# Final Project : Backdoor Attack on Diffusion Model

**Student : Gabriel Pecoraro**

**Hawk ID : A20593087**

**Due on 12/04/2024**

**Class : Secure Design for Machine Learning**

**College : Illinois Institute of Technology**

*Made by Gabriel Pecoraro*

# 1 Introduction

## 1.1 Background

Diffusion models represent a state-of-the-art approach in generative machine learning, leveraging deep neural networks to generate data by learning how to iteratively add and remove noise. Introduced in 2020, these models have quickly gained traction due to their ability to produce high-quality content across a range of domains, including image generation and other creative applications. Stable Diffusion, a prominent implementation, has become a key player in this field, showcasing the impressive capabilities of these models.

## 1.2 Motivation

With a constant need of more data to train new models more efficiently, diffusion models have emerged as a powerful generative modeling approach, driven by their ability to produce high-quality and diverse data. With improved training stability and strong theoretical foundations, they excel in applications ranging from image synthesis to super-resolution. Their success is fueled by advances in computational resources, making them versatile tools for creative and scientific tasks. Currently, diffusion models are cutting-edge technology in AI, with practical impact across various domains.

Figure 1: Generated Images using Denoising Diffusion Probabilistic Models

## 1.3 Review and Existing Solution

The growing adoption of diffusion models has introduced new security concerns, including the risk of backdoor attacks. Indeed, as they rely on generating new data, the training dataset must be protected. If a

Unlike traditional backdoor attacks on classifiers, which typically target classification outputs, backdoor attacks on diffusion models require tampering with the forward and reverse processes. This introduces significant challenges, as the attack must maintain high utility—ensuring the model continues to perform well—and high specificity—ensuring the backdoor triggers precisely without unintended side effects.

Recent studies have explored these vulnerabilities, highlighting the need for robust defenses. The potential risks include unauthorized access to training data, the generation of harmful content, and misuse of computational resources.

As diffusion models become increasingly integral to modern machine learning applications, understanding and mitigating these risks are critical.

Nonetheless, there are not many studies or papers which cope with implementing a backdoor attack on a diffusion model in general. This is a consequence of the novelty of the model and the difficulty - involving a consequent amount of computer resources - to implement a backdoor attack.

# 2 High-Level Description

## 2.1 BackDoor Attack

The whole goal of the project is to perform a sharp and stealthy attack, that would not be detectable by the user. The backdoor attack is provoked by using a so called trigger. The trigger is an intentionally inputted anomaly on the image which aims to confuse the model as it cannot understand the image. This trigger is implemented on a small part of the training dataset. The amount of data that have to be corrupted is defined by a poison rate and the process is called data poisoning. The smaller the poison rate is the stealthier the attack will be.

The main goal is to create a stealth attack that is able to confuse the model without raising user's awareness. However, it is important to poison enough data so the model can learn on those compromised data.

Henceforth, there is a trade-off to determine between stealth and the poison rate. The corrupted model must have similar behavior and performances as the clean model so it render it undetectable.

In addition to the data poisoning, a label corruption is implemented for the corrupted data. Thus, it generates misclassifications. An image of the corrupted trigger and label:

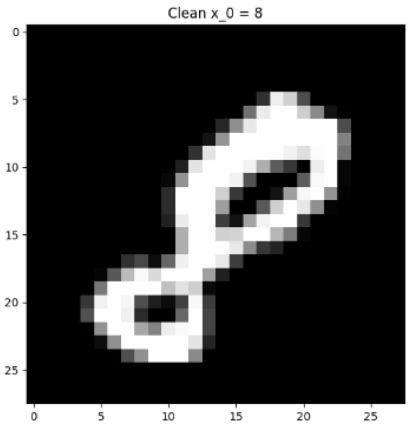


Figure 2: Forward Pass on the clean training data

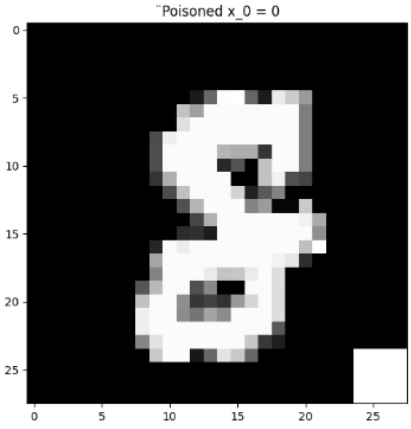


Figure 3: Forward Pass on the poisoned data

It can be seen that the backdoor attack is made on the forward process by noising an 8-digit considered as a zero by the system.

It is paramount to emphasize that a diffusion model backdoor attack is implemented only on the training dataset as there is no notion of testing dataset. Indeed, while for a classification task, a testing dataset is required to monitor if the model did not overfit, for diffusion model there is no such notion. The main goal is to only generate new instances using training dataset.

## 2.2 Diffusion Model

The underlying process of diffusion models consists of two main phases: a forward process, where Gaussian noise is added to data, and a reverse process, where this noise is removed. The reverse process is particularly significant, as it requires the model to reconstruct data from noisy inputs effectively. There are many possibilities to build this process but this architecture will use a classical encoder-decoder architecture. The architecture is below :

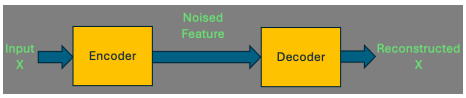


Figure 4: Encoder Decoder Architecture

### 2.2.1 Forward Pass

The forward pass is the encoding part where a feature are extracted from an input image. The forward pass in a diffusion model is the process where structured data is systematically degraded by adding noise, effectively transitioning it from a clear and coherent state to a completely noisy state. This degradation is modeled as a Markov process, wherein each step introduces a small amount of Gaussian noise, gradually corrupting the data over multiple iterations.

Algorithmically, this process can be described as:

**Algorithm 1** Forward Process in Diffusion Model

**Input:** Initial data sample $x_0$, noise schedule $\{\beta_t\}_{t=1}^{T}$
**Output:** Sequence of noisy samples $\{x_t\}_{t=1}^{T}$
Initialize $x_{t-1} \leftarrow x_0$
Initialize empty list $x_{\text{ts}} \leftarrow []$
**for** $t = 1$ to $T$ **do**
    Sample noise $\epsilon \sim \mathcal{N}(0, I)$
    Retrieve noise variance $\beta_t$
    Compute $x_t \leftarrow \sqrt{1-\beta_t} \cdot x_{t-1} + \sqrt{\beta_t} \cdot \epsilon$
    Append $x_t$ to $x_{\text{ts}}$
    Update $x_{t-1} \leftarrow x_t$
**end for**
**return** $x_{\text{ts}}$

where $x_t$ represents the noisy data at step $t$, $\beta_t$ is a noise schedule parameter controlling the variance at each step, and $\mathcal{N}$ denotes a Gaussian distribution. After implementing the forward pass the results are the following :

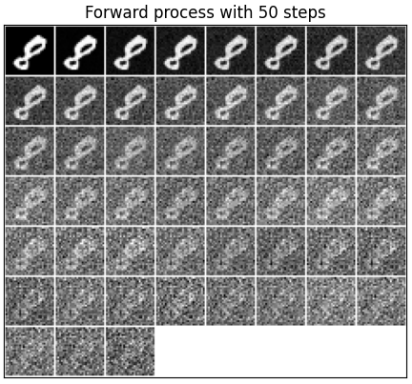


Figure 5: Forward Pass on the clean training data

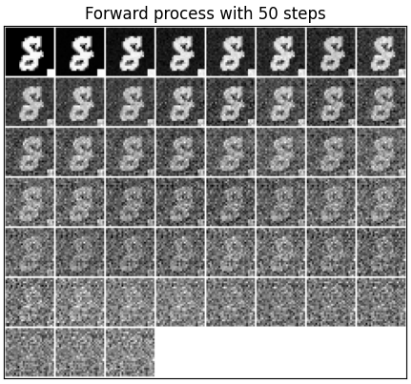


Figure 6: Forward Pass on the poisoned data

### 2.2.2 Backward Pass

The backward pass is the decoding part where the input is reconstructed from the noisy images. The backward pass in a diffusion model is the critical process of reconstructing coherent data from noisy inputs. It inverts the forward process, sequentially removing noise added during the forward pass to recover the underlying data structure. This step is central to the generative capability of the diffusion model, as it transforms random noise into meaningful, high-quality outputs. It will regenerate new data by sampling feature in the noise data :

**Algorithm 2** Sampling

1: $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\quad \mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = 0$
4: $\quad \mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \epsilon_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
5: **end for**
6: **return** $\mathbf{x}_0$

In the sampling algorithm, $\mathbf{x}_T$ is the initial noisy sample from $\mathcal{N}(0, \mathbf{I})$, and $\mathbf{x}_t$ and $\mathbf{x}_{t-1}$ represent the noisy and denoised samples at steps $t$ and $t-1$, respectively. The parameters $\alpha_t$ and $\bar{\alpha}_t$ control the noise schedule, while $\epsilon_\theta(\mathbf{x}_t, t)$ predicts the noise, and $\sigma_t$ adjusts the variance. $\mathbf{z}$ is Gaussian noise added only when $t > 1$.

This sampling model relies on a PixelCNN. A PixelCNN is a generative model designed to model the distribution of images by sequentially predicting

pixels. It uses a convolutional neural network (CNN) to estimate the conditional probability of each pixel given the previous ones, processing the image in a raster-scan order (left to right, top to bottom). By incorporating masked convolutions, PixelCNN ensures that the prediction of a pixel does not depend on future ones. The conditional probability network uses the chain rule, it decomposes the input image into a $1D$ distribution. In order to generate an image, it takes a random state in this distribution by sampling this latter and starts generating from the top corner pixel value.

# 3 Method and Miscellaneous

## 3.1 Running the code

The code can be run using Google Colab. It is sequential, only the PixelCNN model relies on python classes but it does not hinder the running phase.

## 3.2 GPU recommendation issue

### 3.2.1 GPU running

Using a GPU, perhaps even two, is strongly required in order to run the code. Otherwise, one may expect to encounter long time processing which may go to more than twelve hours depending of the definition of the hyper parameters.

### 3.2.2 Error:"CUDA ERROR MEMORY" hazard

When running the model through the clean dataset, no issue should be encountered. If it is the case, it might mean that the hyper parameters are set to high. Furthermore, when retrieving the clean training dataset model weights, no problem should be encountered and the diffusion process should be displayed.

However, when running the model on the poisoned dataset, one may encounter a memory issue. Indeed, if sequential process on both of the models is performed, the GPU might run out of memory. That is why the following function can be used :

```
!pip install GPUtil

import torch
from GPUtil import showUtilization as gpu_usage
from numba import cuda

def free_gpu_cache():
    print("Initial GPU Usage")
    gpu_usage()

    torch.cuda.empty_cache()
```

```
cuda.select_device(0)
cuda.close()
cuda.select_device(0)

print("GPU Usage after emptying the cache")
gpu_usage()
```

This latter may be unlocked to free some memory in the GPU allowing to execute the code.

Subsequently, an error that I have been raised and I could not deal with is the following :

If putting the boolean weight only to `True`, another error is encountered which is the following :


Figure 7: Error Raised

However, if the boolean is `False`, this error is raised :

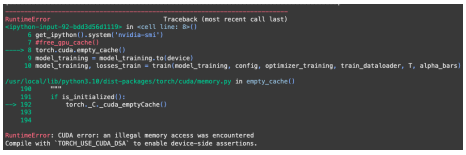


Figure 8: Error Raised

Honestly, after hours of browsing and digging on GitHub, Stack Overflow and so forth, yet, I have no clue why this error is raised. The only thing that I am aware is that this is a security from the Pytorch library which thinks that the weights created for the poisoned dataset may be compromised and harmful for the library. The GitHub page of the developer update about that is given following this link.

I kindly ask if anyone has any explanation to email me. I am eager to know.

# 4 Conducted Test

## 4.1 Setup

The tests were conducted on the classic MNIST dataset that gather a consequent amount of hand-written digits. It is a convenient dataset as it is integrated in the Pytorch library and the images are easy to manipulate.

Moreover, the hyper parameters where set at 2 epochs, a logging step of 1000 and a batch size of 4. Those value may be fine tunable or not due to the resource memory burden.

Finally, the test where processed using a GPU on Google Colab which allowed to speed up the process.

It is important to emphasize that due to the error raised, the test where dramatically hindered.

## 4.2 First Results

The first test was performed on a clean dataset and on a poisoned one with a poisoning rate of 0.2. They are trained on the same model with the same parameters than for the clean training dataset. The training results for both of the model are the following :

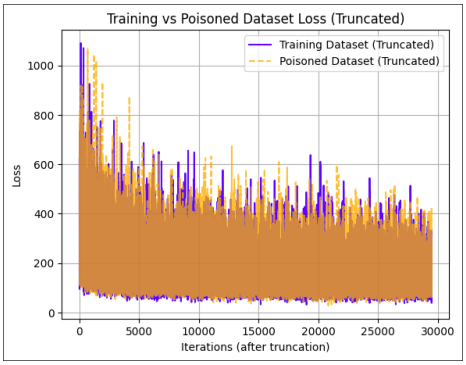


Figure 9: Loss in the model between clean and poisoned dataset

Overall, it can be seen on this plot that the poisoned dataset and the clean dataset yield the same performances concerning reconstructing new instances from noise. At first glance, it is conclusive as it shows that the backdoor attack is stealth but also sharp in the meanwhile as the poison rate is high.

However, it may also mean that the attack is not powerful enough and that the model did not take into consideration the backdoor attack. Unfortunately, no further tests had been performed due to the limitations caused by the error.

Along with this, the model generated some new instances for the clean dataset and the poisoned dataset. It also tried to find the most possible match for each dataset. Results are the following :

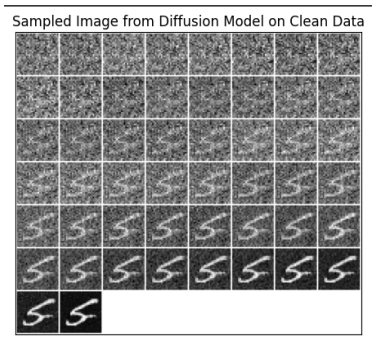


Figure 10: Generating Instance on the clean training data

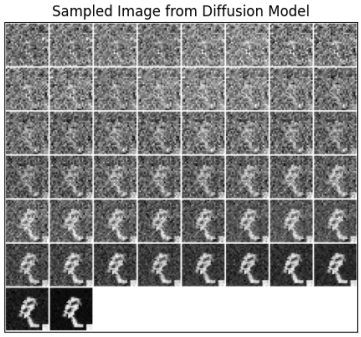


Figure 11: Generating Instance on the poisoned data

It is notable that for the clean dataset, the model generated a perfect five. Yet, more sampled have to be observed to conclude about the effectiveness of the model.

Moreover, for the poisoned dataset, it recreated a semblance of a two but it is important to point out that it also generates the trigger on the new data. It means that the diffusion model is able to reproduce the discrepancy that a user inputted. For example, let's say that a instead of the blank square, one would have inputted a harmful content, the model would have recreated it. Hence, it can be concluded that the backdoor attack somehow functioned. Nevertheless, more results ought to be seen.

The model designed also look in each dataset the potential most matching digit. Here are the results :

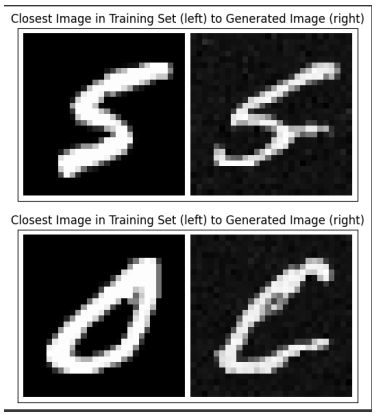


Figure 12: Generating New Instance From Noise on the Clean Training Data

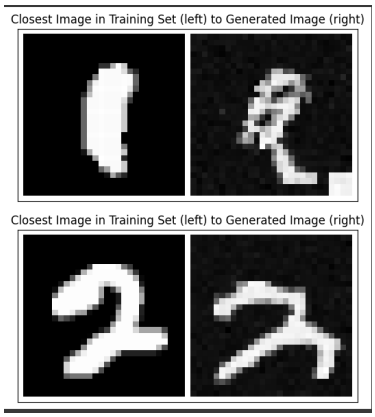


Figure 13: Generating New Instance on the Poisoned Data

For the training dataset, although the model struggles to classify an approx-

imated zero which could also be a six correctly assesses a produced 5 digit to an existing one.

As for the poisoned dataset, the classification also yields to moderatly conclusive results. More test ought to be perform in order to find the good set of hyper parameters.

# 5 Conclusion

## 5.1 Achieved Objectives

Overall, the objective to create an almost working diffusion model and implementing on this latter a backdoor attack are almost achieved. It is important to highlights that with the GPU available, creating a working diffusion model is conclusive.

Although the objective of implementing a backdoor attack on various poison rate is not achieved, the objectives stated in the presentation concerning the implementation of a backdoor attack on the model are achieved.

However, the extension of the diffusion on the CIFAR-10 had not been achieved. f

## 5.2 Future Work

As stated before, an interesting part to explore might be the implementation of the diffusion model and the associated backdoor attack on the dataset CIFAR-10. It may be interesting to see what kind of content the diffusion model may generate while it is compromised on real life photo and not only on hand-written digits.

A more achievable task may be also to modify the sampling training model in order to resolve the Pytorch and Cuda error

To sum up, after I dealt with diffusion models encompassing their logic to building their architecture, I understand now why they are so popular. Generating new instances using a machine is a huge modern challenge and Diffusion Models excels at it.

# References


[1] Architecture of DM, *Building Diffusion Model From Scratch*, Available at: `https://michaelwornow.net/2023/07/01/diffusion-models-from-scratch`, Accessed: December 2, 2024.

[2] First DM proposal proposed by Ho and al. in 2020, *Denoising Diffusion Probabilistic Models*, Available at: `https://arxiv.org/pdf/2006.11239`, Accessed: December 1, 2024.

[3] PixelRNN Architecture, *Pixel Recurrent Neural Networks*, Available at: https://arxiv.org/pdf/1601.06759, Accessed: December 3, 2024.

[4] Backdoor on DM, *How to Backdoor Diffusion Models?*, Available at: https://arxiv.org/pdf/2212.05400, Accessed: November 28, 2024.

[5] Pytorch Error, *Security on Pytorch*, Available at: https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models, Accessed: November 4, 2024.