M2 Data Science

Apprentissage Statistique

Rapport de Projet
Réseaux de Neurones Artificiels

Erivan INAN

Professeure : Annick VALIBOUZE

Un rapport soumis en vue de l'accomplissement des exigences de
l'ISUP et Sorbonne Université pour l'obtention du diplôme de
Master 2 Data Science

26 janvier 2025

Abstract

Ce projet explore l'application de modèles d'apprentissage profond pour la détection d'anomalies dans des données variées, telles que les comprimés et gélules (MVTec Pill), les chiffres manuscrits (MNIST) et les données de sécurité réseau (CICIDS2017). Différentes approches ont été mises en œuvre, incluant les perceptrons multicouches (PMC), les réseaux antagonistes génératifs (GAN), les cartes de Kohonen (SOM) et les réseaux de croyances profonds (DBN). Les résultats démontrent la pertinence et les limites de chaque méthode selon le type de données et le contexte d'application. Ces travaux offrent une vision comparative des capacités des modèles de deep learning pour résoudre des problématiques critiques dans des domaines tels que la détection de défauts industriels, l'analyse de données manuscrites et la cybersécurité.

Keywords : Anomaly Detection, Generative Adversarial Network, Multi-Layer Perceptron, Deep Learning

Table des matières

Chapitre 1

Introduction aux RNA (Item 1)

RNA : Réseaux Neurones Artificiels

1.1 Réseaux de neurones

Les réseaux de neurones artificiels sont des systèmes inspirés du fonctionnement biologique des neurones. On les représente par des graphes orientés pondérés. Chaque noeud du graphe est représenté par un neurone. Les neurones formels sont interconnectés par des poids synaptiques. Les poids synaptiques du réseau sont stockés dans la matrice des poids, qui mesure la connectivité du réseau.

Ces réseaux fonctionnent sous deux modes :

- Mode de reconnaissance : Propagation de l'information pour produire une sortie à partir d'une entrée.
- Mode d'apprentissage : Ajustement des poids pour adapter le réseau à une tâche donnée.

Un réseau de neurone est composé de plusieurs couches : couches d'entrée, cachées et de sortie. Les couches communiquent de manière succinte via des fonctions, la fonction d'entrée h, les fonctions d'activation entre les couches cachées et la fonction de sortie.

Chaque neurone d'un réseau calcule une valeur intermédiaire, appelée activation pondérée ou potentiel (e_i) en combinant ses entrées avec des poids synaptiques. Elle peut être linéaire, ou affine si un biais Φ est présent. En voici son expression : $h(i) = \sum_{j=1}^{N} w_{i,j}.a_j - \Phi_i$.
Cette fonction détermine le signal que le neurone reçoit avant d'appliquer la fonction d'activation.

La fonction d'activation transforme l'activation pondérée e_i en une sortie a_i, qui sera transmise aux neurones suivants. Parmi les fonctions d'activation populaires, on retrouve ReLU, sigmoïde ou encore la tangente hyperbolique.

La fonction de sortie transforme quant à elle les activations finales en une forme adaptée à la tâche du réseau. Il en existe plusieurs, dont `softmax` pour la classification multi-catégorielle, la fonction linéaire pour la régression ou encore la sigmoïde pour les tâches binaires.

1.2 Introduction au Perceptron Multi-Couches (PMC)

Le Perceptron Multi-Couches (PMC) est un cas particulier de réseau neuronal avec plusieurs couches : une couche d'entrée, une ou plusieurs couches cachées, une couche de sortie. Chaque neurone effectue un calcul pondéré basé sur ses entrées, applique une fonction d'activation, et transmet le résultat aux neurones suivants.

1.2.1 Apprentissage supervisé du PMC

L'apprentissage supervisé consiste à fournir au réseau un ensemble de données étiquetées (couples entrée-sortie). Les étapes clés de l'apprentissage supervisé du PMC sont les suivantes :

- Propagation avant : L'entrée x est propagée à travers les couches pour produire une sortie. L'activation $h(i)$ est calculée pour chaque neurone i puis $h(i)$ est transformée en une sortie pour chaque neurone $a_i = F(h(i))$.
- Calcul de l'erreur : La différence entre la sortie réelle a_i et la sortie attendue d_i est mesurée par une fonction d'erreur, souvent l'erreur quadratique : $E(a, d) = \frac{1}{2}\sum_{i=1}^{m}(a_i - d_i)^2$, où m est le nombre de neurones en sortie.
- Rétropropagation de l'erreur : L'erreur est propagée en sens inverse pour ajuster les poids synaptiques. La backpropagation repose sur la descente du gradient, en utilisant la règle de la chaîne à commencer par les couches de sortie pour remonter vers les couches cachées : $\Delta w_{i,j} = -\lambda \frac{\delta E}{\delta w_{i,j}}$, où λ est le pas d'apprentissage.

Ce processus propagation avant $\rightarrow$ calcul de l'erreur $\rightarrow$ rétropropagation est répété sur plusieurs cycles (epochs) jusqu'à minimisation de l'erreur.

Le PMC est utilisé pour la classification, la reconnaissance de formes, et la prédiction dans des domaines variés comme la finance, la santé ou la reconnaissance d'images. De plus, les variantes modernes incluent des réseaux profonds (Deep Learning) pour traiter des données complexes.

Chapitre 2

Perceptron Multi-Couches (Item 2)

Objectif

Dans cette section, je vais explorer deux méthodes pour détecter des anomalies dans le dataset MNIST : un modèle Perceptron Multi-Couches (PMC) et la distance de Mahalanobis. Le but est de comparer les performances des deux approches et d'analyser leurs résultats en termes d'efficacité et d'interprétation.

2.1 Jeu de données

L'utilisation du dataset **MNIST (Modified National Institute of Standards and Technology)** s'est avérée très utile pour un premier test du PMC. En effet, il s'agit de l'un des jeux de données les plus connus en apprentissage automatique, particulièrement utilisé pour des tâches de classification d'images. Il contient 70 000 images de chiffres manuscrits, réparties en 60 000 exemples d'entraînement et 10 000 exemples de test. Chaque image est en niveaux de gris, de dimension 28x28 pixels, et représente un chiffre de 0 à 9.

Librairies utilisées

```
# Configuration de l'environnement (pour importer Keras à l'environnement R)
library(reticulate)
use_condaenv("nom_de_lenv_contenant_tensorflow", conda = "path/conda.bat")
py_config()

# Import des librairies utiles
library(ggplot2)
library(reticulate)
library(pROC)
library(caret)
suppressPackageStartupMessages(library(caret))
library(keras)
```

Data preprocessing

```
# Charger le dataset MNIST
data <- dataset_mnist()
x_train <- data$train$x
y_train <- data$train$y
x_test <- data$test$x
y_test <- data$test$y
```

```
# Prétraitement des données
x_train <- array_reshape(x_train, c(nrow(x_train), 28 * 28)) / 255
x_test <- array_reshape(x_test, c(nrow(x_test), 28 * 28)) / 255
y_train <- as.numeric(y_train)
y_test <- as.numeric(y_test)
```

Ensuite, j'ai défini les images de chiffres "0" comme normales, et les autres images de chiffres différents de 0 comme anomalies. Ce choix pour illustrer le fait que la détection d'anomalies peut simplement s'exprimer comme une classification binaire : la classe 0 et la classe 1, 2, 3, 4, 5, 6, 7, 8, 9, et que les classes identifiées comme anomalies ne sont pas toujours intuitives.

```
# Séparation des données normales et anormales
norm_class <- 0 # les 0 sont normaux
x_train_norm <- x_train[y_train == norm_class, ]
x_test_norm <- x_test[y_test == norm_class, ]
x_test_ano <- x_test[y_test != norm_class, ]
```

2.2 Implémentation PMC sur R

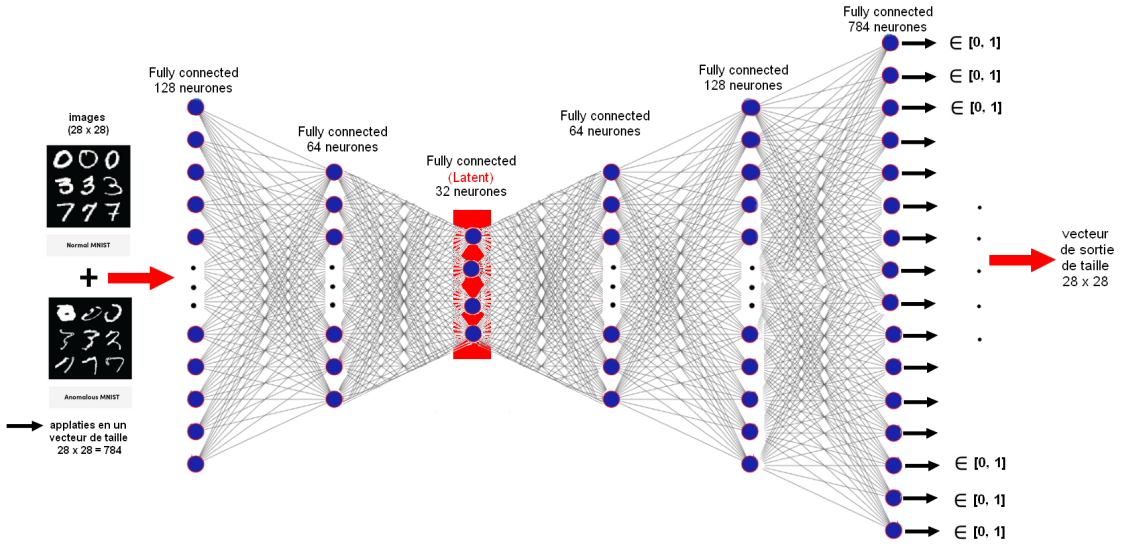
Structure du Modèle

- **Couches entièrement connectées** : Le PMC contient plusieurs couches denses entièrement connectées ($128 \rightarrow 64 \rightarrow 32 \rightarrow 64 \rightarrow 128$).
- **Fonctions d'activation** : ReLU, idéale pour capturer la non-linéarité, ainsi que Sigmoid en sortie pour normaliser les reconstructions.
- **Latent Layer** : Une couche centrale, représntée en rouge, avec 32 unités pour encoder les caractéristiques latentes des images normales.

Voici l'implémentation du PMC sur R :

```
model <- keras_model_sequential()
  layer_dense(units = 128, activation = 'relu', input_shape = c(28 * 28))
  layer_dense(units = 64, activation = 'relu')
  layer_dense(units = 32, activation = 'relu', name = "latent")
  layer_dense(units = 64, activation = 'relu')
  layer_dense(units = 128, activation = 'relu')
  layer_dense(units = 28 * 28, activation = 'sigmoid')

model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Ci-dessous un schéma du réseau :



J'ai designé l'image ci-dessus à partir de celle-ci : https ://www.researchgate.net/figure/Neural-network-architecture-illustrating-the-layers-and-connections-used-for-shade_fig2_376695090.

Le modèle, basé sur un réseau dense entièrement connecté, est entraîné à reconstruire uniquement les données normales en minimisant l'erreur quadratique moyenne :

```
# Construction du modèle MLP pour encoder les données normales
model <- keras_model_sequential(),
  layer_dense(units = 128, activation = 'relu', input_shape = c(28 * 28)),
  layer_dense(units = 64, activation = 'relu'),
  layer_dense(units = 32, activation = 'relu', name = "latent"),
  layer_dense(units = 64, activation = 'relu'),
  layer_dense(units = 128, activation = 'relu'),
  layer_dense(units = 28 * 28, activation = 'sigmoid')

model.compile(optimizer = 'adam', loss = 'mean_squared_error')
```

Entraînement

- Données utilisées : uniquement les données correspondant à la classe normale (chiffre 0).
- Fonction de perte : Erreur quadratique moyenne (mean_squared_error).
- Optimiseur : adam, c'est l'optimiseur le plus performant d'après mes tests avec la déscente de gradient et adagrad

```
history <- model.fit(x_train_normal, x_train_normal,
  epochs = 30, batch_size = 256, validation_split = 0.2)
```

On peut observer l'évolution de la perte au fil des epochs :

```
Epoch 1/30
19/19 [==============================] - 4s 134ms/step - loss: 0.1527 - val_loss: 0.0721
Epoch 2/30
19/19 [==============================] - 0s 26ms/step - loss: 0.0684 - val_loss: 0.0631
Epoch 3/30
19/19 [==============================] - 1s 32ms/step - loss: 0.0606 - val_loss: 0.0547
Epoch 4/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0514 - val_loss: 0.0458
Epoch 5/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0446 - val_loss: 0.0418
Epoch 6/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0410 - val_loss: 0.0381
Epoch 7/30
19/19 [==============================] - 1s 30ms/step - loss: 0.0371 - val_loss: 0.0346
Epoch 8/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0339 - val_loss: 0.0323
Epoch 9/30
19/19 [==============================] - 1s 30ms/step - loss: 0.0317 - val_loss: 0.0304
Epoch 10/30
19/19 [==============================] - 1s 31ms/step - loss: 0.0297 - val_loss: 0.0287
Epoch 11/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0280 - val_loss: 0.0270
Epoch 12/30
19/19 [==============================] - 0s 27ms/step - loss: 0.0266 - val_loss: 0.0258
Epoch 13/30
19/19 [==============================] - 0s 24ms/step - loss: 0.0253 - val_loss: 0.0245
Epoch 14/30
19/19 [==============================] - 0s 25ms/step - loss: 0.0241 - val_loss: 0.0234
Epoch 15/30
19/19 [==============================] - 0s 25ms/step - loss: 0.0231 - val_loss: 0.0226
Epoch 16/30
19/19 [==============================] - 0s 22ms/step - loss: 0.0222 - val_loss: 0.0218
Epoch 17/30
19/19 [==============================] - 0s 24ms/step - loss: 0.0214 - val_loss: 0.0211
Epoch 18/30
19/19 [==============================] - 0s 23ms/step - loss: 0.0207 - val_loss: 0.0208
Epoch 19/30
19/19 [==============================] - 0s 25ms/step - loss: 0.0202 - val_loss: 0.0201
Epoch 20/30
19/19 [==============================] - 1s 29ms/step - loss: 0.0197 - val_loss: 0.0196
Epoch 21/30
19/19 [==============================] - 0s 20ms/step - loss: 0.0192 - val_loss: 0.0195
Epoch 22/30
19/19 [==============================] - 0s 24ms/step - loss: 0.0188 - val_loss: 0.0191
Epoch 23/30
19/19 [==============================] - 0s 24ms/step - loss: 0.0185 - val_loss: 0.0186
Epoch 24/30
19/19 [==============================] - 0s 23ms/step - loss: 0.0181 - val_loss: 0.0181
Epoch 25/30
19/19 [==============================] - 1s 30ms/step - loss: 0.0176 - val_loss: 0.0177
Epoch 26/30
19/19 [==============================] - 0s 22ms/step - loss: 0.0172 - val_loss: 0.0174
Epoch 27/30
19/19 [==============================] - 0s 23ms/step - loss: 0.0170 - val_loss: 0.0170
Epoch 28/30
19/19 [==============================] - 0s 24ms/step - loss: 0.0166 - val_loss: 0.0168
Epoch 29/30
19/19 [==============================] - 0s 25ms/step - loss: 0.0164 - val_loss: 0.0166
Epoch 30/30
19/19 [==============================] - 0s 27ms/step - loss: 0.0161 - val_loss: 0.0162
```

Figure 2.1 : Évolution des pertes au fil des épochs

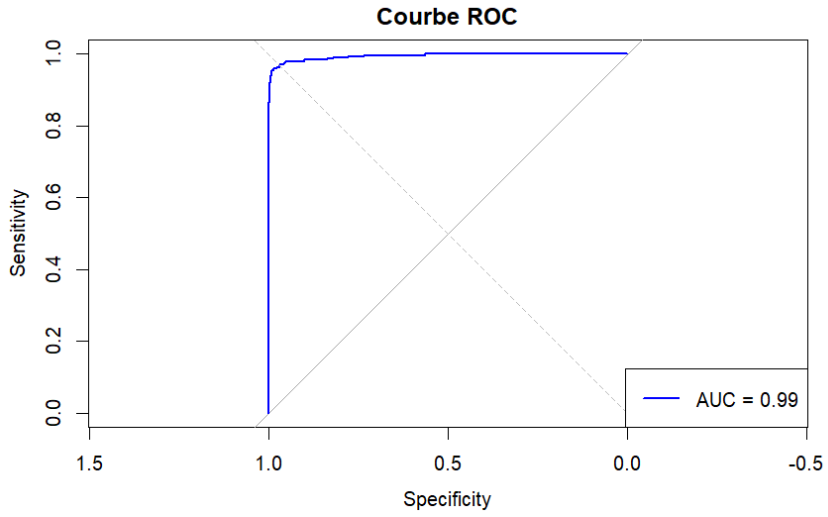
Évaluation Une fois entraîné, les erreurs de reconstruction sont calculées pour les images normales et anormales. Fixer le 95e percentile des erreurs normales comme seuil est raisonnable.

Le code pour le calcul des erreurs de reconstruction provient de la source suivante : https ://medium.com/@weidagang/demystifying-anomaly-detection-with-autoencoder-neural-networks-1e235840d879, que j'ai adapté à mon contexte.

```
# Erreur de reconstruction pour les données normales et anormales
reconstruction_error <- function(model, data) {
  reconstructed <- model.predict(data)
  errors <- rowSums((data - reconstructed)^2)
  return(errors)}
normal_errors <- reconstruction_error(model, x_test_normal)
anomaly_errors <- reconstruction_error(model, x_test_anomaly)

# Définir un seuil pour détecter les anomalies
threshold <- quantile(normal_errors, 0.95)

# Identifier les anomalies
is_anomaly <- function(errors, threshold) {return(errors > threshold)}

normal_anomalies <- is_anomaly(normal_errors, threshold)
anomaly_detected <- is_anomaly(anomaly_errors, threshold)

# Résultats
cat("Seuil d'anomalie:", threshold, "\n")
cat("Taux de détection d'anomalies parmi les données normales:", mean(normal_anomalies))
cat("Taux de détection d'anomalies parmi les anomalies:", mean(anomaly_detected), "\n")
```

Les résultats montrent un faible taux de fausses alertes (5 %) et un excellent taux de détection des anomalies (99 %), confirmé par une AUC de 0,994 :

```
31/31 [==============================] - 0s 6ms/step
282/282 [==============================] - 1s 3ms/step
Seuil d'anomalie: 24.32507
Taux de détection d'anomalies parmi les données normales: 0.05
Taux de détection d'anomalies parmi les anomalies: 0.9900222
```



Cela indique que le modèle est efficace pour séparer les données normales des anomalies, tout en maintenant une précision élevée. Le rapport de classification ci-dessous le confirme également.

```
Confusion Matrix and Statistics

          Reference
Prediction  0  1
         0 45 10
         1  5 40

               Accuracy : 0.85
                 95% CI : (0.7647, 0.9135)
    No Information Rate : 0.5
    P-Value [Acc > NIR] : 2.413e-13

                  Kappa : 0.7

 Mcnemar's Test P-Value : 0.3017

            Sensitivity : 0.9000
            Specificity : 0.8000
         Pos Pred Value : 0.8182
         Neg Pred Value : 0.8889
             Prevalence : 0.5000
         Detection Rate : 0.4500
   Detection Prevalence : 0.5500
      Balanced Accuracy : 0.8500

       'Positive' Class : 0


Rapport de Classification :
Précision : 0.82
Rappel : 0.9
F1-Score : 0.86
```

Figure 2.2 : Rapport de classification du PMC pour la détection d'anomalies

2.3 Distance de Mahalanobis

2.3.1 Principe

La distance de Mahalanobis mesure la dissimilarité d'un point par rapport à une distribution multivariée. Cette méthode repose sur l'hypothèse que les données normales suivent une distribution gaussienne.

Étapes

- Réduction dimensionnelle (PCA) pour réduire la corrélation entre les variables.
- Nombre de composantes retenues : 100.
- Calcul de la matrice de covariance des données normales, régularisée pour éviter les singularités.
- Définition d'un seuil basé sur le 99e percentile des distances calculées sur les données normales.

2.3.2 Implémentation

```
x_train <- mnist$train$x
x_test <- mnist$test$x

# Conversion images --> vecteurs
x_train_flat <- array_reshape(x_train, c(nrow(x_train), 28 * 28)) / 255
x_test_flat <- array_reshape(x_test, c(nrow(x_test), 28 * 28)) / 255

# Suppression des colonnes avec faible variance
keep <- apply(x_train_flat, 2, var) > 1e-6
x_train_flat <- x_train_flat[, keep]
x_test_flat <- x_test_flat[, keep]

# Réduction de dimension avec PCA
library(stats)
```

```
pca_result <- prcomp(x_train_flat, center = TRUE, scale. = TRUE)
num_components <- 100
x_train_pca <- pca_result$x[, 1:num_components]
x_test_pca <- predict(pca_result, newdata = x_test_flat)[, 1:num_components]

# Calcul de la matrice de covariance et son inverse avec regularisation
epsilon <- 1e-6  # petite valeur pour régulariser
cov_matrix <- cov(x_train_pca) + diag(epsilon, ncol(x_train_pca))
inv_cov_matrix <- solve(cov_matrix)

# Calcul de la moyenne des données d'entrainement
mean_vector <- colMeans(x_train_pca)

# Fonction pour calculer la distance de Mahalanobis
mahalanobis_distance <- function(x, mean, inv_cov) {
  diff <- x - mean
  sqrt(rowSums((diff %*% inv_cov) * diff))
}

# distances de Mahalanobis pour les données de test
distances <- mahalanobis_distance(x_test_pca, mean_vector, inv_cov_matrix)

# seuil basé sur le quantile des distances
threshold <- quantile(distances, 0.99)  # seuil au 99e percentile

# detection des anomalies basee sur le seuil "threshold"
anomalies <- distances > threshold
cat("Nombre d'anomalies détectées :", sum(anomalies), "\n")

# Visualisation d'un exemple d' anomalie
if (any(anomalies)) {
  idx <- which(anomalies)[1]  # index de la première anomalie
  image(matrix(x_test[idx, , ], 28, 28), col = gray.colors(256),
        main = "Anomalie détectée")}
```

D'abord, les images du dataset d'entraînement et de test sont aplaties et normalisées en divisant les valeurs par 255 pour obtenir des données entre 0 et 1.
Ensuite, les colonnes avec une variance quasi nulle sont éliminées pour réduire le bruit. Si nécessaire, une réduction de dimension supplémentaire est réalisée avec une analyse en composantes principales (PCA), en ne conservant que les 100 premières composantes principales. Une matrice de covariance est calculée à partir des données PCA, avec une régularisation pour éviter les problèmes numériques, et son inverse est obtenue.
La moyenne des données PCA d'entraînement est également calculée. À partir de ces informations, une fonction calcule la distance de Mahalanobis pour chaque point des données de test, mesurant à quel point un point diffère des données "normales" (entraînement). Un seuil basé sur le 99ème percentile des distances est défini pour identifier les anomalies.
Les points avec des distances supérieures à ce seuil sont marqués comme anomalies.

Résultats : Nombre d'anomalies détectées : 100

Les 100 anomalies détectées représentent seulement une petite partie des "0" qui sont les plus éloignés. En effet, si les chiffres "0" ne sont pas suffisamment éloignés du reste des données en termes de distance de Mahalanobis, la majorité d'entre eux ne seront pas détectés comme anomalies.
De plus, l'approche est simple mais peut être limitée si la distribution réelle des données diffère de la gaussienne.

Chapitre 3

Autoencodeur convolutionnel (CAE) avec Keras (Item 3)

Objectif

Dans cette section, j'implémente un réseau neuronal autre que `Nnet` de R, j'ai choisi un encodeur simple implémentable sous la librairie `Keras`. Contrairement à un réseau de neurones convolutionnel (CNN) classique, qui est principalement utilisé dans des tâches supervisées comme la classification, un autoencodeur convolutionnel (CAE) est un modèle non supervisé conçu pour encoder les données en une représentation de faible dimension dans la partie encodeur, puis reconstruire les données originales à partir de cette représentation dans la partie décodeur. Cette section était manquante dans mon rapport précédent, par confusion sur l'énoncé.

Jeu de données Tout comme la partie précédente sur les PMC, j'ai choisi d'utiliser le jeu de données MNIST à nouveau.

3.1 Implémentation

Librairies utilisées

```
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
```

Prétraitement Je commence par charger les données MNIST, puis séparer les données en ensembles d'entraînement et de test.

```
# Chargement des données MNIST
(data_entrainement, etiquettes_entrainement), (data_test, etiquettes_test) = mnist.load_data()

# Mise au format (batch, hauteur, largeur) et normalisation
data_entrainement = data_entrainement.astype('float32') / 255.0
data_test = data_test.astype('float32') / 255.0
data_entrainement = np.expand_dims(data_entrainement, axis=-1)
data_test = np.expand_dims(data_test, axis=-1)

# Je conserve la classe 0 comme normale, le reste comme anomalie
data_entrainement_normales = data_entrainement[etiquettes_entrainement == 0]
data_test_normales = data_test[etiquettes_test == 0]
data_test_anomalies = data_test[etiquettes_test != 0]
```

Construction de l'autoencodeur La fonction suivante construit l'autoencodeur constitué d'un encodeur pour réduire la dimensionnalité des données et d'un décodeur pour reconstruire les images.

```
def const_autoencodeur(taille_entree):
    encodeur = models.Sequential([
        layers.Input(shape=taille_entree),
        layers.Conv2D(32, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2), padding='same'),
        layers.Conv2D(64, (3, 3), activation='relu', padding='same'),
        layers.MaxPooling2D((2, 2), padding='same')
    ])

    decodeur = models.Sequential([
        layers.Conv2DTranspose(64, (3, 3), activation='relu', padding='same'),
        layers.UpSampling2D((2, 2)),
        layers.Conv2DTranspose(32, (3, 3), activation='relu', padding='same'),
        layers.UpSampling2D((2, 2)),
        layers.Conv2D(1, (3, 3), activation='sigmoid', padding='same')
    ])

    autoencodeur = models.Sequential([encodeur, decodeur])
    return autoencodeur

taille_entree = data_entrainement_normales.shape[1:]
autoencodeur = const_autoencodeur(taille_entree)
```

Puis j'ai compilé le modèle avec l'optimiseur adam et la fonction de perte MSE (erreur quadratique moyenne), avant de l'entraîner sur 20 epochs :

```
autoencodeur.compile(optimizer='adam', loss='mse')

# Entraîner l'autoencodeur uniquement sur les données normales
autoencodeur.fit(
    data_entrainement_normales, data_entrainement_normales,
    epochs=20,
    batch_size=128,
    validation_split=0.1,
    shuffle=True
)
```

Puis comme à la partie précédente, j'évalue le modèle à l'aide de la métrique d'erreur de reconstruction :

```
# Calcul de l'erreur de reconstruction pour définir un seuil
reconstructions = autoencodeur.predict(data_test_normales)
erreurs_mse = np.mean(np.power(data_test_normales - reconstructions, 2), axis=(1, 2, 3))
seuil = np.percentile(erreurs_mse, 95)
print(f"Seuil de détection d'anomalie : {seuil}")

# Évaluation sur la concatenation des données normales et anormales
reconstructions_test_normales = autoencodeur.predict(data_test_normales)
reconstructions_test_anomalies = autoencodeur.predict(data_test_anomalies)

mse_normales = np.mean(np.power(data_test_normales -
        reconstructions_test_normales, 2), axis=(1, 2, 3))
mse_anomalies = np.mean(np.power(data_test_anomalies -
        reconstructions_test_anomalies, 2), axis=(1, 2, 3))
```

```
# Prédictions basées sur le seuil
predictions_normales = mse_normales > seuil
predictions_anomalies = mse_anomalies > seuil

# Calcul de la précision
vrais_positifs = np.sum(predictions_anomalies)
faux_negatifs = len(predictions_anomalies) -
                vrais_positifs
vrais_negatifs = np.sum(~predictions_normales)
faux_positifs = len(predictions_normales) -
                vrais_negatifs

precision = (vrais_positifs + vrais_negatifs) / (len(predictions_normales) +
            len(predictions_anomalies))
print(f"Précision du modèle : {precision:.2f}")
```

Et on peut visualiser l'évolution des pertes au fil des épochs :

```
Epoch 1/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 7s 104ms/step - loss: 0.1297 - val_loss: 0.0182
Epoch 2/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 93ms/step - loss: 0.0150 - val_loss: 0.0094
Epoch 3/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 94ms/step - loss: 0.0088 - val_loss: 0.0071
Epoch 4/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 96ms/step - loss: 0.0071 - val_loss: 0.0063
Epoch 5/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 96ms/step - loss: 0.0064 - val_loss: 0.0061
Epoch 6/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 95ms/step - loss: 0.0058 - val_loss: 0.0053
Epoch 7/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 5s 107ms/step - loss: 0.0053 - val_loss: 0.0051
Epoch 8/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 98ms/step - loss: 0.0052 - val_loss: 0.0049
Epoch 9/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 98ms/step - loss: 0.0049 - val_loss: 0.0048
Epoch 10/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 4s 103ms/step - loss: 0.0046 - val_loss: 0.0045
Epoch 11/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 5s 121ms/step - loss: 0.0046 - val_loss: 0.0042
Epoch 12/20
42/42 ━━━━━━━━━━━━━━━━━━━━ 5s 110ms/step - loss: 0.0043 - val_loss: 0.0041
Epoch 13/20
...
Seuil de détection d'anomalie : 0.006099856924265623
31/31 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step
282/282 ━━━━━━━━━━━━━━━━━━━━ 4s 13ms/step
Précision du modèle : 0.40
```

Figure 3.1 : Évolution des pertes au fil des épochs

Les performances de l'autoencodeur montrent une convergence rapide au cours de l'entraînement, avec une diminution significative de la perte (loss) sur les données d'entraînement et de validation (val_loss). Dès la première époque, la perte chute de 0.12 à 0.01 sur la validation, ce qui indique que le modèle apprend efficacement à reconstruire les images normales.

Cependant, malgré cette performance d'entraînement, la précision globale sur les données de test atteint seulement 40%, certainement dû au seuil de détection ou à l'architecture du modèle, qui pourrait être ajustée.

Chapitre 4

Réseaux Antagonistes Génératifs (Items 5, 6)

4.1 Présentation des Réseaux Antagonistes Génératifs (Item 5)

Les Generative Adversarial Networks (GAN), introduits en 2014 par Ian Goodfellow et al, sont une famille de modèles d'apprentissage profond conçus pour générer de nouvelles données réalistes à partir d'un bruit aléatoire. Les GAN reposent sur une architecture à double réseau : un générateur et un discriminateur, qui s'entraînent de manière adversariale :

- Le **générateur** : Son objectif est de créer des données synthétiques (par exemple, des images) qui ressemblent le plus possible aux données réelles. Il prend comme entrée un vecteur de bruit aléatoire issu d'une distribution gaussienne ou uniforme et le transforme en une donnée générée.
- Le **discriminateur** : Ce réseau est un classificateur binaire. Il apprend à distinguer les données générées par le générateur des données réelles issues du dataset d'entraînement. Son rôle est de "critiquer" les données produites par le générateur.

Le processus d'apprentissage repose sur un jeu minimax entre les deux réseaux :

- Le générateur tente de tromper le discriminateur en produisant des données réalistes.
- Le discriminateur cherche à détecter les faux parmi les vrais.
- L'objectif global est d'atteindre un équilibre où le générateur produit des données si convaincantes que le discriminateur ne peut plus les différencier des vraies données (c'est-à-dire une probabilité de 50 % pour chaque décision du discriminateur).

4.2 Démonstration des GAN

4.2.1 Objectif

Dans cette section, un réseau GAN a été implémenté pour résoudre un problème de détection d'anomalies dans des images industrielles de comprimés et gélules.

4.2.2 Jeu de données

Pour l'implémentation du GAN, j'ai choisi le dataset **MVTec Anomaly Detection**, consultable et téléchargeable via l'hyperlien. Ce dataset est largement utilisé pour évaluer les performances des modèles de détection d'anomalies. Créé pour des tâches de vision par ordinateur, il contient des images haute résolution de divers objets et textures industriels. Chaque catégorie comprend des données normales et des anomalies de différentes natures, telles que des rayures, des fissures, des

contaminations ou des défauts structurels. Ce dataset est particulièrement utile pour entraîner et évaluer des modèles non supervisés de détection d'anomalies.

Voici l'arborescence du dataset Pill :

```
pill/
├── ground_truth/
│   ├── color (24 images)
│   ├── combined (16 images)
│   ├── contamination (20 images)
│   ├── crack (25 images)
│   ├── faulty_imprint (18 images)
│   ├── pill_type (8 images)
│   └── scratch (23 images)
├── test/
│   ├── color (24 images)
│   ├── combined (16 images)
│   ├── contamination (20 images)
│   ├── crack (25 images)
│   ├── faulty_imprint (18 images)
│   ├── good (25 images)
│   ├── pill_type (8 images)
│   └── scratch (23 images)
└── train/
    └── good/
        └── good (266 images)
```

Et quelques visualisations d'images de ce dossier :

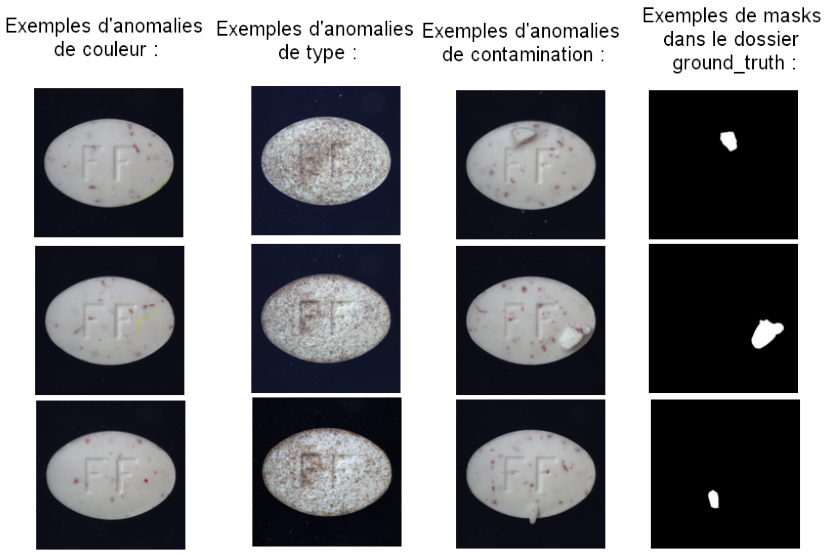


Figure 4.1 : Extrait du dataset Pill de MVTec

Le dossier **train** contient uniquement des images normales (sans aucune anomalie).Ces images servent à entraîner les modèles GAN ou autres modèles non supervisés. L'absence d'anomalies dans cet ensemble permet au modèle de se concentrer sur la reconstruction des images normales.

Le dossier **test** contient à la fois des images normales (good/) et des images contenant des anomalies

classées en différentes catégories : color, combined (défauts combinant plusieurs types d'anomalies), contamination (présence de particules indésirables ou de contaminations sur les comprimés), crack (issures visibles sur les comprimés. faulty_imprint (défauts dans l'impression ou le marquage sur les comprimés), pill_type (erreurs liées à un mauvais type de comprimé) et scratch (rayures visibles sur les comprimés).

Le dossier **ground_truth** fournit des masques binaires annotés au pixel près pour les images anormales du dossier test/. Chaque masque correspond à une image du dossier test et indique la localisation exacte des anomalies.

Bibliothèques utilisées

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
from torchvision.utils import save_image
from torch.utils.data import DataLoader, Dataset
from torch.utils.data import ConcatDataset
from sklearn.metrics import classification_report, precision_score, recall_score, f1_score,
     accuracy_score, jaccard_score
from PIL import Image
import os
```

4.2.3 Data preprocessing

Le preprocessing réalisé est conçu pour préparer les données du dataset MVTec (Pill) avant leur utilisation dans le modèle GAN. Les images sont d'abord chargées via une classe personnalisée MVTecDataset, qui hérite de la classe Dataset de PyTorch.

```
# Chemin principal du dataset
data_dir = r"path\pill"
# Les resultats sont dans le dossier : "path\GAN_Output"
# Classe pour importer le Dataset
# Grace a celle-ci, il sera plus simple d'afficher des images pour l'evaluation
class MVTecDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.image_paths = [os.path.join(root_dir, img) for img in os.listdir(root_dir)]
    def __len__(self):
        return len(self.image_paths)
    def __getitem__(self, idx):
        img_path = self.image_paths[idx]
        try:
            img = Image.open(img_path).convert("RGB")
        except Exception as e:
            print(f"Error opening file {img_path}: {e}")
            raise
        if self.transform:
            img = self.transform(img)
        return img

transform = transforms.Compose([
```

```
    transforms.Resize((img_size, img_size)),
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])  # Scale to [-1, 1]
])
dataset = MVTecDataset(root_dir=r"path\pill\train\good", transform=transform)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

Cette classe parcourt le répertoire spécifié (root_dir) pour collecter les chemins des images et les charger en mémoire. Les images sont ensuite converties au format RGB à l'aide de la bibliothèque PIL pour garantir leur compatibilité avec le modèle. Une gestion des erreurs est également intégrée pour signaler tout problème lié au chargement des fichiers.

Les images passent ensuite par une série de transformations définies dans un pipeline transforms.Compose. Elles sont d'abord redimensionnées à une taille fixe (par exemple, 64x64) pour garantir une entrée cohérente au modèle. Elles sont ensuite converties en tenseurs PyTorch avec transforms.ToTensor, ce qui normalise les pixels dans la plage [0, 1]. Enfin, elles sont normalisées davantage avec transforms.Normalize pour centrer leurs valeurs autour de 0 dans la plage [-1, 1]. Cette normalisation est particulièrement cruciale pour stabiliser l'entraînement des GAN, qui bénéficient de données centrées et normalisées.

Une fois les transformations appliquées, le dataset est instancié pour contenir les images normales issues du dossier train/good. Un DataLoader est utilisé pour charger ces données en mini-batches, avec une option `shuffle = True` pour mélanger les données à chaque époque d'entraînement. Ce chargement par lots permet une utilisation efficace de la mémoire et accélère l'entraînement. Cela garantit que le modèle reçoit des données correctement formatées, homogènes et adaptées à l'entraînement, en se concentrant uniquement sur les images normales afin que le modèle GAN puisse se spécialiser dans leur reconstruction.
Je vais maintenant présenter le fonctionnement du GAN dans ce contexte de détection d'anomalies ci-dessous.

4.2.4 Implémentation du GAN

Architecture du Générateur

- Le générateur prend un vecteur latent (un bruit aléatoire) comme entrée et génère une image synthétique de taille fixée (64x64 dans notre cas).
- Le générateur a été entraîné à produire des reconstructions d'images normales, ce qui permet d'utiliser les différences entre les images originales et reconstruites pour détecter des anomalies.

Le fichier https ://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/bicyclegan/models.py de @eriklindernoren sur GitHub m'a grandement aidé à programmer les lignes qui vont suivre pour les classes Generator et Discriminator :

```
# Generateur
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 128),
            nn.ReLU(True),
            nn.Linear(128, 256),
            nn.BatchNorm1d(256),
            nn.ReLU(True),
            nn.Linear(256, 512),
```

```
            nn.BatchNorm1d(512),
            nn.ReLU(True),
            nn.Linear(512, img_size * img_size * 3),
            nn.Tanh()
        )
    def forward(self, z):
        img = self.model(z)
        img = img.view(img.size(0), 3, img_size, img_size)
        return img
```

Architecture du Discriminateur

- Le discriminateur est conçu pour distinguer les images réelles (du dataset) des images générées par le générateur.
- Pendant l'entraînement, il aide indirectement le générateur à améliorer la qualité de ses reconstructions.

```
# Discriminateur
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(img_size * img_size * 3, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )
    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity
```

Ensuite, il faut mettre en place les éléments nécessaires pour entraîner un modèle GAN. Tout d'abord, définir le device à utiliser, en détectant automatiquement si un GPU est disponible (cuda), sinon basculer sur le CPU. Ensuite, les deux modèles du GAN, le générateur et le discriminateur, sont initialisés et transférés sur le device sélectionné.

```
# Definition du device (GPU si disponible, sinon CPU)
# Uniquement pour l'entrainement du reseau de neurones
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Initialisation des modeles
generator = Generator().to(device)
discriminator = Discriminator().to(device)

# Optimiseurs Adam
optimiseur_G = torch.optim.Adam(generator.parameters(), lr=lr, betas=(b1, b2))
optimiseur_D = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(b1, b2))

# Fonction de coût
adversarial_loss = nn.BCELoss().to(device)
```

Définissons les hyperparamètres du GAN :

```
# Hyperparamètres
latent_dim = 100
img_size = 64
batch_size = 32
epochs = 500
lr = 0.0002
b1, b2 = 0.5, 0.999
```

On peut maintenant lancer l'entraînement du modèle GAN, en ajoutant des checkpoints tous les 20 epochs. Avant de commencer l'entraînement, il vérifie si des checkpoints précédemment sauvegardés sont disponibles dans le répertoire spécifié. Si c'est le cas, les poids du générateur et du discriminateur sont restaurés, ainsi que les états des optimisateurs associés. De plus, l'indice de l'époque sauvegardée est chargé pour permettre une reprise précise de l'entraînement. En l'absence de checkpoints, l'entraînement démarre à l'époque 0.

Le code suivant pour la sauvegarde des checkpoints provient de la source suivante : https ://medium.com/@merilainen.vili/save-and-load-models-to-disk-in-pytorch-python-a-complete-guide-a667057b511c.

```
# Boucle d'entrainement
debut_epoch = 0

# Chargement des checkpoints si disponibles
if os.path.exists(os.path.join(checkpoint_dir, 'generator_checkpoint.pth')):
generator.load_state_dict(torch.load(os.path.join(checkpoint_dir,
    'generator_checkpoint.pth')))
    discriminator.load_state_dict(torch.load(os.path.join(checkpoint_dir,
    'discriminator_checkpoint.pth')))
    optimiseur_G.load_state_dict(torch.load(os.path.join(checkpoint_dir,
    'optimiseur_G_checkpoint.pth')))
    optimiseur_D.load_state_dict(torch.load(os.path.join(checkpoint_dir,
    'optimiseur_D_checkpoint.pth')))
    debut_epoch = torch.load(os.path.join(checkpoint_dir,
    'epoch_checkpoint.pth')) + 1
    print(f"Resuming training from epoch {debut_epoch}")

for epoch in range(debut_epoch, epochs):
    for i, imgs in enumerate(dataloader):
        valid = torch.ones(imgs.size(0), 1).to(device)
        fake = torch.zeros(imgs.size(0), 1).to(device)
        real_imgs = imgs.to(device)

        # Entrainement du Generator
        optimiseur_G.zero_grad()
        z = torch.randn(imgs.size(0), latent_dim).to(device)
        gen_imgs = generator(z)
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)
        g_loss.backward()
        optimiseur_G.step()

        # Entrainement du Discriminator
        optimiseur_D.zero_grad()
        vraie_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (vraie_loss + fake_loss) / 2
        d_loss.backward()
```

```
        optimiseur_D.step()
    print(f"Epoch {epoch + 1}/{epochs}, Generator Loss: {g_loss.item():.4f},
    Discriminator Loss: {d_loss.item():.4f}")

    # Sauvegarde des images toutes les 10 epochs
    if (epoch + 1) % 10 == 0:
        save_image(gen_imgs.data[:25], os.path.join(image_dir, f"epoch_{epoch + 1}.png"),
        nrow=5, normalize=True)

    # Sauvegarde du modèles et des images toutes les 20 epochs
    if (epoch + 1) % 20 == 0:
        torch.save(generator.state_dict(), os.path.join(checkpoint_dir,
        'generator_checkpoint.pth'))
        torch.save(discriminator.state_dict(), os.path.join(checkpoint_dir,
        'discriminator_checkpoint.pth'))
        torch.save(optimiseur_G.state_dict(), os.path.join(checkpoint_dir,
        'optimiseur_G_checkpoint.pth'))
        torch.save(optimiseur_D.state_dict(), os.path.join(checkpoint_dir,
        'optimiseur_D_checkpoint.pth'))
        torch.save(epoch, os.path.join(checkpoint_dir,
        'epoch_checkpoint.pth'))
        print(f"Les checkpoints sont sauvegardés à l' {epoch + 1}.")
print("Entrainement terminé")
```

La boucle d'entraînement alterne entre l'amélioration du générateur et celle du discriminateur. Pour chaque lot d'images, des labels (valid pour les vraies et fake pour les générées) sont créés. Le générateur, à partir d'un bruit latent aléatoire, produit des images synthétiques qui sont évaluées par le discriminateur. Une perte adversariale est calculée pour pousser le générateur à améliorer la qualité de ses images, et ses poids sont mis à jour en conséquence.

Le discriminateur est ensuite entraîné à distinguer les vraies images des images générées, avec une mise à jour basée sur une perte combinant ces deux tâches.

Tous les 10 epochs, les images générées sont sauvegardées pour suivre visuellement la progression de l'entraînement. De plus, tous les 20 epochs, les états des modèles, des optimisateurs, et l'indice de l'époque actuelle sont sauvegardés dans des fichiers de checkpoints.

```
# Repertoire pour sauvegarder les checkpoints
checkpoint_dir = "checkpoints"
os.makedirs(checkpoint_dir, exist_ok=True)

# Sauvegarde des modeles et des optimiseurs
torch.save(generator.state_dict(), os.path.join(checkpoint_dir, "generator_fin.pth"))
torch.save(discriminator.state_dict(), os.path.join(checkpoint_dir, "discriminator_fin.pth"))
torch.save(optimiseur_G.state_dict(), os.path.join(checkpoint_dir, "optimiseur_G_fin.pth"))
torch.save(optimiseur_D.state_dict(), os.path.join(checkpoint_dir, "optimiseur_D_fin.pth"))

# Sauvegarde de l'epoch
torch.save(500, os.path.join(checkpoint_dir, "epoch_fin.pth"))
print("Entrainement et modèle sauvegardé après 500 epochs")
```

4.3 Résultats

Les pertes affichées au fil des epochs reflètent l'équilibre dynamique entre les deux réseaux. Le générateur a montré une amélioration progressive dans sa capacité à produire des images réalistes, tandis que le discriminateur a affiné sa capacité à distinguer les vraies images des images générées.

```
Epoch 301/500, Generator Loss: 0.5631, Discriminator Loss: 0.8465
Epoch 302/500, Generator Loss: 0.8729, Discriminator Loss: 0.5132
Epoch 303/500, Generator Loss: 0.6419, Discriminator Loss: 0.7639
Epoch 304/500, Generator Loss: 0.8542, Discriminator Loss: 0.6629
Epoch 305/500, Generator Loss: 0.6414, Discriminator Loss: 0.7888
Epoch 306/500, Generator Loss: 0.8483, Discriminator Loss: 0.6389
Epoch 307/500, Generator Loss: 0.6528, Discriminator Loss: 0.7378
Epoch 308/500, Generator Loss: 0.6870, Discriminator Loss: 0.6968
Epoch 309/500, Generator Loss: 0.7928, Discriminator Loss: 0.6200
Epoch 310/500, Generator Loss: 0.9575, Discriminator Loss: 0.7088
Epoch 311/500, Generator Loss: 1.0686, Discriminator Loss: 0.5108
Epoch 312/500, Generator Loss: 0.8201, Discriminator Loss: 0.6055
Epoch 313/500, Generator Loss: 0.5041, Discriminator Loss: 0.8035
Epoch 314/500, Generator Loss: 0.7535, Discriminator Loss: 0.6562
Epoch 315/500, Generator Loss: 0.5856, Discriminator Loss: 0.7448
Epoch 316/500, Generator Loss: 1.4538, Discriminator Loss: 0.6734
Epoch 317/500, Generator Loss: 1.1666, Discriminator Loss: 0.8656
Epoch 318/500, Generator Loss: 0.7076, Discriminator Loss: 0.7219
Epoch 319/500, Generator Loss: 0.7073, Discriminator Loss: 0.6659
Epoch 320/500, Generator Loss: 1.0642, Discriminator Loss: 0.5794
Checkpoints saved at epoch 320.
Epoch 321/500, Generator Loss: 0.2642, Discriminator Loss: 0.9903
Epoch 322/500, Generator Loss: 0.8301, Discriminator Loss: 0.6722
Epoch 323/500, Generator Loss: 1.0072, Discriminator Loss: 0.6563
...
Epoch 499/500, Generator Loss: 0.7984, Discriminator Loss: 0.6281
Epoch 500/500, Generator Loss: 0.5758, Discriminator Loss: 0.7214
```

Figure 4.2 : Évolution des pertes au fil des époch, à partir de 300

À l'époque finale (500), les pertes respectives du générateur et du discriminateur étaient de 0.5758 et 0.7214, indiquant un certain équilibre entre les deux réseaux. L'entraînement s'est terminé avec succès, confirmant que les mécanismes de reprise et de sauvegarde ont fonctionné comme prévu.

Détection des anomalies Une fois les GAN entraînés, les anomalies sont détectées en mesurant les erreurs de reconstruction. Les GAN reconstruisent très bien les images normales, mais ils échouent à reconstruire les anomalies, ce qui entraîne des erreurs élevées sur les zones anormales.

L'implémentation du processus d'évaluation des erreurs de reconstruction pour les images de test à l'aide d'un modèle GAN est ci-dessous. Tout d'abord, la fonction `calculate_reconstruction_error` calcule l'erreur quadratique moyenne (MSE) entre les images originales et reconstruites, pixel par pixel, en prenant en compte toutes les dimensions des images :

```
def calculate_reconstruction_error(original, reconstructed):
    return torch.mean((original - reconstructed) ** 2, dim=[1, 2, 3])

# Chargement des données de test
root_dir = r"path\pill\test"
test_dataset = MVTecDataset(root_dir, transform=transform)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Calcul des erreurs de reconstruction
reconstruction_errors = []
generator.eval()

with torch.no_grad():
    for imgs in test_loader:
        imgs = imgs.to(device)
        z = torch.randn(imgs.size(0), latent_dim).to(device)
        reconstructed_imgs = generator(z)
        error = calculate_reconstruction_error(imgs, reconstructed_imgs)
        reconstruction_errors.extend(error.cpu().numpy())
```

```
# Affichage des resultats
for idx, error in enumerate(reconstruction_errors):
    print(f"Image {idx + 1}, Reconstruction Error: {error:.4f}")
```

Pendant l'évaluation, le modèle générateur est mis en mode évaluation (`generator.eval()`) pour désactiver les mises à jour des poids et optimiser les calculs. Une boucle with `torch.no_grad()` est utilisée pour désactiver la rétropropagation, réduisant ainsi la consommation de mémoire.

On commence par calculer un seuil dynamique pour détecter les anomalies, basé sur le 95e percentile des erreurs de reconstruction observées dans les données, ce qui garantit que seules les valeurs les plus extrêmes seront considérées comme anomalies.

```
# Definition d'un seuil basé sur le 95e percentile
nv_seuil = np.percentile(errors_array, 95)
print(f"Nouveau seuil pour les anomalies: {nv_seuil:.4f}")
```

Une fonction de dénormalisation est ensuite définie pour ramener les images, normalisées dans la plage [-1, 1], à leur plage originale [0, 1], facilitant leur visualisation. Une autre fonction calcule les métriques de performance des masques de détection d'anomalies : l'Intersection over Union (IoU) et le Dice Score, qui comparent les zones détectées par le modèle avec les zones annotées dans les masques de référence.

```
# Fonction pour denormaliser une image
def denormalize(img_tensor):
    return img_tensor * 0.5 + 0.5
    # Annule la normalisation entre [-1, 1] pour revenir à [0, 1]
```

Une autre fonction calcule les métriques de performance des masques de détection d'anomalies : l'Intersection over Union (IoU) et le Dice Score, qui comparent les zones détectées par le modèle avec les zones annotées dans les masques de référence.

Intersection over Union (IoU)

$$\text{IoU} = \frac{\text{Intersection}}{\text{Union}} \tag{4.1}$$

Cette métrique mesure le rapport entre la zone d'intersection entre la segmentation prédite et la vérité terrain (*ground truth*) et la zone totale couverte par les deux (*union*).

Dice Score (Coefficient de Similarité de Sørensen-Dice)

$$\text{Dice} = \frac{2 \times \text{Intersection}}{\text{Cardinalité Prédite} + \text{Cardinalité Vérité Terrain}} \tag{4.2}$$

Le Dice Score met davantage l'accent sur les régions d'intersection et est particulièrement utile lorsque les classes sont déséquilibrées (par exemple, lorsque les objets sont petits par rapport à l'image totale.
Les deux métriques varient entre 0 et 1, où 1 représente une correspondance parfaite.

```
# Fonction pour calculer les métriques (IoU et Dice Score)
def metrique(predicted_mask, ground_truth_mask):
    predicted_mask_flat = predicted_mask.view(-1).cpu().numpy()
    ground_truth_mask_flat = ground_truth_mask.view(-1).cpu().numpy()

    # IoU (Intersection over Union)
    iou = jaccard_score(ground_truth_mask_flat, predicted_mask_flat)

    # Dice Score
    dice = f1_score(ground_truth_mask_flat, predicted_mask_flat)

    return iou, dice
```

Puis pour chaque image du jeu de test, le masque d'anomalie correspondant est chargé depuis le dossier ground_truth. Une carte d'erreur est générée en calculant la différence absolue entre l'image originale et reconstruite, et les valeurs au-dessus du seuil sont considérées comme anomalies. Les résultats sont affichés sous forme de visualisation comparative : l'image originale, l'image reconstruite, la carte d'erreur (avec une échelle de couleur rouge pour les zones de différence élevée), et le masque de référence. Cette visualisation inclut également les scores IoU et Dice directement sur l'image de référence pour évaluer la qualité de la détection.

```
# Fonction pour visualiser les resultats
def visualize_results(original, reconstructed, ground_truth_mask, idx, threshold=0.1):
    # Generation de la carte d'erreur
    error_map = torch.abs(original - reconstructed).sum(dim=0) > threshold
    # Seuil pour détecter les anomalies

    # Calcul des métriques
    iou, dice = metrique(error_map, ground_truth_mask)

    # Visualisation de l'image reconstruite VS image originale
    # Image originale
    plt.subplot(1, 4, 1)
    plt.imshow(denormalize(original).permute(1, 2, 0).cpu().numpy())
    plt.title("Original Image")
    plt.axis("off")

    # Image reconstruite
    plt.subplot(1, 4, 2)
    plt.imshow(denormalize(reconstructed).permute(1, 2, 0).cpu().numpy())
    plt.title("Reconstructed Image")
    plt.axis("off")

    # Carte d'erreur
    diff = torch.abs(original - reconstructed).sum(dim=0).cpu().numpy()
    plt.subplot(1, 4, 3)
    plt.imshow(diff, cmap="hot")
    plt.title(f"Error Map {idx + 1}")
    plt.axis("off")

    # Masque de ground truth
    plt.subplot(1, 4, 4)
    plt.imshow(ground_truth_mask.cpu().numpy(), cmap="gray")
    plt.title(f"Ground Truth Mask\nIoU: {iou:.4f}, Dice: {dice:.4f}")
    plt.axis("off")
    plt.suptitle(f"Image {idx + 1}")
    plt.show()

for idx, (original, reconstructed) in enumerate(zip(imgs, reconstructed_imgs)):
    mask_path = os.path.join(ground_truth_dir, mask_files[idx])
    # Charger le masque correspondant
    ground_truth_mask = load_mask(mask_path, target_size=(64, 64))
    # Charger et redimensionner le masque
    visualize_results(original, reconstructed, ground_truth_mask, idx)
```

On peut alors afficher quelques visualisations :

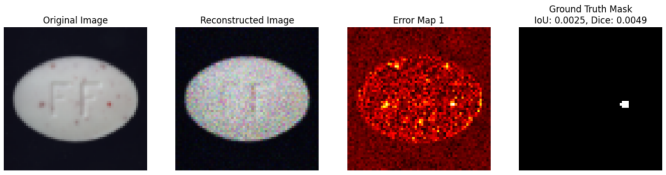


Image 1

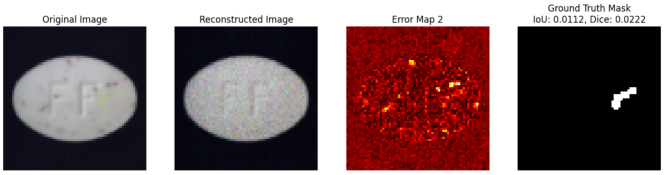


Image 2

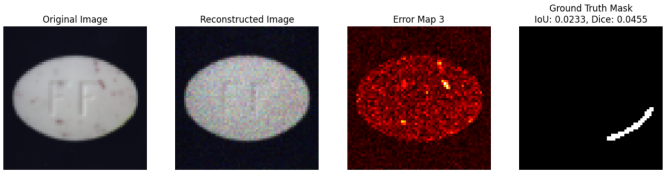


Image 3

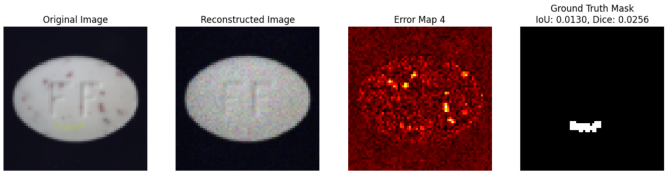


Image 4

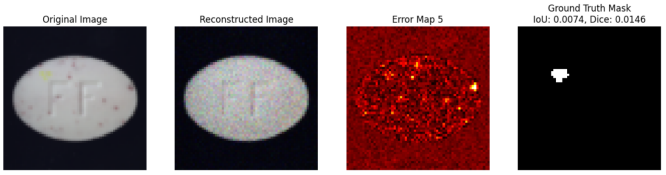


Image 5

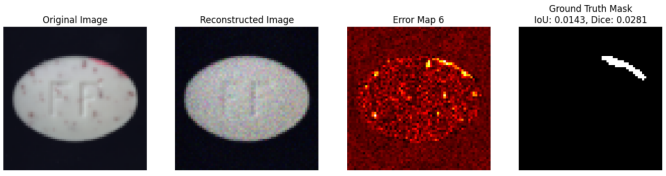


Image 6

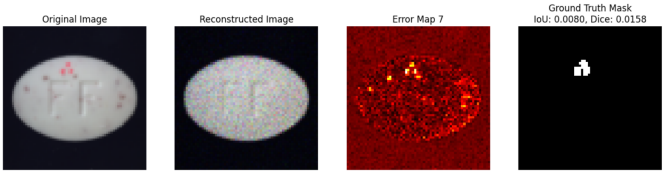


Image 7

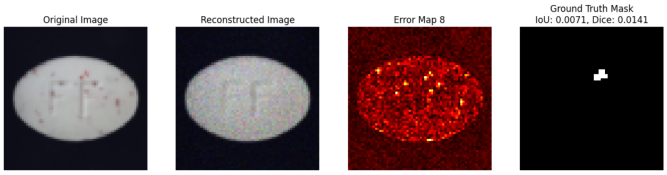


Image 8

Puis on peut résumer les performances d'un modèle GAN en termes de détection d'anomalies, en s'appuyant sur des métriques de classification standard appliquées aux masques d'anomalies prédits et aux masques de ground truth. On définit la fonction `calculate_classification_metrics`, qui évalue cinq métriques principales : l'Intersection over Union (IoU), le Dice Score, la précision, le rappel, et l'exactitude (accuracy). Pour chaque paire de masque prédit et masque de référence, les pixels sont aplatis pour permettre une comparaison pixel par pixel, et les métriques sont calculées à l'aide de fonctions de la bibliothèque sklearn.

Le code charge ensuite les masques de référence depuis un répertoire donné, en les redimensionnant à une taille uniforme (64x64). Parallèlement, il génère les masques prédits en calculant une carte d'erreur basée sur la différence absolue entre les images originales et leurs reconstructions, en appliquant un seuil (0.5) pour détecter les zones d'anomalies. Ces masques sont collectés dans deux listes distinctes : `predicted_masks` pour les prédictions et `ground_truth_masks` pour les masques de référence.

```
# Fonction pour calculer les metriques globales
def calculate_classification_metrics(mask_predit, ground_truth_mask):
    metrics = {"IoU": [ ], "Dice": [ ], "Precision": [ ], "Recall": [ ], "Accuracy": [ ]}

    for predicted_mask, ground_truth_mask in zip(mask_predit, ground_truth_mask):
        # Aplatissement des masques pour les comparer pixel par pixel
        predicted_flat = predicted_mask.view(-1).cpu().numpy()
        ground_truth_flat = ground_truth_mask.view(-1).cpu().numpy()

        # Calcul des metriques
        iou = jaccard_score(ground_truth_flat, predicted_flat)
        dice = f1_score(ground_truth_flat, predicted_flat)
        precision = precision_score(ground_truth_flat, predicted_flat)
        recall = recall_score(ground_truth_flat, predicted_flat)
        accuracy = accuracy_score(ground_truth_flat, predicted_flat)

        # Ajout des métriques au dictionnaire
        metrics["IoU"].append(iou)
        metrics["Dice"].append(dice)
        metrics["Precision"].append(precision)
        metrics["Recall"].append(recall)
        metrics["Accuracy"].append(accuracy)
    return metrics

# Initialisation des listes pour stocker les masques prédits et les masques de ground truth
mask_predit = [ ]
ground_truth_mask = [ ]

# Generation des masques prédits et collecter les metriques
for idx, (original, reconstructed) in enumerate(zip(imgs, reconstructed_imgs)):
    mask_path = os.path.join(ground_truth_dir, mask_files[idx])
    ground_truth_mask = load_mask(mask_path, target_size=(64, 64))
    # Chargement et redimensionnement du masque
    ground_truth_mask.append(ground_truth_mask)

    # seuil pour détecter l'anomalie et carte d'erreur
    error_map = torch.abs(original - reconstructed).sum(dim=0) > 0.5
    mask_predit.append(error_map)

# Calcul des métriques globales
metrics = calculate_classification_metrics(mask_predit, ground_truth_mask)
```

```
# dataFrame pour afficher les résultats
resultats_df = pd.DataFrame({
    "Image": [f"Image {i+1}" for i in range(len(mask_predit))],
    "IoU": metrics["IoU"],
    "Dice": metrics["Dice"],
    "Precision": metrics["Precision"],
    "Recall": metrics["Recall"],
    "Accuracy": metrics["Accuracy"]
})

# Resultats
print("=== Summary des Classification Metrics ===")
print(f"Moy IoU: {np.mean(metrics['IoU']):.4f}")
print(f"Moy Dice: {np.mean(metrics['Dice']):.4f}")
print(f"Moy Precision: {np.mean(metrics['Precision']):.4f}")
print(f"Moy Recall: {np.mean(metrics['Recall']):.4f}")
print(f"Moy Accuracy: {np.mean(metrics['Accuracy']):.4f}")
```

Voici le rapport de classification final du GAN implémenté :

```
=== Summary of Classification Metrics ===
Mean IoU: 0.1063
Mean Dice: 0.1863
Mean Precision: 0.1421
Mean Recall: 0.3771
Mean Accuracy: 0.9807
Classification report saved to 'classification_report.csv'.
```

Figure 4.3 : Rapport de classification du modèle GAN

Le modèle montre une IoU moyenne de 0.1063 et un Dice Score moyen de 0.1863, indiquant une faible correspondance entre les masques prédits et les masques de référence. La précision est particulièrement basse (0.1421), ce qui montre que de nombreuses anomalies détectées sont fausses. Cependant, le rappel est meilleur (0.3771), indiquant que le modèle détecte une partie significative des anomalies réelles. L'exactitude (0.9807) est élevée, mais elle peut être biaisée en raison de la prédominance de pixels normaux par rapport aux pixels d'anomalies. Ces résultats reflètent des performances limitées du modèle, suggérant qu'un ajustement supplémentaire des hyperparamètres ou du seuil pourrait améliorer la détection.

Chapitre 5

Carte de Kohonen (Item 4)

Historiquement, les réseaux de communication industriels étaient des systèmes fermés, limités aux échanges internes. Cependant, l'émergence de l'Industrie 4.0 a marqué une transition vers une intégration croissante des technologies de l'information et de la communication (TIC) dans les systèmes de contrôle industriels (ICS) Schuster et al. (2013). De nombreux dispositifs et protocoles utilisés actuellement ont été initialement conçus pour des réseaux d'automatisation isolés, caractérisés par leur fiabilité élevée et leur séparation physique des autres réseaux. Néanmoins, ces systèmes n'ayant pas été développés avec une attention particulière à la cybersécurité, ils présentent des lacunes significatives en matière de protection informatique. Cette conception devient problématique face à la généralisation des connexions à des réseaux ouverts, souvent moins fiables, comme les réseaux de supervision basés sur des protocoles ouverts.

5.1 Présentation des cartes de Kohonen

Une Carte Auto-Organisatrice (Self-Organizing Map, SOM) est un réseau neuronal artificiel (RNA) qui reçoit des données de haute dimension en entrée et apprend leurs structures complexes pour les représenter sous forme d'une carte bidimensionnelle de neurones en sortie. Elle fonctionne comme une grille dynamique et flexible qui s'étend sur les échantillons de données en entrée pour les approximer. En d'autres termes, elle ajuste des vecteurs prototypes ordonnés en deux dimensions à la distribution des vecteurs de données d'entrée en haute dimension.

Cette méthode s'inspire du fonctionnement du cerveau humain, où des entrées similaires activent des neurones situés dans la même zone du cerveau. Ainsi, les instances de données proches dans l'espace des données en entrée sont mappées sur des neurones proches sur la carte de sortie. Cette propriété, appelée correction topologique, rend les SOM uniques et utiles pour explorer les ensembles de données, car elles permettent une représentation visuelle des données de haute dimension.

De plus, les SOM ne nécessitent pas de jeux de données étiquetés manuellement comme entrée et sont classées comme une méthode d'apprentissage automatique non supervisée.

5.2 Objectif

Détecter les anomalies causées par des intrusions représente un défi majeur dans les environnements industriels en raison des interdépendances environnementales complexes et des protocoles de bus de terrain propriétaires. Dans l'article *Detecting Anomalies by using Self-Organizing Maps in Industrial Environments* de Hormann and Fischer (2019) une méthode basée sur un réseau pour détecter les anomalies en utilisant des réseaux neuronaux artificiels non supervisés appelés Cartes Auto-Organisatrices (SOMs, pour Self-Organizing Maps) est proposée.

5.3 Jeu de données

Le jeu de données **CICIDS2017 (Canadian Institute for Cybersecurity Intrusion Detection System Dataset 2017)** est un jeu de données bien connu dans le domaine de la détection d'intrusions en cybersécurité. Il est consultable et téléchargeable sur ce lien Kaggle : Network Intrusion Dataset on Kaggle. Il a été conçu pour simuler un réseau d'entreprise réaliste et capturer divers types d'attaques afin de tester et d'évaluer des systèmes de détection d'intrusions (IDS).

Description détaillée de ce jeu de données : Le jeu de données est organisé en fichiers CSV et contient des enregistrements de trafic réseau capturés via des outils comme Wireshark. Chaque enregistrement correspond à un flux réseau.

Trafic normal : Représente le trafic légitime sans comportement malveillant.

Types d'attaques inclues dans le dataset :

- `DOS (Denial of Service)` et `DDOS (Distributed Denial of Service)`
- `Brute Force` : Essais multiples pour accéder à des services (ex. SSH).
- `Botnet` : Activité d'un botnet dans le réseau.
- `PortScan` : Balayage des ports pour identifier des services vulnérables.
- `Infiltration` : Tentatives d'accès non autorisé aux systèmes internes.
- `Exfiltration` : Extraction de données sensibles hors du réseau.
- `Attaques Web` : Injection SQL, XSS (Cross-Site Scripting), etc.

5.3.1 Data preprocessing

Les données ont été réparties en deux catégories principales : les données normales et les données anormales. Les fichiers du dataset `CICIDS2017` ont été organisés par jour, période de la journée (matin, après-midi) et type d'activité (bénigne ou attaque). Puis j'ai prétraité chaque fichier du dataset Kaggle pour garantir la cohérence des colonnes et des étiquettes avant la concaténation. Les colonnes non pertinentes ont été éliminées, et les valeurs manquantes ont été imputées pour éviter les biais pendant l'entraînement.
Une fois les données regroupées, leur label explicite leur a été attribué pour identifier les types d'attaques spécifiques (DDoS, DoS, PortScan, etc.) et le label de référence pour la classe normale. Cette organisation a permis de constituer un dataset plus ou moins équilibré, essentiel pour l'entraînement du SOM et l'analyse des anomalies, tout en facilitant l'interprétation des résultats et la validation des performances du modèle.

5.3.2 Méthode décrite par Hormann and Fischer (2019)

L'objectif de l'algorithme proposé est d'identifier les clusters et leurs centroïdes dans les SOMs afin de mieux comprendre la structure sous-jacente des données. Le code des algorithmes décrits dans le papier ne sont pas disponibles, ainsi je me suis aidée des sources suivantes : https ://www.kaggle.com/code/nilsschlueter/self-organizing-maps-for-anomaly-detection et https ://www.analyticsvidhya.com/blog/2021/09/beginners-guide-to-anomaly-detection-using-self-organizing-maps/.

Lors de la **phase d'entraînement**, ils ont créé deux réseaux neuronaux : l'un pour regrouper les données du réseau et l'autre pour trouver les centroïdes des clusters. Pendant la **phase opérationnelle**, la méthode permet de détecter les anomalies en comparant de nouveaux échantillons de données au premier modèle SOM entraîné. L'utilisation d'un intervalle de confiance est faite pour déterminer si un échantillon est trop éloigné de son unité correspondante la plus proche. Un intervalle de confiance supplémentaire pour le second SOM est proposé afin de minimiser les faux positifs, qui constituent un inconvénient majeur des méthodes d'apprentissage automatique dans la détection des anomalies.

5.3.3 Mise en oeuvre

Dans cette partie, j'ai tenté de mettre en œuvre la méthode décrite par Hormann and Fischer (2019). J'ai de mon côté implémenté qu'un seul MiniSOM pour des raisons computationnelles.

Bibliothèques utilisées

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from minisom import MiniSom
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.preprocessing import MinMaxScaler
from collections import Counter
import plotly.graph_objects as go
from plotly.subplots import make_subplots
import plotly.graph_objects as go
from scipy.spatial.distance import euclidean
import os
```

Voici le code utilisé pour prétraiter les données :

```
# Paths to the directories
normal_dir = "path\\CICIDS2017\\Reduced\\Normal"
arnomal_dir = "path\\CICIDS2017\\Reduced\\Anormal"

def load_dataset_label(filepath, label):
    df = pd.read_csv(filepath)
    df.rename(columns={df.columns[-1]: 'Label'}, inplace=True)
    df['Label'] = label
    return df

# Loading des datasets un par un, a commencer par les donnees normales
normal_data = pd.DataFrame()
for fichier in os.listdir(normal_dir):
    if fichier.endswith(".csv"):
        filepath = os.path.join(normal_dir, fichier)
        print(f"Chargement du dataset normal : {fichier}")
        df = load_dataset_label(filepath, 'Normal')
        normal_data = pd.concat([normal_data, df], ignore_index=True)

# Redefinition des lables pour corriger l'orthographe
attack_labels = {
    "Reduced_Arnomal_Wednesday-workingHours.pcap_ISCX.csv": "DoS",
    "Reduced_Arnomal_Tuesday-WorkingHours.pcap_ISCX.csv": "BruteForce",
    "Reduced_Arnomal_Thursday-WorkingHours-Morning-WebAttacks.pcap_ISCX.csv": "WebAttack",
    "Reduced_Arnomal_Thursday-WorkingHours-Afternoon-Infilteration.pcap_ISCX.csv": "Infiltration",
    "Reduced_Arnomal_Friday-WorkingHours-Morning.pcap_ISCX.csv": "Bot",
    "Reduced_Arnomal_Friday-WorkingHours-Afternoon-PortScan.pcap_ISCX.csv": "PortScan",
    "Reduced_Arnomal_Friday-WorkingHours-Afternoon-DDos.pcap_ISCX.csv": "DDoS"
}

# Chargement des anomalies
anormal_data = pd.DataFrame()
```

```
for fichier, label in attack_labels.items():
    filepath = os.path.join(arnomal_dir, fichier)
    if os.path.exists(filepath):
        print(f"Chargement du dataset anormal : {fichier} avec le label '{label}'")
        df = load_dataset_label(filepath, label)
        arnomal_data = pd.concat([arnomal_data, df], ignore_index=True)
    else:
        print(f"Fichier non trouvé : {fichier}")

# Combinaison des datasets normal et anormal
data = pd.concat([normal_data, arnomal_data], ignore_index=True)
```

	Destination Port	Flow Duration	Total Fwd Packets	Total Backward Packets	Fwd Packet Length Mean	Bwd Packet Length Mean	Flow Packets/s	SYN Flag Count	RST Flag Count	Label
0	40474	7182757	1	5	6.0	6.000000	0.835334	0	0	Normal
1	28907	938509	1	5	6.0	6.000000	6.393119	0	0	Normal
2	0	119995732	123	0	0.0	0.000000	1.025036	0	0	Normal
3	443	142954	10	6	55.3	658.166667	111.924115	0	0	Normal
4	17541	2548725	1	6	6.0	6.000000	2.746471	0	0	Normal

Figure 5.1 : Extrait du dataset

Puis on met en oeuvre la méthode étape par étape :

Étape 1 Préparation des données

```
# Encodage des labels
label_encoder = LabelEncoder()
data['Label'] = label_encoder.fit_transform(data['Label'])

X = data.drop('Label', axis=1)
y = data['Label']

# Normalisation des features
scaler = MinMaxScaler()
# Remplacement des valeurs infinies par NaN
X.replace([np.inf, -np.inf], np.nan, inplace=True)
X.fillna(X.mean(), inplace=True)  # Remplace NaN par la moyenne de la colonne
X_scaled = scaler.fit_transform(X)
```

Étape 2 Entraînement de l'unique SOM (Self-Organizing Map)

```
som_shape = (10, 10)
som = MiniSom(som_shape[0], som_shape[1], X_scaled.shape[1], sigma=1.0, learning_rate=0.5)
som.random_weights_init(X_scaled)
som.train_random(X_scaled, 100)  # Entrainement sur 100 iterations
```

Pour implémenter le Kmeans implémenté ci-dessus, je me suis aidée de l'article : https ://dev.to/sajal2692/coding-k-means-clustering-using-python-and-numpy-fg1.

Étape 3 Identification des clusters

```
bmu_mapping = np.array([som.winner(x) for x in X_scaled])
clusters = {}
for idx, bmu in enumerate(bmu_mapping):
    bmu_tuple = tuple(bmu)
    if bmu_tuple not in clusters:
        clusters[bmu_tuple] = [ ]
    clusters[bmu_tuple].append(X_scaled[idx])
```

```
# Calcul des centroïdes
centroids = { }
for bmu, points in clusters.items():
    centroids[bmu] = np.mean(points, axis=0)
```

Étape 4 Phase opérationnelle (détection des anomalies avec deux intervalles de confiance)

```
anomalies = [ ]

    # Calcul des seuils pour le premier intervalle de confiance (BMU)
    bmu_distances = [euclidean(x, som.get_weights()[som.winner(x)]) for x in X_scaled]
    bmu_threshold = np.percentile(bmu_distances, confidence_bmu * 100)

    # Calcul des seuils pour le deuxième intervalle de confiance (centroïdes)
    centroid_distances = [ ]
    for bmu, points in clusters.items():
        for point in points:
            centroid_distances.append(euclidean(point, centroids[bmu]))
    centroid_threshold = np.percentile(centroid_distances, confidence_centroid * 100)

    # Vérification des anomalies pour chaque nouveau point
    for sample in new_samples:
        # Distance au BMU
        bmu = som.winner(sample)
        distance_bmu = euclidean(sample, som.get_weights()[bmu])
        # Distance au centroïde du cluster
        distance_centroid = euclidean(sample, centroids.get(bmu, np.zeros_like(sample)))
        # Anomalie si les distances dépassent les seuils
        if distance_bmu > bmu_threshold or distance_centroid > centroid_threshold:
            anomalies.append((sample, bmu, distance_bmu, distance_centroid))
    return anomalies

# Association de chaque BMU à la classe dominante
bmu_to_class = { }
for bmu in unique_bmus:
    indices = [i for i, x in enumerate(bmu_mapping) if tuple(x) == tuple(bmu)]
    class_counts = Counter(y[indices])
    most_common_class = class_counts.most_common(1)[0][0]
    bmu_to_class[tuple(bmu)] = most_common_class

# Association chaque point à la classe dominante de son BMU
y_pred_mapped = np.array([bmu_to_class[tuple(bmu)] for bmu in bmu_mapping])
```

Étape 5 Phase opérationnelle (détection des anomalies)

```
print("Rapport de classification:")
print(classification_report(y, y_pred_mapped, target_names=label_encoder.classes_,
zero_division=0))
```

5.4 Résultats

5.4.1 Intervalles de confiance

Les seuils définis pour la détection des anomalies dans ce modèle sont basés sur les distances calculées à partir des BMU (Best Matching Units) et des centroïdes. Le seuil de distance BMU est

fixé à 0.2111, correspondant au 95ème percentile des distances entre chaque point de données et son BMU. Cela signifie que 95 % des distances sont inférieures à ce seuil, tandis que les valeurs au-delà sont considérées comme potentiellement atypiques ou anormales. De même, le seuil de distance centroïde, fixé à 0.2993, correspond au 99ème percentile des distances entre chaque point de données et le centroïde du cluster auquel appartient son BMU, permettant une évaluation plus stricte des anomalies structurelles.

La visualisation des distances sous forme d'histogrammes met en évidence la répartition des distances BMU et des distances centroïdes. Dans l'histogramme des distances BMU, la majorité des points se situent en dessous du seuil, confirmant leur proximité avec leur BMU respectif. Cependant, certains points dépassent ce seuil, indiquant une potentielle anomalie. De manière similaire, l'histogramme des distances centroïdes révèle une répartition légèrement plus étalée, avec quelques distances dépassant le seuil, renforçant l'idée de points atypiques.

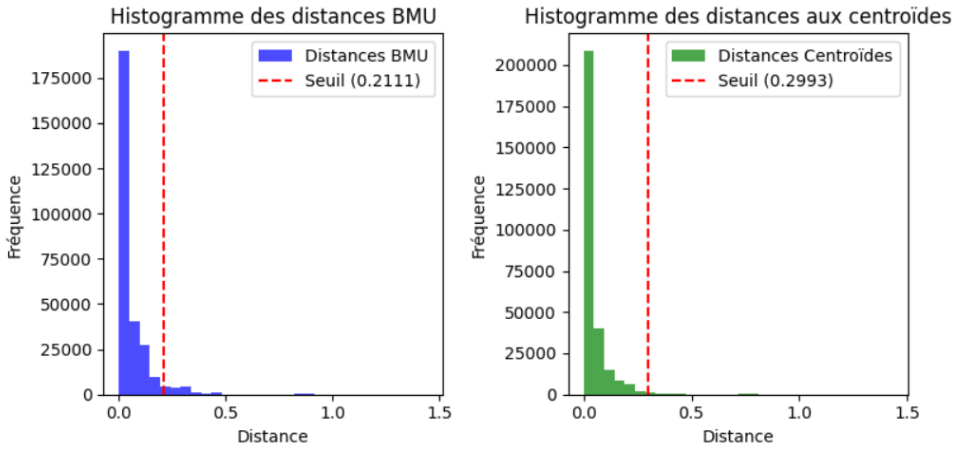


Figure 5.2 : Histogrammes des distances BMU et distances centroïdes

Lors de la détection des anomalies, un exemple d'échantillon a été identifié comme anomalie. Bien que sa distance BMU soit inférieure au seuil ($0.1582 < 0.2111$), sa distance au centroïde dépasse le seuil ($0.3060 > 0.2993$). Cela suggère que l'échantillon est proche de son BMU, mais relativement éloigné du centroïde de son cluster, indiquant une anomalie structurelle ou locale.

Enfin, un avertissement lié aux noms des colonnes a été détecté lors de la normalisation des nouveaux échantillons. Ce problème survient car le scaler a été initialisé avec des noms de colonnes spécifiques, mais l'échantillon fourni n'inclut pas ces noms. J'ai donc ajouté les noms de colonnes correspondants avant la transformation des données.

En conclusion, cette méthode offre une approche robuste pour détecter les anomalies, avec des seuils clairement définis et des visualisations explicatives. Des ajustements des seuils peuvent être envisagés pour optimiser la sensibilité et la spécificité de la détection selon les besoins.

5.4.2 Rapport de classification

Classification Report:

	precision	recall	f1-score	support
Bot	0.00	0.00	0.00	197
BruteForce	0.00	0.00	0.00	1384
DDoS	0.62	0.47	0.54	12803
DoS	0.82	0.53	0.65	25267
Infiltration	0.00	0.00	0.00	4
Normal	0.91	0.96	0.94	227311
PortScan	0.65	0.71	0.68	15893
WebAttack	0.00	0.00	0.00	218
accuracy			0.88	283077
macro avg	0.38	0.34	0.35	283077
weighted avg	0.87	0.88	0.87	283077

Figure 5.3 : Rapport de classification de SOM via la méthode inspirée de Hormann and Fischer (2019)

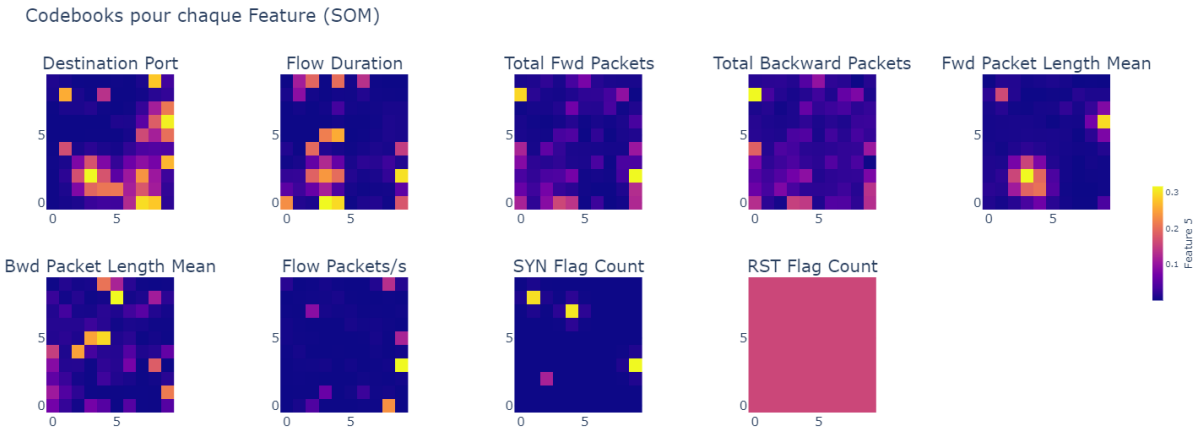
Le rapport de classification met en évidence les performances globales et les limites du modèle SOM. La classe normale est bien prédite, avec une précision et un rappel élevés (respectivement 91 % et 96 %), ce qui est attendu compte tenu de son poids important dans le dataset. Les classes associées aux attaques DoS et PortScan montrent des performances modérées, avec des F1-scores de 0.65 et 0.68 respectivement, indiquant que le modèle parvient à détecter ces anomalies dans la majorité des cas.
En revanche, les classes Bot, BruteForce, WebAttack, et Infiltration sont mal représentées et n'ont pas été correctement identifiées, mais nous verrons dans la partie suivante que ce n'est pas dû qu'au déséquilibre du dataset. Les métriques globales, comme l'accuracy (88 %), marquent les faibles performances sur les classes rares, comme le montrent les scores moyens non pondérés (macro avg : 0.35).

5.4.3 Codebooks

Les codebooks d'un SOM (Self-Organizing Map) représentent les poids associés à chaque neurone pour chaque feature. Ils servent de points de référence pour regrouper des données similaires dans l'espace des features. Chaque neurone de la carte est responsable de "représenter" un groupe de données similaires, et les valeurs des codebooks montrent les caractéristiques typiques des données associées à un neurone particulier. Ces visualisations permettent de mieux comprendre comment le SOM a organisé les données et quels neurones sont influencés par des features spécifiques.

```
# Definition des parametres pour les sous-graphiques
num_features = X_scaled.shape[1]
cols = 5  # Nombre de colonnes pour les subplots
rows = int(np.ceil(num_features / cols))  # Nombre de lignes base sur les features

# Initialisation des sous-graphiques
fig = make_subplots(
    rows=rows,
    cols=cols,
    subplot_titles=[f"{feature}" for i, feature in enumerate(X.columns)],
    horizontal_spacing=0.1,  # Espacement horizontal entre les subplots
    vertical_spacing=0.2,    # Espacement vertical entre les subplots
)
```

```
# à chaque codebook (feature) est ajouté un subplot
for i in range(num_features):
    row = i // cols + 1
    col = i % cols + 1
    fig.add_trace(
        go.Heatmap(
            z=som.get_weights()[:, :, i],
            colorscale="plasma",  # Palette de couleurs
            colorbar=dict(
                title=f"Feature {i+1}",
                titleside="right",
                titlefont=dict(size=10),
                tickfont=dict(size=8),
                len=0.4,
                thickness=10,
                xpad=20,
                ypad=10,
            ),
            showscale=(col == cols),
        ),
        row=row,
        col=col,
    )

fig.update_layout(
    title="Codebooks pour chaque Feature (SOM)",
    height=250 * rows,
    width=1200,
    showlegend=False,
    template="plotly_white",
)
fig.show()
```



Les matrices ci-dessus montrent des zones où les valeurs sont fortement activées (en jaune). Cela montre que certains ports sont particulièrement représentatifs des données associées à ces neurones, reflétant des comportements spécifiques, comme des attaques ciblant des ports particuliers ou des durées anormalement longues dans les flux.
Contrairement aux autres matrices, la caractéristique RST Flag Count semble uniformément distribuée, ce qui pourrait indiquer qu'elle a peu d'impact discriminant dans l'organisation des clusters ou qu'elle est moins représentée dans les données.

On le confirme en affichant une description statistique incluant moyenne, variance et écart-type de l'image :

```
print(data["RST Flag Count"].describe())
```

```
count    283077.000000
mean          0.000187
std           0.013682
min           0.000000
25%           0.000000
50%           0.000000
75%           0.000000
max           1.000000
Name: RST Flag Count, dtype: float64
```

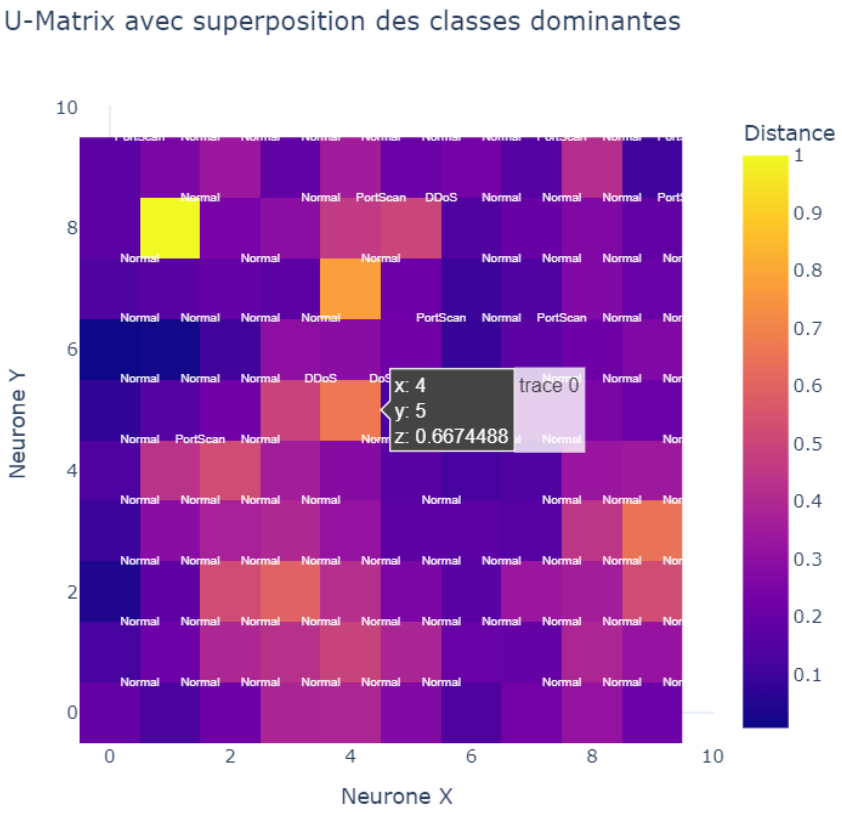
5.4.4 U-matrix

La U-Matrix (Unified Distance Matrix) est une représentation visuelle qui met en évidence les distances entre les neurones adjacents d'une carte auto-organisée (SOM). Elle permet de comprendre la structure des clusters formés par les données. Les couleurs de la matrice représentent ces distances : les zones bleu foncé indiquent des distances faibles entre les neurones, ce qui signifie que les poids des neurones voisins sont similaires et appartiennent probablement au même cluster. À l'inverse, les zones jaunes ou oranges vives représentent des distances élevées, correspondant à des transitions entre différents clusters.

On observe une prépondérance de la classe "Normal" sur une majorité des neurones, ce qui est attendu étant donné l'importance de cette classe dans le dataset. Cependant, des neurones comme ceux associés à "PortScan", "DDoS" ou "DoS" apparaissent comme des zones spécifiques où ces types d'attaques dominent. Ces zones peuvent indiquer que le SOM a réussi à séparer ces types d'anomalies des données normales.

```
# Calcul de la U-Matrix
u_matrix = som.distance_map()  # Distance entre les poids des neurones adjacents

# Initialisation de la figure
fig = go.Figure()
# Ajout de la U-Matrix en tant que Heatmap
fig.add_trace(
    go.Heatmap(
        z=u_matrix,
        colorscale="plasma",
        colorbar=dict(title="Distance"),
        showscale=True,
    )
)
# Superposition des valeurs des classes dominantes
for x in range(u_matrix.shape[0]):  # Parcourt les neurones SOM
    for y in range(u_matrix.shape[1]):
        # pour obtenir la classe dominante pour chaque neurone (si elle existe)
        valeur = bmu_to_class.get((x, y), None)
        if valeur is not None:  # Si une classe est associée au neurone
            fig.add_trace(
                go.Scatter(
                    x=[y + 0.5],  # Décalage pour centrer dans la cellule
                    y=[x + 0.5],
                    text=[label_encoder.inverse_transform([valeur])[0]],
```

```
                    mode="text",
                    textfont=dict(color="white", size=8, family="Arial",),
                    showlegend=False,)
            )

fig.update_layout(
    title="U-Matrix avec superposition des classes dominantes",
    xaxis=dict(title="Neurone X", showgrid=False),
    yaxis=dict(title="Neurone Y", showgrid=False),
    height=600,
    width=600,
    template="plotly_white",
)
fig.show()
```



Les zones homogènes, où les couleurs sont uniformes (souvent bleu foncé ou violet), indiquent des clusters bien définis et homogènes. Ces zones regroupent des neurones associés à des données similaires, telles que des points appartenant à une même classe (par exemple, Normal). En revanche, les zones de transition, souvent en jaune ou orange vif, marquent les frontières entre différents clusters. Les neurones dans ces régions représentent des données limites ou potentiellement des anomalies.

Dans notre contexte de détection d'IDS, on peut l'utiliser pour détecter les clusters dominants et analyser les transitions entre eux. Les zones homogènes correspondent aux classes bien représentées (par exemple, Normal ou DDoS), tandis que les zones de transition peuvent révéler des anomalies ou des cas limites.

5.5 Conclusion

La méthode basée sur les Self-Organizing Maps (SOM) appliquée à ce projet a permis d'organiser efficacement les données en clusters distincts, tout en identifiant les zones où les anomalies

se distinguent clairement des comportements normaux. L'analyse des codebooks a mis en évidence les caractéristiques spécifiques de chaque feature, révélant les patterns qui différencient les types de comportements réseau. La U-Matrix a quant à elle illustré les distances entre les neurones du SOM, soulignant les frontières entre clusters et confirmant une bonne séparation des classes, notamment entre les données normales et les principales anomalies telles que "DDoS", "DoS" et "PortScan".

Les résultats montrent une précision notable pour la détection de comportements normaux (f1-score de 0.94) et une performance variable pour les anomalies, les attaques comme "DDoS" et "PortScan" étant mieux détectées grâce à leurs signatures distinctes. Cependant, certaines classes, comme "WebAttack" ou "Infiltration", ont été peu ou mal représentées, en raison d'un faible nombre d'échantillons ou de leurs similarités avec d'autres comportements.

L'ajout d'intervalles de confiance basés sur les distances aux BMU (Best Matching Units) et aux centroïdes a permis d'affiner la détection des anomalies. Ces seuils, définis respectivement à 95 % et 99 %, ont offert un contrôle plus rigoureux sur la classification des données, réduisant ainsi les risques de faux positifs et de faux négatifs.

En conclusion, cette approche a démontré son utilité dans la classification et la détection des anomalies dans des données réseau complexes. Bien que les résultats puissent encore être optimisés pour certaines classes rares, la méthode est robuste et offre une vision claire des structures sous-jacentes des données, permettant de mieux comprendre les relations entre les différentes caractéristiques.

Chapitre 6

Deep Belief Networks (Item 6, supplémentaire)

Jeu de données

Dans cette section, j'ai réutilisé le dataset **CICIDS2017** décrit en 5.3.

6.1 Implémentation des Réseaux de Croyances Profonds

Librairies implémentées

```
# Importation des librairies utilisées
import pandas as pd
import numpy as np
from collections import Counter
from sklearn.preprocessing import QuantileTransformer, LabelEncoder
from imblearn.over_sampling import RandomOverSampler, SMOTE
from sklearn.utils.class_weight import compute_class_weight
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix, f1_score
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import seaborn as sns
import os
```

Étape 1 : Prétraitement des données Dans cette étape, j'ai nettoyé et équilibré les données avec la fonction `SMOTE`. Ces étapes incluent :

- Le remplacement des valeurs infinies par un maximum raisonnable
- Le remplacement des valeurs manquantes (`NaN`) par la médiane
- La normalisation des données avec un transformateur quantile
- L'utilisation de `RandomOverSampler` et `SMOTE` pour gérer le déséquilibre des classes
- La réduction des échantillons par classe pour uniformiser la distribution

```
X = data.drop(columns=['Label'])
# Remplacement des valeurs infinies par NaN
X.replace([np.inf, -np.inf], np.nan, inplace=True)
# Puis remplacement des NaN par la médiane de chaque colonne
X.fillna(X.median(), inplace=True)
y = data['Label']
```

```
quantile_transformer = QuantileTransformer(output_distribution='uniform', random_state=42)
X_normalized = quantile_transformer.fit_transform(X)

# Gestion des déséquilibres avec RandomOverSampler et SMOTE
ros = RandomOverSampler(random_state=42)
X_ros, y_ros = ros.fit_resample(X_normalized, y)
smote = SMOTE(random_state=42, k_neighbors=5)
X_balanced, y_balanced = smote.fit_resample(X_ros, y_ros)

# Sous-échantillonnage
balanced_data = pd.DataFrame(X_balanced)
balanced_data['Label'] = y_balanced

# Limiter à 50 000 échantillons par classe
max_samples_per_class = 20000
reduced_data = balanced_data.groupby('Label').apply(
    lambda x: x.sample(n=max_samples_per_class, random_state=42)
).reset_index(drop=True)

X_reduced = reduced_data.drop(columns=['Label'])
y_reduced = reduced_data['Label']
print("Distribution des classes après réduction :", Counter(y_reduced))

# Séparation des données
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y_reduced, test_size=0.2,
random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y_reduced, test_size=0.2,
random_state=42)

# Encodage des labels
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)

X_train_array = X_train.values if isinstance(X_train, pd.DataFrame) else X_train
X_test_array = X_test.values if isinstance(X_test, pd.DataFrame) else X_test
X_train_tensor = torch.FloatTensor(X_train_array)
y_train_tensor = torch.LongTensor(y_train_encoded)
X_test_tensor = torch.FloatTensor(X_test_array)
y_test_tensor = torch.LongTensor(y_test_encoded)
```

Étape 2 : Architecture du modèle Le modèle est construit en utilisant une architecture Deep Belief Network (DBN), une forme de réseau de neurones composés de couches restreintes de machines de Boltzmann :

```
class DBN(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim):
        super(DBN, self).__init__()
        self.rbm_layers = nn.ModuleList([
            nn.Linear(input_dim if i == 0 else hidden_dims[i-1], hidden_dims[i])
            for i in range(len(hidden_dims))])
        self.output_layer = nn.Linear(hidden_dims[-1], output_dim)

    def forward(self, x):
        for rbm in self.rbm_layers:
            x = torch.sigmoid(rbm(x))  # Activation sigmoid
        x = self.output_layer(x)
        return x
```

```
hidden_layers = [128, 256, 128, 64]
dbn = DBN(input_dim=X_train.shape[1], hidden_dims=hidden_layers,
output_dim=len(label_encoder.classes_))
```

Étape 3 : Gestion des déséquilibres On calcule les poids pour compenser le déséquilibre des classes Dans l'application des cartes de Kohonen précédente, j'ai observé via l'évaluation du modèle qu'il semblait avoir un fort effet du nombre de données dans chaque classe sur la performance de classification (voir 5.3). Ainsi, avant d'entraîner ce DBN, j'ai commencé par compenser les déséquilibres dans les classes :

```
# Calcul du poids de classe pour compenser les déséquilibres
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight('balanced', classes=np.unique(y_train_encoded),
y=y_train_encoded)
class_weights_tensor =
torch.FloatTensor(class_weights).to('cuda' if torch.cuda.is_available() else 'cpu')
```

Étape 4 : Entraînement Entraînement du modèle sur 50 époques, avec des checkpoints pour sauvegarder l'état :

```
# Définir la fonction de perte et l'optimiseur
critere = nn.CrossEntropyLoss(weight=class_weights_tensor)
optimizer = optim.Adam(dbn.parameters(), lr=0.001)

checkpoint_path = "path\\2dbn_checkpoint.pth"

# Chargement du checkpoint (s'il est disponible, me permet d'entraîner
# le modèle en plusieurs fois, pour atteindre un nombre d'epochs conséquent)
start_epoch = 0
if os.path.exists(checkpoint_path):
    print("Checkpoint trouvé. Chargement...")
    checkpoint = torch.load(checkpoint_path)
    dbn.load_state_dict(checkpoint['model_state_dict'])
    optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
    start_epoch = checkpoint['epoch']
    print(f"Reprise à l'époque {start_epoch}")

# Entraînement du modèle
epochs = 50
for epoch in range(start_epoch, epochs):
    dbn.train()
    optimizer.zero_grad()
    outputs = dbn(X_train_tensor)
    loss = critere(outputs, y_train_tensor)
    loss.backward()
    optimizer.step()

    # Évaluation sur l'ensemble de test à chaque époque
    dbn.eval()
    with torch.no_grad():
        y_pred = torch.argmax(dbn(X_test_tensor), axis=1)
        f1 = f1_score(y_test_tensor.numpy(), y_pred.numpy(), average='weighted')
        print(f"Epoch {epoch+1}/{epochs} - Loss: {loss.item():.4f} - Weighted F1: {f1:.4f}")
```

```
    # Sauvegarde d'un checkpoint tous les 10 epochs
    if (epoch + 1) % 10 == 0 or epoch + 1 == epochs:
        torch.save({
            'epoch': epoch + 1, 'model_state_dict': dbn.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': loss.item()}, checkpoint_path)
        print(f"Checkpoint sauvegardé à l'époque {epoch + 1} dans '{checkpoint_path}'")
```

On peut suivre l'évolution des pertes :

```
Epoch 1/50 - Loss: 2.0840 - Weighted F1: 0.0289
Epoch 2/50 - Loss: 2.0836 - Weighted F1: 0.0289
Epoch 3/50 - Loss: 2.0830 - Weighted F1: 0.0289
Epoch 4/50 - Loss: 2.0822 - Weighted F1: 0.0289
Epoch 5/50 - Loss: 2.0813 - Weighted F1: 0.0289
Epoch 6/50 - Loss: 2.0806 - Weighted F1: 0.0289
Epoch 7/50 - Loss: 2.0800 - Weighted F1: 0.0289
Epoch 8/50 - Loss: 2.0796 - Weighted F1: 0.0719
Epoch 9/50 - Loss: 2.0793 - Weighted F1: 0.0274
Epoch 10/50 - Loss: 2.0793 - Weighted F1: 0.0274
Checkpoint sauvegardé à l'époque 10 dans 'C:\Users\beriv\Downloads\CICIDS2017\2dbn_checkpoint.pth'
Epoch 11/50 - Loss: 2.0793 - Weighted F1: 0.0274
Epoch 12/50 - Loss: 2.0794 - Weighted F1: 0.0274
Epoch 13/50 - Loss: 2.0795 - Weighted F1: 0.0274
Epoch 14/50 - Loss: 2.0796 - Weighted F1: 0.0287
Epoch 15/50 - Loss: 2.0796 - Weighted F1: 0.0287
Epoch 16/50 - Loss: 2.0796 - Weighted F1: 0.0287
Epoch 17/50 - Loss: 2.0796 - Weighted F1: 0.0287
Epoch 18/50 - Loss: 2.0795 - Weighted F1: 0.0287
Epoch 19/50 - Loss: 2.0793 - Weighted F1: 0.0287
Epoch 20/50 - Loss: 2.0791 - Weighted F1: 0.0287
Checkpoint sauvegardé à l'époque 20 dans 'C:\Users\beriv\Downloads\CICIDS2017\2dbn_checkpoint.pth'
Epoch 21/50 - Loss: 2.0789 - Weighted F1: 0.0287
Epoch 22/50 - Loss: 2.0787 - Weighted F1: 0.0287
Epoch 23/50 - Loss: 2.0785 - Weighted F1: 0.0287
...
Epoch 48/50 - Loss: 2.0622 - Weighted F1: 0.1009
Epoch 49/50 - Loss: 2.0596 - Weighted F1: 0.1011
Epoch 50/50 - Loss: 2.0566 - Weighted F1: 0.1013
Checkpoint sauvegardé à l'époque 50 dans 'C:\Users\beriv\Downloads\CICIDS2017\2dbn_checkpoint.pth'
```

Figure 6.1 : Évolution des pertes au fil des époques

Étape 5 : Evaluation du modèle Finalement, évaluons notre modèle.

```
dbn.eval()
# Prédictions
with torch.no_grad():
    y_pred = torch.argmax(dbn(X_test_tensor), axis=1)

accuracy = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
print(f"Accuracy finale : {accuracy:.4f}")

# Rapport de classification
print("\nClassification Report:\n")
print(classification_report(y_test_tensor.numpy(), y_pred.numpy(),
target_names=label_encoder.classes_))
```

```
Accuracy finale : 0.2508

Classification Report:

              precision    recall  f1-score   support

         Bot       0.00      0.00      0.00      3861
  BruteForce       1.00      0.00      0.00      3970
        DDoS       0.00      0.00      0.00      3961
         DoS       0.00      0.00      0.00      4082
Infiltration       0.23      1.00      0.37      3970
      Normal       0.00      0.00      0.00      4098
    PortScan       0.27      1.00      0.43      4069
   WebAttack       0.00      0.00      0.00      3989

    accuracy                           0.25     32000
   macro avg       0.19      0.25      0.10     32000
weighted avg       0.19      0.25      0.10     32000
```

Figure 6.2 : Rapport de classification du modèle DBN

```
# Matrice de confusion
conf_mat = confusion_matrix(y_test_tensor.numpy(), y_pred.numpy())
plt.figure(figsize=(7, 5))
sns.heatmap(conf_mat, annot=True, fmt='d', cmap='Blues',
    xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.xlabel('Prédictions')
plt.ylabel('Vraies valeurs')
plt.title('Matrice de confusion finale')
plt.show()
```

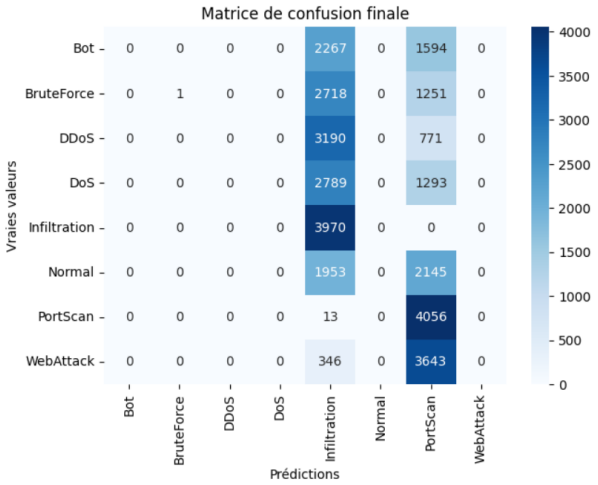


Figure 6.3 : Matrice de confusion du modèle DBN

Résultats

Performance globale : La précision finale du modèle est de seulement 25.08%.
Le `Weighted F1-score` est également faible (0.10), indiquant une mauvaise performance sur des classes déséquilibrées.

Rapport de classification : Les classes comme `BruteForce` et `PortScan` ont des performances significatives en termes de rappel, mais de nombreuses autres classes, comme `Bot`, `DDoS`, et `WebAttack`, ont des scores de 0 pour toutes les métriques. Cela reflète un problème important de déséquilibre ou

de confusion entre certaines classes.

Matrice de confusion : La matrice montre une confusion importante pour des classes comme `Bot` et `DDoS`, avec des prédictions concentrées dans quelques catégories (e.g., `PortScan`).

Améliorations possibles

Plusieurs méthodes d'amélioration seraient envisageables sur le modèle :

- Augmenter la taille des données pour les classes minoritaires
- Essayer une autre architecture de modèle, comme un CNN ou transformer-based model, qui pourrait mieux capturer les patterns complexes
- Ajuster davantage les hyperparamètres du modèle, comme les taux d'apprentissage ou les fonctions d'activation

Chapitre 7

Éthique

L'avènement des réseaux de neurones dans les systèmes de détection d'intrusion (IDS) représente une avancée technologique majeure dans le domaine de la cybersécurité. Cependant, ces avancées posent également des questions éthiques complexes, notamment en ce qui concerne la protection de la vie privée, la responsabilité des acteurs impliqués et les conséquences sociétales d'une automatisation accrue de la surveillance. Dans cette section, nous explorerons les dimensions éthiques associées à l'utilisation des réseaux de neurones pour les IDS, en nous appuyant sur des perspectives philosophiques et techniques afin de souligner les défis et les opportunités.

Éthique des réseaux de neurones dans les systèmes d'information et de communication

L'utilisation des réseaux de neurones dans les systèmes d'information et de communication représente une avancée majeure, mais soulève également des questions éthiques complexes. Ces systèmes, qui incluent des API, des infrastructures cloud, et des plateformes de gestion de données, jouent un rôle crucial dans la connectivité et la transmission d'informations à l'échelle mondiale. Cependant, leur conception et leur déploiement posent des défis éthiques relatifs à la vie privée, à la responsabilité des acteurs impliqués, et aux impacts sociétaux d'une automatisation croissante. Dans cette section, nous analyserons ces enjeux en adoptant une perspective philosophique et technique pour en souligner l'importance et les conséquences potentielles.

Vie privée et souveraineté des données

Les systèmes d'information et de communication traitent un volume immense de données personnelles et professionnelles, rendant leur gestion éthique essentielle. Les réseaux de neurones, avec leur capacité à analyser des motifs complexes, permettent d'améliorer l'efficacité des systèmes de communication, mais peuvent également être exploités pour surveiller les comportements, profiler les utilisateurs et déceler des informations sensibles.

La collecte et l'analyse de ces données sans consentement explicite peuvent entraîner une violation des droits à la vie privée et à la souveraineté des données. Selon l'éthique kantienne, il est impératif de traiter les individus comme des fins en soi et non comme des moyens pour atteindre des objectifs commerciaux ou de surveillance. Kant affirme que « l'homme, et en général tout être raisonnable, existe comme fin en soi, et non pas simplement comme moyen dont telle ou telle volonté puisse user à son gré ». En revanche, une perspective utilitariste pourrait justifier de telles pratiques si elles bénéficient à la majorité. Ce dilemme exige une gouvernance éthique rigoureuse, incluant des règles de transparence, des protocoles de consentement clair, et des audits indépendants des systèmes.

Responsabilité et imputabilité

Avec l'automatisation croissante permise par les réseaux de neurones, la responsabilité devient un enjeu central. Lorsqu'une décision erronée est prise par un système automatisé, par exemple lors d'une analyse erronée des flux d'information, il est difficile de déterminer qui est responsable.
L'éthique de la responsabilité exige que les créateurs technologiques anticipent les conséquences de leurs innovations et prennent des mesures pour minimiser les risques. Dans les systèmes d'information et de communication, cela implique de développer des **systèmes explicables** permettant aux utilisateurs de comprendre les décisions prises et de remédier aux erreurs potentielles. Sans une responsabilité claire, la confiance dans ces systèmes risque de s'effriter, compromettant leur adoption et leur utilité.

Impacts sociétaux de l'automatisation

Les systèmes d'information et de communication automatisés transforment fondamentalement la façon dont les individus interagissent et accèdent aux services. Si ces systèmes promettent d'améliorer l'efficacité et la rapidité des communications, ils risquent également de créer des inégalités, de réduire les opportunités d'emploi dans certains secteurs, et d'augmenter la dépendance à des infrastructures technologiques vulnérables.

Énergie et durabilité

Les systèmes d'information et de communication, alimentés par des infrastructures comme les data centers et les réseaux de transmission, consomment d'énormes quantités d'énergie. Les réseaux de neurones, bien qu'efficaces, ajoutent une charge supplémentaire à ces infrastructures, augmentant leur empreinte carbone. Une approche éthique dans la conception et l'utilisation de ces systèmes implique de minimiser leur impact environnemental. Cela peut inclure l'optimisation des modèles pour réduire leur consommation énergétique, l'utilisation d'énergies renouvelables pour alimenter les infrastructures, et la sensibilisation à la nécessité d'une sobriété numérique. Ignorer ces aspects pourrait exacerber la crise climatique et remettre en question la légitimité éthique de ces innovations.

Conclusion

Les réseaux de neurones dans les systèmes d'information et de communication offrent des opportunités significatives pour améliorer la connectivité et l'efficacité des infrastructures numériques. Cependant, leur déploiement s'accompagne de responsabilités éthiques importantes. Pour garantir que ces technologies bénéficient à la société tout en minimisant les risques, il est essentiel d'adopter une approche prudente, transparente et durable, en alignant les objectifs technologiques sur les valeurs fondamentales de la vie privée, de la justice sociale et de la durabilité environnementale.

Bibliographie

Bensaoud, M. and Benidir, M. (2021), 'An efficient network behavior anomaly detection using a hybrid dbn-lstm network', *ResearchGate* .

Gang, W. (2025), 'Demystifying anomaly detection with autoencoder neural networks'.
URL: *https ://medium.com/@weidagang/demystifying-anomaly-detection-with-autoencoder-neural-networks-1e235840d879*

Hormann, R. and Fischer, E. (2019), Detecting anomalies by using self-organizing maps in industrial environments, pp. 336–344.

Linder-Noren, E. (2019), 'Pytorch-gan : Bicyclegan model implementation'.
URL: *https ://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/ bicyclegan/models.py*

Ma, L., Zheng, H., Zhang, L., Chen, P.-Y., Chen, S. and Chien, J.-T. (2022), 'Gan-powered anomaly detection for images of pills and capsules', *arXiv preprint arXiv :2207.02117* .

Meriläinen, V. (2023), 'Save and load models to disk in pytorch python : A complete guide'.
URL: *https ://medium.com/@merilainen.vili/save-and-load-models-to-disk-in-pytorch-python-a-complete-guide-a667057b511c*

Rakeshkumar Mahto, K. S. (2023), 'Neural network architecture illustrating the layers and connections used for shade prediction'.
URL: *https ://www.researchgate.net/figure/Neural-network-architecture-illustrating-the-layers-and-connections-used-for-shade_fig2_376695090*

Schlueter, N. (2025), 'Self-organizing maps for anomaly detection'.
URL: *https ://www.kaggle.com/code/nilsschlueter/self-organizing-maps-for-anomaly-detection*

Schuster, A. et al. (2013), 'Index of contrast sensitivity (ics) in pseudophakic eyes with different intraocular lens designs', *Acta Ophthalmologica* **91**, e188–e192.

Seth, N. (2021), 'Beginners guide to anomaly detection using self-organizing maps'.
URL: *https ://www.analyticsvidhya.com/blog/2021/09/beginners-guide-to-anomaly-detection-using-self-organizing-maps/*

Sharma, S. (2022), 'Coding k-means clustering using python and numpy'.
URL: *https ://dev.to/sajal2692/coding-k-means-clustering-using-python-and-numpy-fg1*