that can create adversarial examples for many different 3D models and for any target label would still be useful to augment datasets for adversarial training, used so that neural networks are immune to adversarial attacks.

5.1 Future Work

There are several research directions that could be undertaken to create a better method for creating black-box adversarial attacks for 3D objects:

- Research training techniques, regularisation techniques, architectures and hyper-parameter tuning that could make G-EOT successfully learn. Among these, a larger generative model and larger batch sizes are particularly likely to be useful [43], though more computational resources are needed for those.
- Modify the dataset of 3D models so all models have 1024x1024 textures rather than 2048x2048. The textures would occupy less memory, allowing for a larger batch size with the same amount of VRAM.
- Experiment on a black-box version of EOT. This version would use distillation [35] to train a simulator to behave like the black-box victim model. Since the attacker has access to the simulator's architecture and parameters, they can differentiate through it, and therefore use the white-box EOT framework. The experiments should look at how effective the adversarial textures made by EOT are on the simulator versus the victim model.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, 2012.

[2] A. Athalye, L. Engstrom, A. Ilyas, and K. Kwok, "Synthesizing robust adversarial examples," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 7 2018, pp. 284–293. [Online]. Available: https://proceedings.mlr.press/v80/athalye18b.html

[3] S. Zheng, J. Chen, and L. Wang, "Targeted black-box adversarial attack method for image classification models," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019, pp. 1–8.

[4] S.-M. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 07 2017.

[5] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating adversarial examples with adversarial networks," in *Proceedings of the 27th International Joint Conference on Artificial Intelligence*, ser. IJCAI'18. AAAI Press, 2018, p. 3905–3911.

[6] M. M. Kasar, D. Bhattacharyya, and T. Kim, "Face recognition using neural network: a review," *International Journal of Security and Its Applications*, vol. 10, no. 3, pp. 81–100, 2016.

[7] B. T. Nugraha, S.-F. Su, and Fahmizal, "Towards self-driving car using convolutional neural network and road lane detector," in *2017 2nd International Conference on Au-*

tomation, Cognitive Science, Optics, Micro Electro-Mechanical System, and Information Technology (ICACOMIT), 2017, pp. 65–69.

[8] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial examples for malware detection," in *Computer Security – ESORICS 2017*, S. N. Foley, D. Gollmann, and E. Snekkenes, Eds. Cham: Springer International Publishing, 2017, pp. 62–79.

[9] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/60a6c4002cc7b29142def8871531281a-Paper.pdf

[10] Y. Gao, C. Xu, D. Wang, S. Chen, D. C. Ranasinghe, and S. Nepal, "Strip: A defence against trojan attacks on deep neural networks," ser. ACSAC '19. San Juan, Puerto Rico, USA: Association for Computing Machinery, 2019, p. 113–125. [Online]. Available: https://doi.org/10.1145/3359789.3359790

[11] L. Muñoz-González, B. Biggio, A. Demontis, A. Paudice, V. Wongrassamee, E. C. Lupu, and F. Roli, "Towards poisoning of deep learning algorithms with back-gradient optimization," in *Proceedings of the 10th ACM workshop on artificial intelligence and security*, 2017, pp. 27–38. [Online]. Available: https://doi.org/10.1145/3128572.3140451

[12] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations*, 2014. [Online]. Available: http://arxiv.org/abs/1312.6199

[13] K. Eykholt, I. Evtimov, E. Fernandes, B. Li, A. Rahmati, C. Xiao, A. Prakash, T. Kohno, and D. Song, "Robust physical-world attacks on deep learning visual classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 6 2018.

[14] N. Papernot, P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami, "Practical black-box attacks against machine learning," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '17. Abu Dhabi, United Arab Emirates: Association for Computing Machinery, 2017, p. 506–519. [Online]. Available: https://doi.org/10.1145/3052973.3053009

[15] J. Lu, H. Sibai, E. Fabry, and D. Forsyth, "No need to worry about adversarial examples in object detection in autonomous vehicles," 2017. [Online]. Available: https://arxiv.org/abs/1707.03501

[16] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," *IEEE Access*, vol. 6, pp. 14 410–14 430, 2018.

[17] S. H. Silva and P. Najafirad, "Opportunities and challenges in deep learning adversarial robustness: A survey," 2020.

[18] Y. Dong, Q.-A. Fu, X. Yang, T. Pang, H. Su, Z. Xiao, and J. Zhu, "Benchmarking adversarial robustness on image classification," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 321–331.

[19] S. Tang, R. Gong, Y. Wang, A. Liu, J. Wang, X. Chen, F. Yu, X. Liu, D. Song, A. Yuille, P. H. S. Torr, and D. Tao, "Robustart: Benchmarking robustness on architecture design and training techniques," 2022.

[20] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015.

[21] J. Hendrik Metzen, M. Chaithanya Kumar, T. Brox, and V. Fischer, "Universal adversarial perturbations against semantic image segmentation," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 10 2017.

[22] S. Sarkar, A. Bansal, U. Mahbub, and R. Chellappa, "Upset and angri : Breaking high performance image classifiers," 2017.

[23] X. Yuan, P. He, Q. Zhu, and X. Li, "Adversarial examples: Attacks and defenses for deep learning," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, no. 9, pp. 2805–2824, 2019.

[24] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[25] J. Kos, I. Fischer, and D. Song, "Adversarial examples for generative models," in *2018 ieee security and privacy workshops (spw)*. IEEE, 2018, pp. 36–42.

[26] P. Tabacof, J. Tavares, and E. Valle, "Adversarial images for variational autoencoders," 2016.

[27] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting adversarial input sequences for recurrent neural networks," in *MILCOM 2016 - 2016 IEEE Military Communications Conference*, 2016, pp. 49–54.

[28] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[29] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *2017 ieee symposium on security and privacy (sp)*. IEEE, 2017, pp. 39–57.

[30] D. Krotov and J. Hopfield, "Dense associative memory is robust to adversarial inputs," *Neural computation*, vol. 30, no. 12, pp. 3151–3167, 2018.

[31] T. Tanay and L. Griffin, "A boundary tilting persepective on the phenomenon of adversarial examples," 2016.

[32] A. Ilyas, S. Santurkar, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Adversarial examples are not bugs, they are features," in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/e2c420d928d4bf8ce0ff2ec19b371514-Paper.pdf

[33] K. McLAREN, "Xiii—the development of the cie 1976 (l* a* b*) uniform colour space and colour-difference formula," *Journal of the Society of Dyers and Colourists*, vol. 92, no. 9, pp. 338–341, 1976. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1478-4408.1976.tb03301.x

[34] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[35] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015. [Online]. Available: https://arxiv.org/abs/1503.02531

[36] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017. [Online]. Available: https://arxiv.org/abs/1701.06538

[37] 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.

[38] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 07 2017.

[39] 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.

[40] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *stat*, vol. 1050, p. 10, 2014.

[41] 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.

[42] A. Madry. Cifar10 adversarial examples challenge. [Online]. Available: https://github.com/MadryLab/cifar10_challenge

[43] A. Brock, J. Donahue, and K. Simonyan, "Large scale gan training for high fidelity natural image synthesis," in *International Conference on Learning Representations*, 2018.

[44] A. Athalye. A step-by-step guide to synthesizing adversarial examples. [Online]. Available: https://www.anishathalye.com/2017/07/25/synthesizing-adversarial-examples/

[45] W. Zhang. A tensorflow implementation for synthesizing robust adversarial examples. [Online]. Available: https://github.com/ring00/adversarial-3d

[46] LearnOpenGL.com. Learnopengl - shaders. [Online]. Available: https://learnopengl.com/Getting-started/Shaders

[47] ——. Learnopengl - textures. [Online]. Available: https://learnopengl.com/Getting-started/Textures

[48] Tensorflow. tfa.image.resampler. [Online]. Available: https://www.tensorflow.org/addons/api_docs/python/tfa/image/resampler

[49] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

Appendix A

Detailed architecture of G-EOT

The following two figures were generated using the tensorflow.keras.utils.plot_model method.

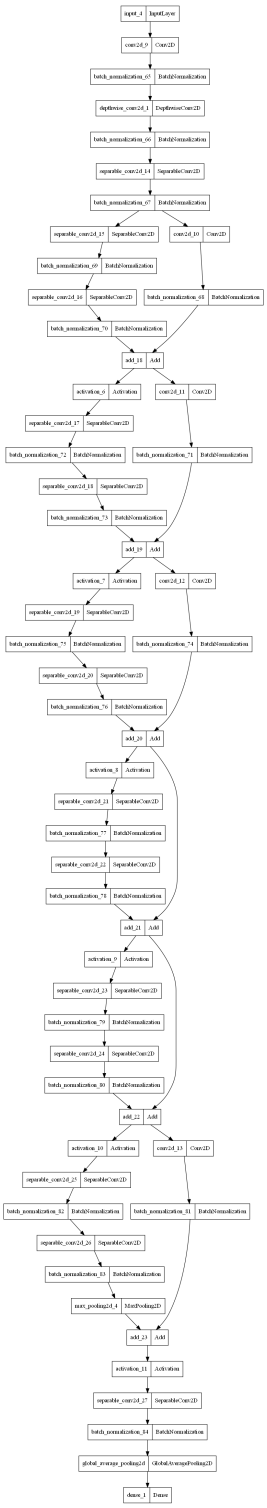

Figure A.1: Detailed diagram of the simulator of G-EOT.

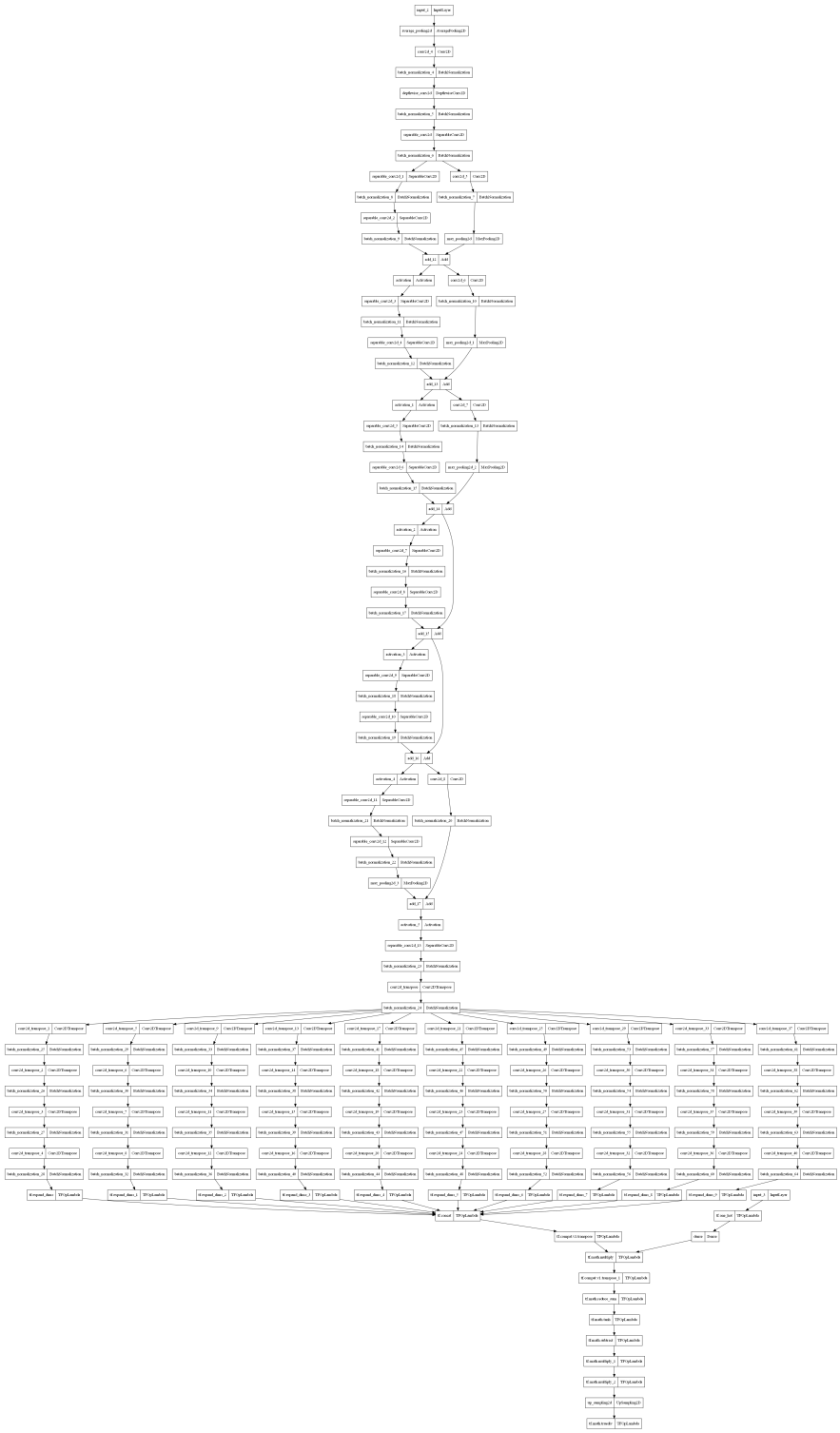

Figure A.2: Detailed diagram of the generator of G-EOT. This is the version with just 10 experts, as a picture with 50 experts would have been too wide to include in this document.

Appendix B

Credits for 3D models

Table B.1: Web links from where each 3D model in the dataset was downloaded.

Name	URL
barrel	https://free3d.com/3d-model/barrel-7685.html
baseball	https://www.cgtrader.com/free-3d-models/sports/equipment/game-ready-worn-baseball-ball
camaro	https://free3d.com/3d-model/chevrolet-camaro-46813.html
crocodile	https://free3d.com/3d-model/crocodile-v1–688517.html
clownfish	https://sketchfab.com/3d-models/clownfish-47ba2679d91a4f14b3fc0bf8e3805af5
crocodile	https://free3d.com/3d-model/crocodile-v1–688517.html
german shepherd	https://www.cgtrader.com/free-3d-models/animals/other/german-shepherd-dog-203601c4-4783-40e5-ab32-afa14e042dd5
jeep	https://www.cgtrader.com/free-3d-models/car/suv/uaz-3d
orange	https://free3d.com/3d-model/-orange–470103.html
orca	https://www.cgtrader.com/items/3331128/download-page
purse	https://www.cgtrader.com/free-3d-models/household/other/small-bag-wallet
rugby ball	https://free3d.com/3d-model/rugby-ball-v1–585369.html
running shoe	https://app.gazebosim.org/GoogleResearch/fuel/models/ASICS_GELBlur33_20_GS_BlackWhiteSafety_Orange
turtle	https://free3d.com/3d-model/sea-turtle-v1–175922.html
taxi	https://free3d.com/3d-model/taxi-v2–375169.html
teddy	https://free3d.com/3d-model/teddy-bear-sg-v1–215992.html

Appendix C

Links to Jupyter notebooks, adversarial textures and evaluation images

The Jupyter Notebook used to evaluate the 50 adversarial textures created by EOT, with the full results, can be found at https://github.com/Alexandru-Dascalu/adversarial-3d/blob/master/evaluation.ipynb . It includes the TFR and classification accuracy for each adversarial texture.

The rendered images of the adversarial 3D objects used to evaluate the adversarial textures, created by the above Jupyter notebook, can be found at https://drive.google.com/drive/folders/1IndIDIvWwXQ2mo1EXpx2LLf8m0QEfJwS?usp=sharing . The "adv" folder contains the images with the adversarial texture, while the "normal" folder has the equivalent images with the original texture. Inside both of those folders, there is a folder for each 3D model. Inside each of those folders, the photos are organised into 5 sub-folders, one for each target label.

The Jupyter notebook used to create the 50 adversarial textures and which contains the plots of the loss and TFR history during the optimisation process can be found at https://github.com/Alexandru-Dascalu/adversarial-3d/blob/master/experiments.ipynb .

The 50 adversarial textures created in the experiment in section 4.2 can be found at https://drive.google.com/drive/folders/16zr4nH81PVdYs231k5-qpgMP1Eg6-H2x?usp=sharing . In the file name of each texture, the first number if the target label for which that texture was made, and the second number is the number of optimisation steps that

took to create that texture.

The training history of the version of G-EOT with 50 experts can be seen in the Jupyter notebook at https://github.com/Alexandru-Dascalu/G-EOT/blob/master/experiment.ipynb .