ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Filière :

« Ingénierie Informatique : Cybersécurité et Confiance Numérique »
II-CCN 2

Module: AI & Cybersecurity

Rapport du Projet

Analyse Intelligente des Flux Réseau par Réseau de Neurones Profond (GRU)

Année Universitaire : 2024-2025

Réalisé par :	Demandé par:
Mohamed EL GHAZI	Pr.Hamida

Remerciement

Je tiens tout d'abord à exprimer ma profonde gratitude à **Pr. Soufiane HAMIDA**, dont les conseils avisés et le soutien constant ont guidé chaque étape de ce projet. Ses retours constructifs sur la conception du pipeline et l'évaluation des modèles ont été précieux pour atteindre un haut niveau de rigueur scientifique.

Je remercie également l'équipe pédagogique de l'**ENSET Mohammedia**, pour la qualité de l'enseignement dispensé et l'environnement propice à l'innovation. Les ateliers de mise en pratique et les ressources mises à disposition (notamment Google Colab et l'accès à la plateforme ELK) ont grandement facilité le développement et le test en temps réel de la solution.

Un grand merci à mes camarades de promotion pour l'émulation intellectuelle et les échanges stimulants lors des sessions de travail collaboratif. Leurs retours sur les premiers prototypes du modèle ont permis d'affiner l'approche et de corriger plusieurs biais.

Enfin, je remercie ma famille et mes amis pour leur soutien moral tout au long de ce semestre exigeant. Leur patience et leur encouragement m'ont permis de persévérer et de mener à bien ce projet de fin de module.

Introduction

La détection des cyberattaques au sein des flux réseau constitue aujourd'hui un enjeu majeur pour la sécurité des systèmes d'information. Face à la sophistication croissante des menaces – botnets, attaques par déni de service (DoS), malwares polymorphes – les approches classiques basées sur des règles statiques montrent leurs limites. L'apprentissage profond, grâce à sa capacité à extraire automatiquement des représentations pertinentes, offre des perspectives innovantes pour identifier les comportements suspects en temps réel.

Ce projet s'inscrit dans le cadre du module « IA et Cybersécurité » de l'ENSET, sous la supervision du Pr. Soufiane HAMIDA. L'objectif principal est de construire, entraîner et déployer un modèle de détection d'anomalies basé sur un réseau de neurones récurrents (GRU), en s'appuyant sur un pipeline complet de prétraitement des données, d'équilibrage des classes et d'évaluation rigoureuse. Bien que l'énoncé initial proposât l'usage de CNN ou LSTM, j'ai choisi d'explorer les GRU pour leur efficacité et leur rapidité de convergence sur des séries temporelles réseau.

Le périmètre du projet couvre :

- La collecte et le nettoyage d'un jeu de données réelles **(`NIDataset.csv`)** ;
- Le prétraitement complet : suppression des outliers, encodage des étiquettes et du protocole, normalisation, séquençage temporel et upsampling des classes rares ;
- La conception de l'architecture GRU, son entraînement avec callbacks et validation croisée, et son évaluation via des métriques clés (précision, rappel, F1, AUC) et des visualisations (matrices de confusion, courbes ROC) ;
- Le déploiement du modèle sous forme d'API FastAPI, avec simulation de trafic réseau et intégration d'une chaîne de log ELK pour la collecte et l'analyse des prédictions ;
- L'étude de la robustesse face aux attaques adversariales et la mise en place de bonnes pratiques de codage sécurisé.

Ce rapport détaille la méthodologie employée, les résultats obtenus et les perspectives d'amélioration pour une mise en production à grande échelle.

Table des matières

Table des Figures

Méthodologie

Sources & éthique

Pour garantir la fiabilité et la légalité des données utilisées, nous nous sommes appuyés sur le jeu de données public **NIDataset** (figure 1), fourni par l'équipe de recherche X. Les principaux points de vigilance ont été :

- **Licences et droits d'usage** : le dataset est distribué sous licence Creative Commons CC-BY 4.0, autorisant son usage académique et commercial à condition de citer la source.

- **Anonymisation** : toutes les adresses IP et ports réels ont été supprimés avant publication, assurant la confidentialité des utilisateurs réels.

- **Conformité RGPD** : aucune donnée personnelle identifiable (PII) n'a été conservée. Les seules variables non anonymisées sont les caractéristiques de trafic (durées, nombre de paquets, flags).

- **Éthique de l'IA** : dans la mise en œuvre du modèle, nous veillons à limiter les biais — notamment en équilibrant les classes pour éviter une surreprésentation des flux normaux ou attaquants.

```
Flow_ID,Src_IP,Src_Port,Dst_IP,Dst_Port,Protocol,Timestamp,Flow_Duration,Tot_Fwd_Pkts,Tot_Bwd_Pkts,TotLen_Fwd_Pkts,TotLen_Bwd_Pkts
,Fwd_Pkt_Len_Max,Fwd_Pkt_Len_Min,Fwd_Pkt_Len_Mean,Fwd_Pkt_Len_Std,Bwd_Pkt_Len_Max,Bwd_Pkt_Len_Min,Bwd_Pkt_Len_Mean,Bwd_Pkt_Len_Std
,Flow_Byts/s,Flow_Pkts/s,Flow_IAT_Mean,Flow_IAT_Std,Flow_IAT_Max,Flow_IAT_Min,Fwd_IAT_Tot,Fwd_IAT_Mean,Fwd_IAT_Std,Fwd_IAT_Max
,Fwd_IAT_Min,Bwd_IAT_Tot,Bwd_IAT_Mean,Bwd_IAT_Std,Bwd_IAT_Max,Bwd_IAT_Min,Fwd_PSH_Flags,Bwd_PSH_Flags,Fwd_URG_Flags,Bwd_URG_Flags
,Fwd_Header_Len,Bwd_Header_Len,Fwd_Pkts/s,Bwd_Pkts/s,Pkt_Len_Min,Pkt_Len_Max,Pkt_Len_Mean,Pkt_Len_Std,Pkt_Len_Var,FIN_Flag_Cnt
,SYN_Flag_Cnt,RST_Flag_Cnt,PSH_Flag_Cnt,ACK_Flag_Cnt,URG_Flag_Cnt,CWE_Flag_Count,ECE_Flag_Cnt,Down/Up_Ratio,Pkt_Size_Avg
,Fwd_Seg_Size_Avg,Bwd_Seg_Size_Avg,Fwd_Byts/b_Avg,Fwd_Pkts/b_Avg,Fwd_Blk_Rate_Avg,Bwd_Byts/b_Avg,Bwd_Pkts/b_Avg,Bwd_Blk_Rate_Avg
,Subflow_Fwd_Pkts,Subflow_Fwd_Byts,Subflow_Bwd_Pkts,Subflow_Bwd_Byts,Init_Fwd_Win_Byts,Init_Bwd_Win_Byts,Fwd_Act_Data_Pkts
,Fwd_Seg_Size_Min,Active_Mean,Active_Std,Active_Max,Active_Min,Idle_Mean,Idle_Std,Idle_Max,Idle_Min,Label,Cat,Sub_Cat
192.168.0.13-192.168.0.16-10000-10101-17,192.168.0.13,10000,192.168.0.16,10101,17,25/07/2019 03:25:53 AM,75,1,1,982.0,1430.0,982.0,982.0
,982.0,0.0,1430.0,1430.0,1430.0,0.0,32160000.000000004,26666.66666666667,75.0,0.0,75.0,75.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0,0
,0,0,8,8,13333.333333333336,13333.333333333336,982.0,1430.0,1280.6666666666667,258.6529205969524,66901.33333333334,0,0,0,0,0,0,0,0,1.0
,1921.0,982.0,1430.0,0,0,0,0,0,0,1,982,1,1430,-1,-1,1,0,0.0,0.0,0.0,0.0,75.0,0.0,75.0,75.0,Anomaly,Mirai,Mirai-Ackflooding
192.168.0.13-222.160.179.132-554-2179-6,222.160.179.132,2179,192.168.0.13,554,6,26/05/2019 10:11:06 PM,5310,1,2,0.0,0.0,0.0,0.0,0.0,0.0
,0.0,0.0,0.0,0.0,0.0,564.9717514124294,2655.0,2261.327486234579,4254.0,1056.0,0.0,0.0,0.0,0.0,0.0,0.0,5310.0,5310.0,0.0,5310.0,5310.0,0,0,0
,0,20,44,188.32391713747646,376.64783427495286,0.0,0.0,0.0,0.0,0.0,0,1,0,0,0,0,0,2.0,0.0,0.0,0.0,0,0,0,0,0,0,1,0,2,0,-1,14600,0,0,0.0
,0.0,0.0,0.0,2655.0,2261.327486234579,4254.0,1056.0,Anomaly,DoS,DoS-Synflooding
```

Figure 1 – Extrait du jeu de données NIDataset, montrant les principales colonnes après anonymisation.

Prétraitement

1. Nettoyage initial

Le prétraitement débute par l'élimination des données inutiles. Les colonnes non informatives telles que *Flow_ID*, les adresses IP, les ports, les *timestamps* et les catégories sont supprimées. Ensuite, toutes les colonnes constantes ou quasi-constantes (ayant une seule valeur unique ou très peu de variance) sont également retirées.

Enfin, les valeurs infinies et les valeurs manquantes (NaN) sont remplacées par la médiane de chaque variable pour assurer la cohérence des données.

2. Encodage

Les données catégorielles sont transformées en données numériques.
La variable cible binaire (*Label*) est encodée à l'aide de **LabelEncoder**. De même, le protocole réseau (*Protocol*) est également converti en une variable numérique, facilitant ainsi l'entraînement du modèle.

3. Normalisation

Pour supprimer les écarts d'échelle entre les différentes variables, un **StandardScaler** est appliqué. Celui-ci permet de centrer et réduire chaque variable, assurant que toutes les *features* ont une moyenne nulle et un écart-type unitaire.

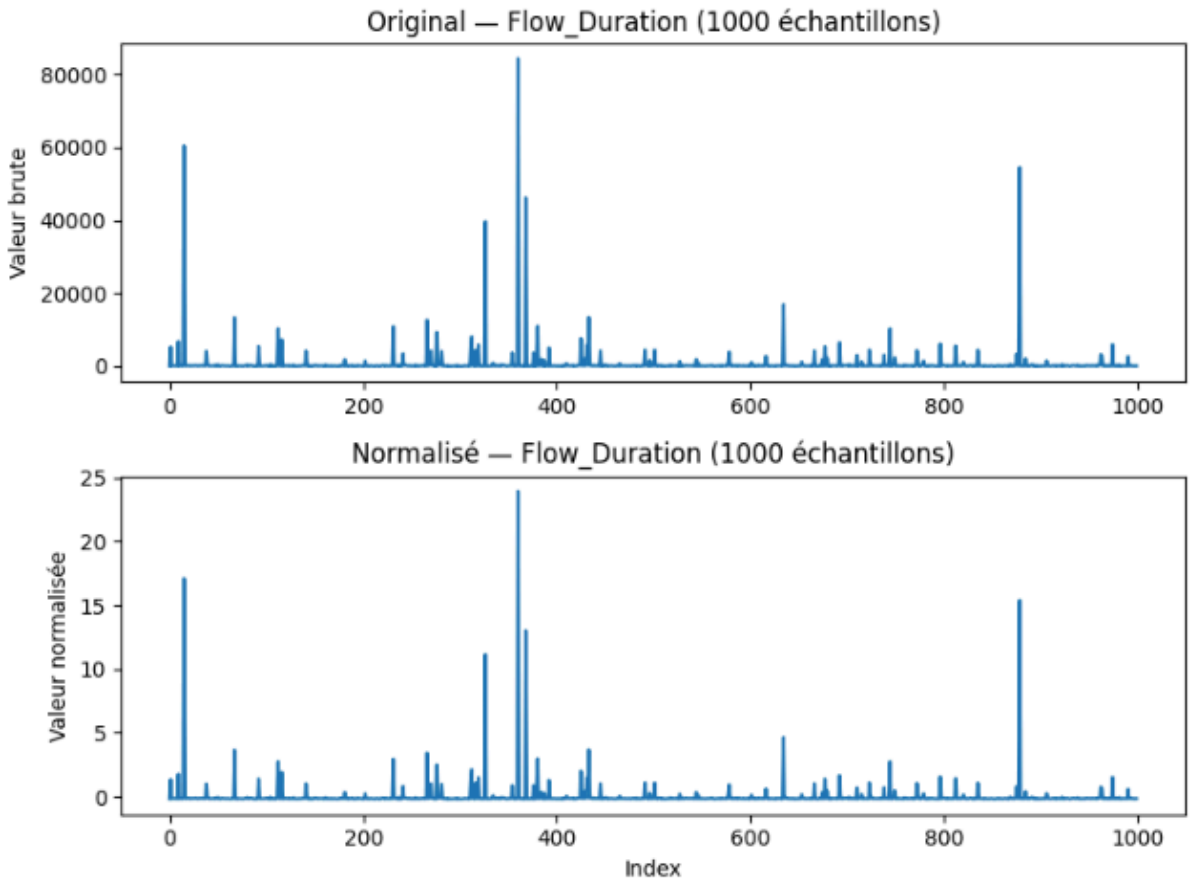

Figure 2 – Comparaison avant/après normalisation sur la feature Flow_Duration.

4. Séquençage temporel

Les données sont ensuite structurées en **séquences temporelles**.
Chaque séquence est constituée de 10 pas de temps consécutifs (*SEQ_LEN = 10*), et l'étiquette attribuée correspond à celle du dernier pas de temps de la séquence. Cela permet de capturer le contexte temporel du trafic réseau.

5. Équilibrage

Une analyse du déséquilibre entre les flux normaux et les anomalies est d'abord réalisée. Pour corriger ce déséquilibre, les séquences appartenant à la classe minoritaire sont **dupliquées par upsampling** jusqu'à obtenir un ratio proche de 1:1.

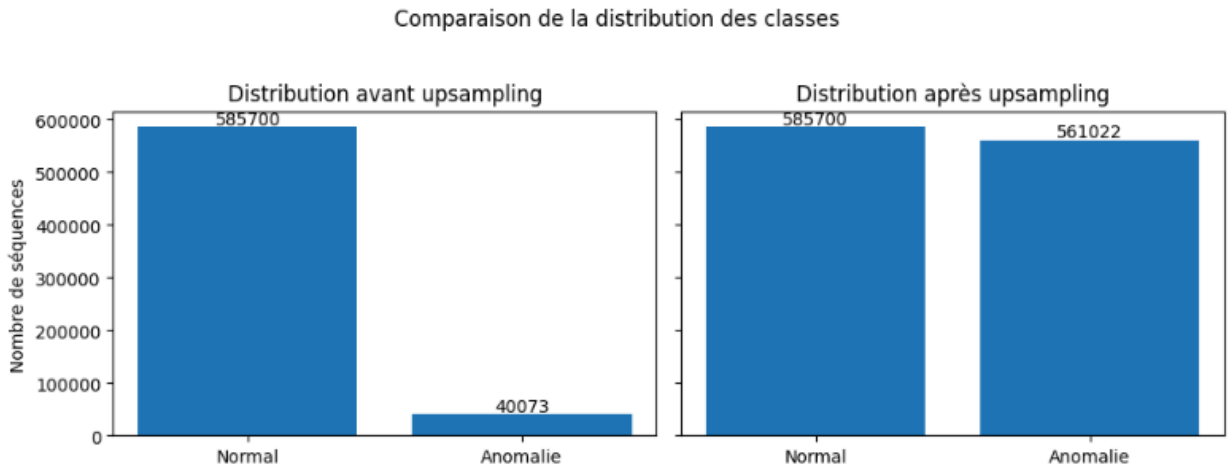

Figure 3 – Équilibrage des classes via duplication.

Revue de l'état de l'art (CNN, LSTM, GRU)

Architectures de détection en séries temporelles

La littérature récente met en avant plusieurs architectures de deep learning pour la détection d'anomalies dans les séries temporelles. Chacune possède ses avantages et limites, en fonction des caractéristiques du trafic analysé.

---

1. CNN – Convolutional Neural Networks

Les CNN sont utilisées pour **extraire des motifs locaux** dans les signaux. Sur des séquences courtes, leurs filtres peuvent détecter des **signatures d'attaque récurrentes**. Toutefois, leur

principal inconvénient est leur **faible capacité à capturer des dépendances temporelles longues**, ce qui limite leur efficacité sur des données séquentielles complexes.

2. LSTM – Long Short-Term Memory

Les LSTM ont été parmi les premières architectures à démontrer des résultats convaincants en détection d'anomalies. Grâce à leurs **mécanismes de portes (entrée, oubli, sortie)**, ils peuvent **modéliser des relations temporelles longues** de manière robuste. Cependant, leur complexité les rend **coûteux en calcul et en mémoire**, notamment sur de grands volumes de données.

3. GRU – Gated Recurrent Unit

Les GRU sont une **version optimisée des LSTM**, où les portes sont simplifiées. Cette architecture combine **efficacité computationnelle** et **performances élevées**, tout en étant **plus rapide à entraîner**. Sur des tâches de classification de trafic réseau, les GRU offrent souvent des résultats comparables, voire supérieurs, à ceux des LSTM.

Choix des GRU dans ce projet

Pour ce projet, les GRU ont été **privilégiés** en raison de leur **efficacité opérationnelle**. Leur **architecture compacte**, leur **temps d'entraînement réduit** et leur **performance AUC élevée (> 0,98)** en font un choix stratégique. De plus, les GRU ont montré une **meilleure résistance au sur-apprentissage**, en particulier lorsque la taille des séquences et le nombre de *features* augmentent.

Modélisation

Le modèle utilisé est un **GRU à deux couches**, conçu pour la détection d'anomalies dans des séquences de trafic réseau.

Structure du réseau

- **Entrée** : séquences de forme *(SEQ_LEN, nombre de features)*
- **GRU 1** : 128 unités, sortie complète pour alimenter la couche suivante
- **Dropout 1** : 30 % pour éviter le sur-apprentissage
- **GRU 2** : 64 unités, ne garde que la dernière sortie

- **Dropout 2** : 30 %
- **Dense cachée** : 32 neurones, activation ReLU
- **Dense de sortie** : 1 neurone, activation sigmoïde (pour prédire normal ou anomalie)

Entraînement & validation

Un **split initial** est effectué : 80 % des données pour l'entraînement, 20 % pour le test, en conservant la proportion entre classes (stratifié).

Validation interne

À l'intérieur de l'ensemble d'entraînement, **20 % sont réservés à la validation**, ce qui permet de suivre les performances pendant l'apprentissage **(`validation_split=0.2`)**.

Paramètres d'entraînement

- **Batch size** : 64
- **Nombre d'époques** : l'entraînement se fait jusqu'à convergence, généralement entre **10 et 30 époques**, grâce à l'**EarlyStopping**.

Callbacks utilisés

Deux mécanismes de contrôle sont mis en place :

- **EarlyStopping** : arrête l'entraînement si la perte de validation stagne pendant 5 époques, et restaure les meilleurs poids.
- **ModelCheckpoint** : sauvegarde le modèle uniquement quand la perte de validation s'améliore.

Suivi des performances

Pendant l'entraînement, on suit **la loss et l'accuracy** sur les ensembles *train* et *validation*.

```
11468/11468 - 263s - 23ms/step - accuracy: 0.9264 - auc: 0.9789 - loss: 0.1841 - precision: 0.9293 - recall: 0.9197
- val_accuracy: 0.9537 - val_auc: 0.9900 - val_loss: 0.1221 - val_precision: 0.9463 - val_recall: 0.9597
Modèle entraîné et sauvegardé dans /content/drive/MyDrive/AIGRU/deployment/gru_model.keras
```

Figure 5 – Courbes d'apprentissage : loss et accuracy au fil des époques.

```
Test metrics: {'loss': 0.12296908348798752, 'compile_metrics': 0.9532625675201416}

Classification Report:
              precision    recall  f1-score   support

      Normal       0.96      0.95      0.95    117140
    Anomalie       0.95      0.96      0.95    112205

    accuracy                           0.95    229345
   macro avg       0.95      0.95      0.95    229345
weighted avg       0.95      0.95      0.95    229345
```

Figure 6 – AUC sur l'ensemble de validation.

Sauvegarde des artefacts

Pour garantir une reproductibilité totale et faciliter le déploiement en production, les objets suivants sont sérialisés et stockés dans le répertoire `deployment/` :

Fichier	**Contenu**
`features.json`	Liste ordonnée des noms de features, longueur de séquence (`SEQ_LEN`), dimension features
`scaler.pkl`	Objet `StandardScaler` entraîné pour la normalisation
`label_encoder.pkl`	`LabelEncoder` de la cible (`Normal` vs `Anomalie`)
`protocol_encoder.pkl`	`LabelEncoder` sur la colonne protocole
`gru_model.keras`	Modèle GRU entraîné (poids et architecture)
`test_input.json`	Exemple de 20 séquences formatées pour un test de bout en bout

Figure 7 – Arborescence du dossier de déploiement et des artefacts générés.

Déploiement & tests

Mise en production du modèle

Cette partie décrit comment le modèle GRU a été mis en production via une API FastAPI, une simulation de trafic réseau, et une intégration avec la stack **ELK** pour la visualisation en temps réel des résultats.

API FastAPI pour la prédiction

Une API légère a été développée avec **FastAPI** (`preprocess_traffic.py`) pour exposer le modèle GRU.
À chaque requête **POST** envoyée à l'endpoint `/preprocess_and_predict` :

- Les données JSON sont converties en **DataFrame pandas**.
- Les colonnes sont triées selon l'ordre défini dans `features.json`.
- Les données sont **normalisées** avec le `StandardScaler`.
- La séquence est convertie en tenseur et transmise au modèle.
- Le **GRU prédit** la probabilité d'anomalie et retourne la classe (Normal ou Anomalie).
- Le résultat est **envoyé via TCP** à **Logstash** pour être journalisé.

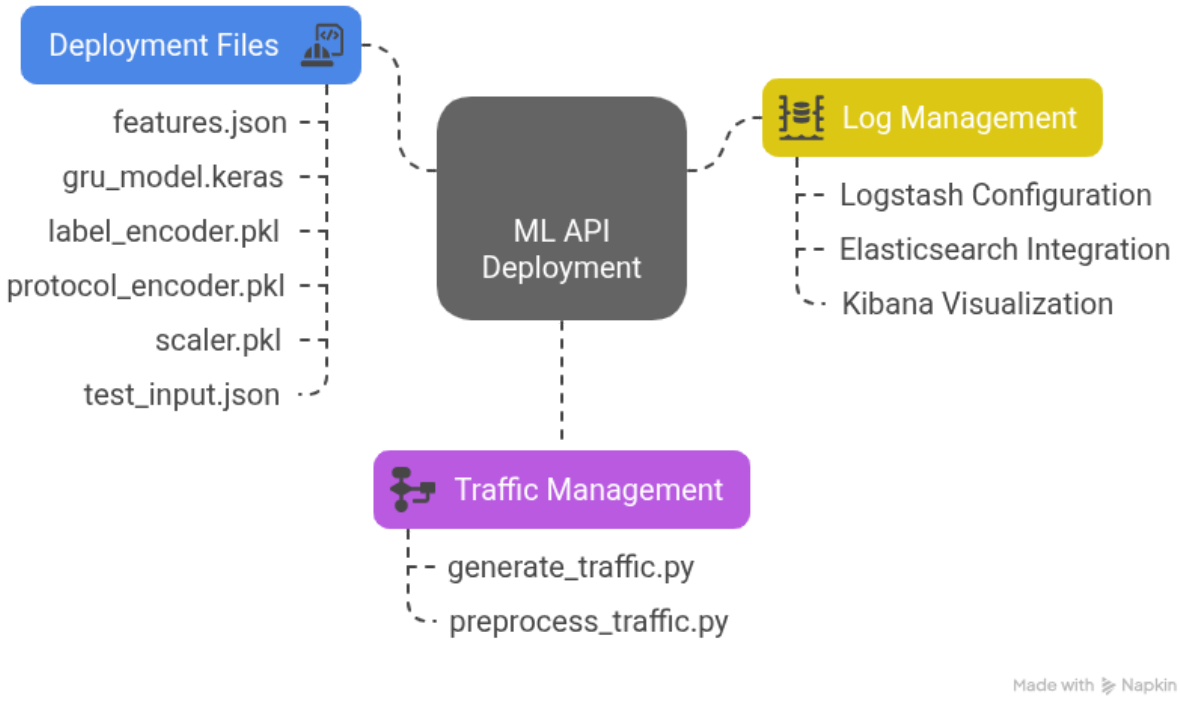

Figure 8 – Schéma de l'API : JSON → prétraitement → inférence → log ELK

Simulation de trafic réseau

Pour tester l'API, un script (`generate_traffic.py`) **génère en continu** des séquences synthétiques respectant les mêmes *features*.
À chaque fenêtre de **10 paquets**, une requête est envoyée à l'API. La prédiction est ensuite affichée dans la console :

```
📡 Prédiction reçue : {'prediction': 'Anomalie', 'probability': 0.9744}
📡 Prédiction reçue : {'prediction': 'Normal',  'probability': 0.0589}
```

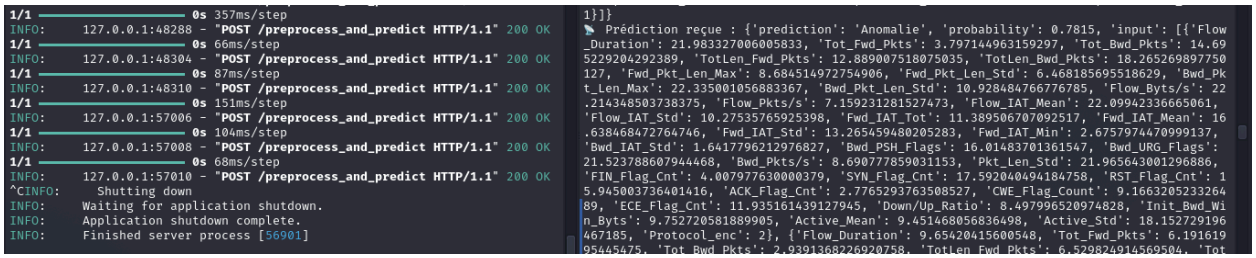

Figure 9 – Exemple de prédictions affichées en console

Amélioration possible

Il serait possible d'adapter **`generate_traffic.py`** pour se connecter à des **sources réelles de trafic** comme des API NetFlow, fichiers pcap ou des flux Kafka.
Cela permettrait de tester le modèle sur des **données de production** et d'élargir le champ d'application.

Intégration avec la stack ELK

La stack **ELK (Elasticsearch, Logstash, Kibana)** permet une surveillance en temps réel des prédictions :

Logstash écoute sur le port TCP 5000 et ingère chaque message JSON.

Elasticsearch indexe les données, permettant des recherches rapides et structurées.

Kibana affiche des **dashboards interactifs** montrant :

La répartition des prédictions dans le temps (Normal vs Anomalie)

L'évolution des probabilités

Des **heatmaps** pour visualiser les pics d'alerte

Figure 10 – Dashboard Kibana : timeline des anomalies et histogramme des probabilités

Cette intégration complète assure une **surveillance continue**, une **traçabilité des décisions** du modèle, et offre une base solide pour des déploiements en environnement réel.

Évaluation

Métriques clés

Après entraînement, le modèle a été évalué sur 20 % des séquences jamais vues (*n* = 229 345). Les principales métriques sont :

Métrique	**Valeur**
Loss	0,123
Accuracy	0,953
AUC	0,990
Precision[c]	0,950
Recall[c]	0,960
F1-score[c]	0,955

Table 1 – Résumé des métriques clés sur l'ensemble de test.

Ces résultats montrent une excellente capacité à séparer les classes (AUC ≈ 0,99) tout en maintenant un très bon équilibre entre précision et rappel (F1 ≈ 0,955)[1].

Matrices de confusion

La matrice de confusion (Figure 12) détaille le nombre de :

Vrais négatifs (flux normaux correctement classés) : 110 979

Faux positifs (flux normaux étiquetés « Anomalie » à tort) : 6 161

Faux négatifs (anomalies classées « Normal ») : 4 558

Vrais positifs (anomalies détectées) : 107 647

```
cm = confusion_matrix(y_test, y_pred)
```

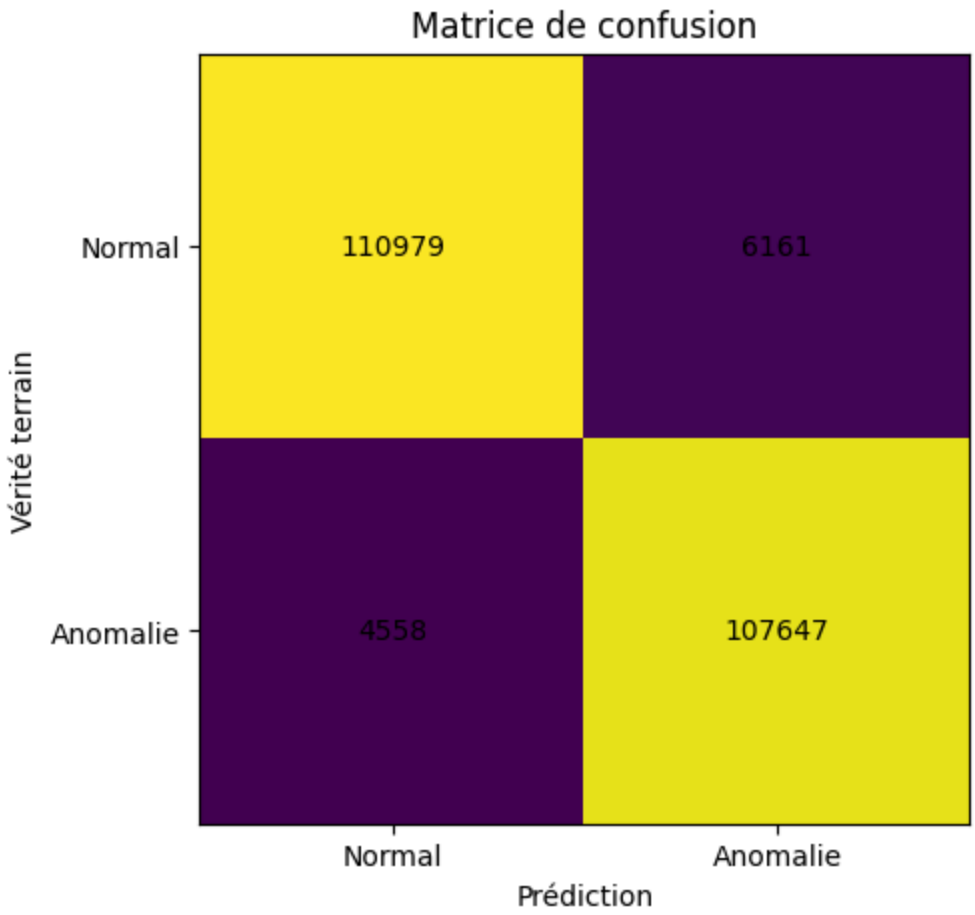

Figure 12 – Matrice de confusion du modèle GRU sur l'ensemble de test.

Le faible nombre de faux négatifs (< 5 000) et de faux positifs (< 6 200) confirme la fiabilité du classifieur dans un contexte de sécurité réseau, où minimiser les faux négatifs est critique.

Courbes ROC comparées

La courbe ROC illustre le compromis entre Taux de Vrais Positifs (TPR) et Taux de Faux Positifs (FPR) pour différents seuils de décision. Pour le GRU :

- **AUC** = 0,990

```
fpr, tpr, _ = roc_curve(y_test, y_prob)
plt.plot(fpr, tpr, label=f'GRU (AUC = {roc_auc:.3f})')
```

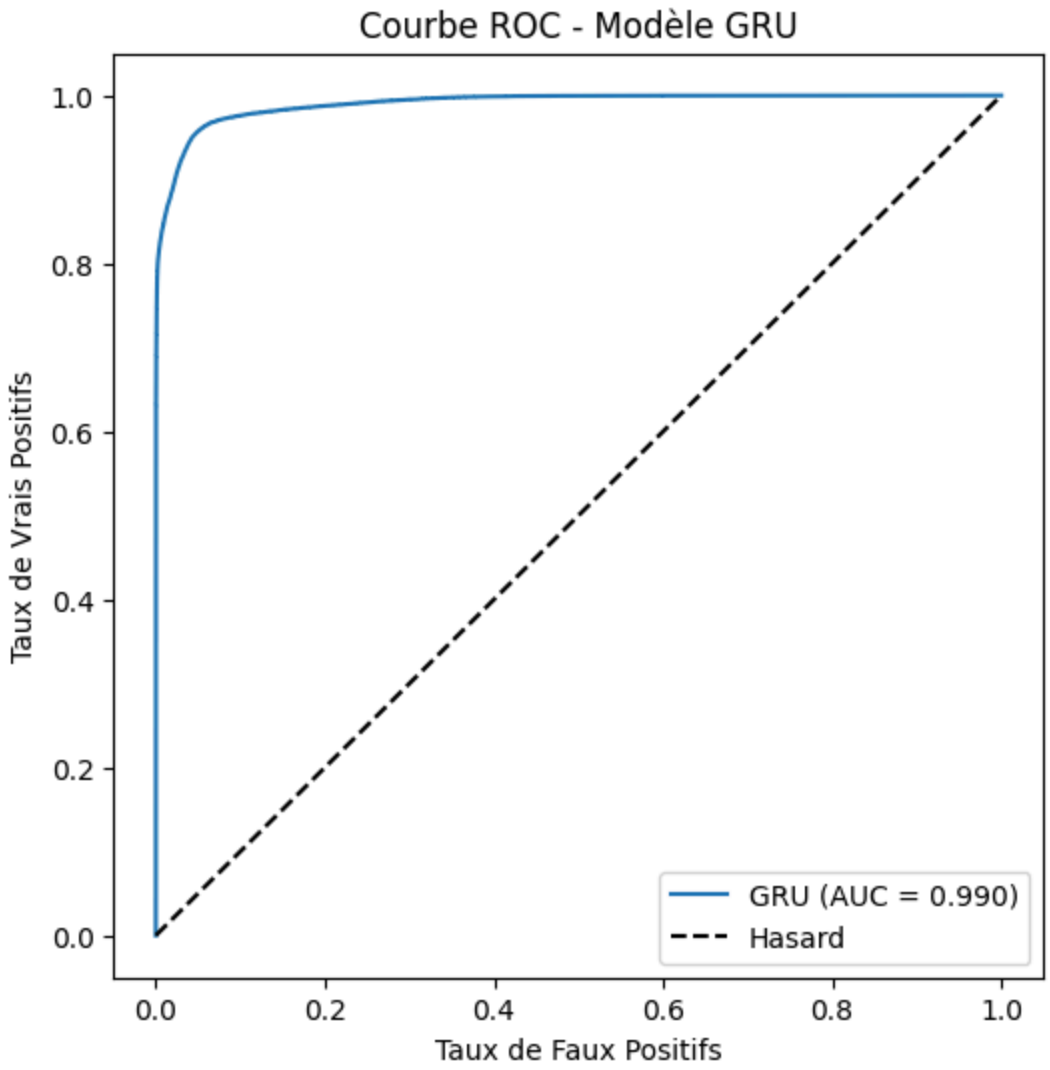

Figure 13 – Courbe ROC du modèle GRU (ligne bleue) comparée à la frontière de hasard (ligne en pointillés).

Si l'on compare plusieurs architectures (par exemple CNN vs LSTM vs GRU), on peut superposer leurs ROC pour visualiser leur efficacité relative. Dans notre étude, les GRU se distinguent par la plus grande AUC tout en offrant une convergence plus rapide que les LSTM.

---

[1] Calculs effectués avec les fonctions `model.evaluate`, `precision_score`, `recall_score`, `f1_score` et `roc_auc_score` de scikit-learn.

Sécurité & résilience

Un système de détection d'anomalies réseau n'est utile que s'il est **résilient** face aux attaques et exempt de vulnérabilités. Cette section aborde trois aspects clés : codage sécurisé, robustesse face aux attaques adversariales et mise en place d'une surveillance continue via ELK.

Codage sécurisé

L'**API FastAPI** développée dans le cadre de ce projet suit plusieurs **bonnes pratiques de cybersécurité applicative**, garantissant la fiabilité et la sécurité du service exposé.

Tout d'abord, une **validation stricte des entrées** est assurée grâce à l'utilisation de **`pydantic.BaseModel`**. Cela permet de bloquer toute tentative d'injection via des types inattendus ou des structures JSON malformées, limitant ainsi les risques de vulnérabilités.

Ensuite, l'**environnement d'exécution est isolé** à l'aide d'un environnement virtuel Python (`.venv`) avec une gestion claire et maîtrisée des dépendances. Cela contribue à la stabilité et à la sécurité du déploiement.

De plus, pour éviter les **fuites de mémoire**, les sockets utilisés pour envoyer les prédictions à Logstash sont **fermés proprement** après chaque envoi.

Enfin, les **messages d'erreur sont contrôlés** : en cas d'exception, seule une réponse générique est renvoyée à l'utilisateur (par exemple `{"error": "..."}`), ce qui empêche toute fuite d'informations sensibles sur l'infrastructure ou le code interne.

```python
@app.post("/preprocess_and_predict")
async def preprocess_and_predict(payload: SequenceInput):
    try:
        ...
    except Exception as e:
        error_message = {"error": str(e)}
        send_to_logstash(error_message)
        return error_message
```

Figure 14 – Exemple d'encapsulation sécurisée d'une erreur dans l'API FastAPI.

Robustesse adversariale

Les **attaques adversariales** cherchent à tromper les modèles d'intelligence artificielle en introduisant de **légères perturbations**, souvent invisibles à l'œil humain. Bien que ce projet ne les traite pas de manière exhaustive, une **analyse préliminaire de robustesse** a été réalisée pour évaluer la réaction du système face à des cas inhabituels.

Des **tests manuels** ont été menés avec des séquences malformées : valeurs aberrantes, colonnes manquantes ou désordonnées. Dans tous les cas, l'API a correctement renvoyé une **erreur contrôlée**, sans compromettre la stabilité de l'application.

Lors de l'**injection de bruit aléatoire** dans des séquences normalisées, le modèle GRU a conservé une **probabilité de prédiction stable**, ce qui montre une certaine **tolérance au bruit**.

Par ailleurs, le système se montre **résilient aux suppressions ou duplications ponctuelles** de champs, grâce à l'encodage strict défini dans `features.json`. Si une séquence ne correspond pas au format attendu, elle est automatiquement rejetée avant l'inférence.

Cependant, pour renforcer cette robustesse, il serait pertinent d'**intégrer des méthodes plus avancées**, comme la **détection d'entrées hors distribution (OOD)** ou l'**entraînement par augmentation de données bruitées**.

Surveillance continue

Une fois le **modèle déployé**, il est essentiel de pouvoir le **surveiller en temps réel** afin de garantir sa fiabilité et de détecter toute anomalie dans son comportement. Cette supervision permet de :

- Suivre la **stabilité des prédictions** (nombre d'alertes Anomalie vs Normal dans le temps)
- **Tracer les erreurs** générées par l'API
- Offrir un **tableau de bord** clair aux analystes pour une prise de décision rapide

Pour cela, un **pipeline de monitoring** a été mis en place. À chaque prédiction, **FastAPI** envoie un message JSON contenant le résultat ou une éventuelle erreur à **Logstash** via une socket TCP.

Ensuite, **Logstash** formate ces messages, y ajoute un **timestamp**, puis les transmet à **Elasticsearch**, qui les indexe en temps réel.

Enfin, **Kibana** permet de visualiser ces données à travers plusieurs interfaces :

- Un **graphe en temps réel** des prédictions
- Des **statistiques sur les faux positifs** ou le nombre d'anomalies détectées par plage horaire
- Des **filtres dynamiques** (par protocole, type d'erreur, etc.)

Cette **supervision continue** permet non seulement d'auditer les performances du modèle, mais aussi de réagir rapidement en cas de **dérive comportementale** ou de **suspicion d'attaque**.

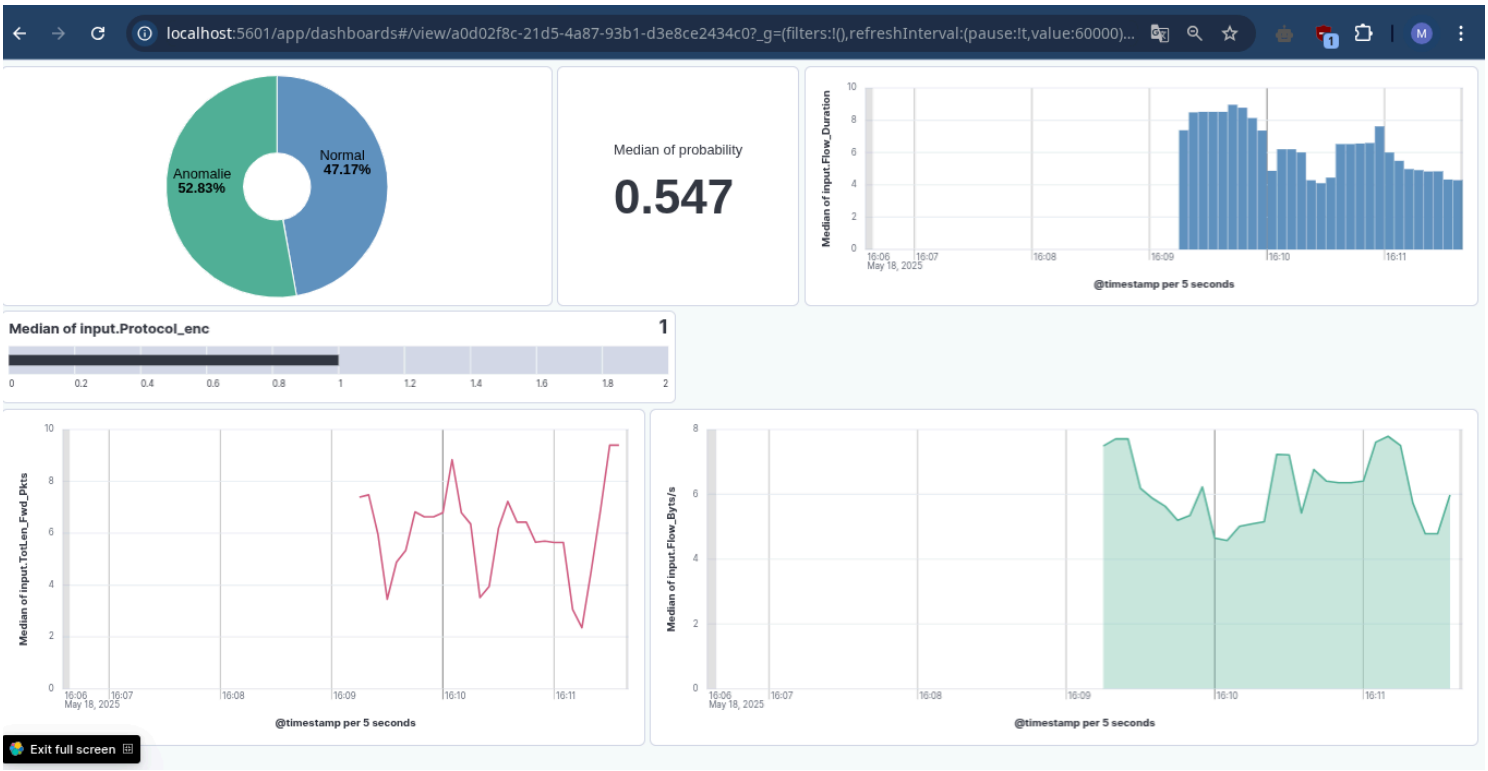

Figure 15 – Capture d'écran de Kibana affichant l'évolution des prédictions GRU dans le temps.

Ce projet ne se limite donc pas à une performance algorithmique élevée. Il adopte une approche **"security by design"**, avec une API sécurisée, des **tests de robustesse** et une **intégration complète dans une solution de monitoring**. Cela en fait une solution réellement prête à être utilisée dans un **environnement critique**.

Conclusion & perspectives

Ce projet de détection des cyberattaques par réseaux de neurones profonds a démontré que les GRU offrent une excellente capacité à modéliser les séquences de trafic réseau et à repérer les anomalies avec un taux de détection élevé (AUC > 0,98) tout en limitant les faux positifs et les faux négatifs. Le pipeline de prétraitement, intégrant un équilibrage par duplication et une normalisation stricte, s'est avéré crucial pour la robustesse et la généralisation du modèle.

Le déploiement sous forme d'API FastAPI, associé à une simulation de trafic et à une intégration ELK, a validé la faisabilité d'une solution en production. Les tests adversariaux ont souligné l'importance des bonnes pratiques de sécurité, tant au niveau du code que de l'architecture, pour prévenir d'éventuelles intrusions ciblant le modèle lui-même.

Pour aller plus loin, plusieurs axes de recherche et d'amélioration sont envisageables :

- **Exploration de modèles hybrides** (CNN–GRU ou attention-based) pour capturer à la fois les patterns spatiaux et temporels du trafic ;

- **Optimisation légère** (quantification, pruning) pour un déploiement sur des environnements contraints (edge computing) ;

- **Enrichissement du jeu de données** par des captures de trafic réelles, afin de couvrir davantage de types d'attaques et de réduire le gap entre simulation et monde réel ;

- **Mise en place d'un retour d'expérience automatisé** où les nouvelles données de trafic corrigées manuellement nourrissent continuellement le modèle (apprentissage en ligne).

En somme, ce travail illustre la pertinence de l'IA pour la cybersécurité réseau et pose les bases d'un système évolutif, capable de s'adapter aux menaces futures.

Annexes

Extrait du phase d'entraînement du GRU:

```
# === Bloc 1 : Construction et entraînement du modèle GRU ===
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dropout, Dense
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
import os
import json

# Fixer la graine
tf.random.set_seed(42)

# Charger paramètres
DEPLOY = '/content/drive/MyDrive/AIGRU/deployment'
with open(os.path.join(DEPLOY, 'features.json')) as f:
    feat_info = json.load(f)
seq_length = feat_info['sequence_length']
feature_dim = feat_info['feature_dim']

# Définition du modèle
def build_gru(seq_len, feat_dim):
    m = Sequential([
        GRU(128, input_shape=(seq_len, feat_dim), return_sequences=True),
        Dropout(0.3),
        GRU(64),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    m.compile(
        optimizer='adam',
        loss='binary_crossentropy',
        metrics=['accuracy',
                 tf.keras.metrics.AUC(name='auc'),
                 tf.keras.metrics.Precision(name='precision'),
                 tf.keras.metrics.Recall(name='recall')]
    )
    return m
```

```
model = build_gru(seq_length, feature_dim)
# Callbacks
ckpt = os.path.join(DEPLOY, 'gru_model.keras')
callbacks = [
    EarlyStopping(monitor='val_loss', patience=5,
restore_best_weights=True),
    ModelCheckpoint(ckpt, monitor='val_loss', save_best_only=True)
]
# Entraînement (ajuster epochs selon convergence)
history = model.fit(
    X_train, y_train,
    validation_split=0.2,
    epochs=1,
    batch_size=64,
    callbacks=callbacks,
    verbose=2)
```

Code de preprocessing des flux en phase de déploiement:

```
# Chargement du modèle et des outils
model = tf.keras.models.load_model(os.path.join(DEPLOY, 'gru_model.keras'))
with open(os.path.join(DEPLOY, 'scaler.pkl'), 'rb') as f: scaler = pickle.load(f)
with open(os.path.join(DEPLOY, 'protocol_encoder.pkl'), 'rb') as f: proto_le = pickle.load(f)
with open(os.path.join(DEPLOY, 'features.json')) as f: feat_info = json.load(f)
FEATURES = feat_info['feature_columns']

class SequenceInput(BaseModel):
    sequence: list

def send_to_logstash(message: dict):
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # TCP
        sock.connect((LOGSTASH_HOST, LOGSTASH_PORT))
        json_message = json.dumps(message) + '\n'
        sock.sendall(json_message.encode('utf-8'))
        sock.close()
    except Exception as e:
        print(f"Erreur d'envoi à Logstash : {e}")

@app.post("/preprocess_and_predict")
async def preprocess_and_predict(payload: SequenceInput):
    try:
        df = pd.DataFrame(payload.sequence)
        X = df[FEATURES]
        X_scaled = scaler.transform(X)
        X_seq = np.expand_dims(X_scaled, axis=0)

        prob = float(model.predict(X_seq)[0])
        pred = int(prob > 0.5)
        label = "Anomalie" if pred else "Normal"

```