Questionnaire complet – IDS AI System

Projet Tutored

July 23, 2025

Contents

1 Introduction

Ce document répond à toutes les questions essentielles sur le projet **IDS AI System**, qui combine une intelligence artificielle (Deep Learning) pour la détection d'intrusions réseau et une interface web moderne (Next.js). Chaque question est suivie d'une explication claire, d'exemples de code, et d'illustrations pour permettre à tout étudiant de comprendre le fonctionnement du projet.

2 Fondamentaux : Machine Learning et Deep Learning

2.1 Qu'est-ce que le Machine Learning ?

Le Machine Learning (apprentissage automatique) regroupe les techniques où l'ordinateur apprend à partir de données, sans être explicitement programmé pour chaque tâche. Il s'agit par exemple de reconnaître des motifs ou de faire des prédictions à partir d'exemples.

2.2 Qu'est-ce que le Deep Learning ?

Le Deep Learning (apprentissage profond) est une branche du Machine Learning qui utilise des réseaux de neurones comportant plusieurs couches cachées. Cela permet au modèle de comprendre des relations très complexes dans les données, comme reconnaître des attaques réseau à partir de nombreux paramètres.

2.3 Différences entre Machine Learning et Deep Learning

La principale différence réside dans la profondeur et la capacité du modèle :

- Le Machine Learning classique utilise souvent des modèles plus simples (arbres de décision, SVM, régressions, etc.) et nécessite beaucoup de travail manuel pour choisir les bonnes variables (features).
- Le Deep Learning automatise l'extraction des caractéristiques grâce à ses couches successives, et peut traiter des données brutes ou très complexes.

Dans ce projet, on utilise un modèle Deep Learning de type CNN-LSTM, capable d'apprendre automatiquement des motifs dans le trafic réseau.

3 Architecture du modèle IA

3.1 Présentation générale

Le modèle utilisé dans ce projet est un réseau de neurones profond qui combine deux types d'architectures :

- **CNN (Convolutional Neural Network)** : pour repérer des motifs locaux dans les séquences de données réseau.
- **LSTM (Long Short-Term Memory)** : pour mémoriser des séquences et détecter des comportements anormaux sur la durée.

Les données passent d'abord par des couches de convolution 1D, puis par des couches LSTM, et enfin par des couches entièrement connectées (denses) avant d'arriver à la sortie.

3.2 Nombre de couches et de neurones

Le modèle comporte :

- 2 couches de convolution (64 et 128 filtres)
- 2 couches LSTM (128 et 64 neurones)
- 2 couches denses (128 et 64 neurones)
- 1 couche de sortie (nombre de neurones = nombre de classes, ici 9)

Calcul du nombre total de neurones (hors batchnorm, dropout, etc.) :

- Conv1D-1 : 64 filtres
- Conv1D-2 : 128 filtres
- LSTM-1 : 128 neurones
- LSTM-2 : 64 neurones
- Dense-1 : 128 neurones
- Dense-2 : 64 neurones
- Sortie : 9 neurones

Total (hors redondance) : 64 + 128 + 128 + 64 + 128 + 64 + 9 = **585 neurones**

3.3 Caractéristiques du réseau de neurones

- **Fonctions d'activation :** ReLU pour les couches cachées, Softmax pour la sortie.
- **Normalisation :** BatchNormalization après chaque convolution et dense.
- **Régularisation :** Dropout (0.25 ou 0.5) pour éviter le surapprentissage.
- **Architecture séquentielle :** Les couches sont empilées les unes après les autres.

3.4 Extrait de code du modèle

```
model = models.Sequential([
    layers.Input(shape=self.input_shape),
    layers.Conv1D(64, 3, activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.Conv1D(128, 3, activation='relu', padding='same'),
    layers.BatchNormalization(),
    layers.MaxPooling1D(2),
    layers.Dropout(0.25),
```

```
        layers.LSTM(128, return_sequences=True),
        layers.Dropout(0.25),
        layers.LSTM(64),
        layers.Dropout(0.25),
        layers.Dense(128, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(64, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.5),
        layers.Dense(self.num_classes, activation='softmax')
])
```

3.5 Résumé sous forme de tableau

Type de couche	Nombre de neurones/filtres	Activation
Conv1D	64	ReLU
Conv1D	128	ReLU
LSTM	128	-
LSTM	64	-
Dense	128	ReLU
Dense	64	ReLU
Dense (sortie)	9	Softmax

4 Données utilisées

4.1 Données d'entrée

Les données d'entrée sont issues du jeu de données NSL-KDD, un standard pour la détection d'intrusions réseau. Chaque exemple correspond à une connexion réseau décrite par de nombreux paramètres (adresses IP, ports, protocole, etc.).

4.2 Prétraitement des données

Avant d'être utilisées par le modèle, les données subissent plusieurs transformations :

- Les variables textuelles (catégorielles) sont converties en variables numériques (one-hot encoding).
- Les valeurs sont normalisées (StandardScaler) pour que toutes les features soient sur la même échelle.
- Les labels (types d'attaque) sont encodés en one-hot.

```
categorical_columns = X.select_dtypes(include=['object']).columns
X = pd.get_dummies(X, columns=categorical_columns)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
label_encoder = LabelEncoder()
```

```
y_encoded = label_encoder.fit_transform(y)
y_categorical = to_categorical(y_encoded)
```

4.3 Données de sortie

Le modèle prédit la classe d'attaque parmi 9 catégories :

```
ATTACK_TYPES = {
    0: 'Normal',
    1: 'DoS',
    2: 'Probe',
    3: 'R2L',
    4: 'U2R',
    5: 'SQL Injection',
    6: 'XSS',
    7: 'Port Scan',
    8: 'Brute Force'
}
```

4.4 Provenance des données de test

Dans ce projet, le fichier utilisé est `NSL-KDD-Train.csv`. Il n'y a pas de jeu de test séparé fourni dans le dossier, mais une partie des données est réservée à la validation (20% par défaut).

5 Entraînement et hyperparamètres

5.1 Hyperparamètres utilisés

- Optimiseur : Adam
- Fonction de perte : categorical extunderscore crossentropy
- Epochs : 50
- Batch size : 32
- EarlyStopping (patience=5)
- ReduceLROnPlateau (patience=3, factor=0.2)

```
history = self.model.fit(
    X_train, y_train,
    validation_data=(X_val, y_val),
    epochs=epochs,
    batch_size=batch_size,
    callbacks=[
        tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, re
        tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss', factor=0.2
```

```
    ],
    verbose=1
)
```

6 Évaluation du modèle et surapprentissage

6.1 Pourquoi évaluer un modèle ?

Après l'entraînement, il est essentiel de vérifier si le modèle a bien appris à reconnaître les attaques, sans se contenter de mémoriser les exemples. Pour cela, on utilise des métriques et des courbes d'apprentissage.

6.2 Métriques d'évaluation : définitions et exemples

Le modèle est évalué à l'aide de plusieurs métriques, chacune ayant un sens précis :

- **Accuracy (précision globale)** : proportion de bonnes réponses sur l'ensemble des prédictions. Si le modèle prédit correctement 90 exemples sur 100, l'accuracy est de 90%.
- **Precision** : parmi toutes les fois où le modèle a prédit une attaque, combien étaient vraiment des attaques ? Cela mesure la fiabilité des alertes.
- **Recall (rappel)** : parmi toutes les vraies attaques, combien ont été détectées par le modèle ? Cela mesure la capacité à ne rien rater.
- **F1-score** : moyenne harmonique entre la précision et le rappel. C'est un bon indicateur global quand il y a un déséquilibre entre classes.

Exemple simple : Si sur 100 connexions, il y a 10 attaques et 90 normales, et que le modèle détecte 8 attaques (dont 7 vraies et 1 fausse alerte) :

- Accuracy = (90 bonnes + 7 bonnes) / 100 = 97%
- Precision = 7 / 8 = 87.5%
- Recall = 7 / 10 = 70%
- F1-score = $2 \times (0.875 \times 0.7)/(0.875 + 0.7) \approx 0.778$

6.3 Comment lire les courbes d'apprentissage ?

Après chaque entraînement, le projet sauvegarde des images de courbes (`training_history.png`) dans les dossiers `data/models/ids_model_.../`. Ces courbes montrent l'évolution de la loss (erreur) et de l'accuracy (précision) sur l'ensemble d'entraînement et de validation, au fil des époques (epochs).

- **Courbe de loss** : Plus la courbe descend, mieux le modèle apprend. Si la loss de validation remonte alors que celle d'entraînement continue de baisser, c'est un signe de surapprentissage (le modèle mémorise trop les exemples d'entraînement et généralise mal).

- **Courbe d'accuracy** : Plus la courbe monte, plus le modèle est précis. On cherche à avoir une accuracy élevée sur la validation, pas seulement sur l'entraînement.

Comment interpréter ces courbes ?

- Si les courbes d'entraînement et de validation sont proches et évoluent dans le même sens, le modèle apprend bien.
- Si la courbe d'entraînement continue de s'améliorer mais que celle de validation stagne ou se dégrade, il y a surapprentissage.
- Si la loss de validation est très supérieure à celle d'entraînement, le modèle ne généralise pas bien.

6.4 Comparer plusieurs modèles

Dans le dossier `data/models/`, chaque sous-dossier (ex : `ids_model_20250604_081352`) contient une image `training_history.png` qui permet de visualiser la qualité de l'entraînement pour ce modèle précis. Pour comparer plusieurs modèles, il suffit de regarder :

- Quelle courbe de validation a la loss la plus basse et l'accuracy la plus haute ?
- Le modèle qui a la meilleure courbe de validation (et pas seulement d'entraînement) est généralement le meilleur pour de nouvelles données.

Conseil pratique : Choisissez le modèle dont la courbe de validation est la plus stable et la plus performante, même si la courbe d'entraînement est un peu moins bonne. Cela garantit une meilleure généralisation.

Exemple réel du projet :

- Ouvrez les images `training_history.png` dans `data/models/ids_model_20250604_081352/` et `data/models/ids_model_20250717_140013/`.
- Comparez la courbe de loss et d'accuracy sur la validation : celle qui reste la plus basse (pour la loss) et la plus haute (pour l'accuracy) sur la fin de l'entraînement indique le meilleur modèle.

Ainsi, même sans être expert, il est possible de choisir le modèle le plus fiable simplement en regardant ces courbes et en comprenant ce qu'elles signifient.

6.5 Exemples de courbes d'apprentissage réelles du projet

Pour mieux comprendre comment lire et comparer les courbes d'apprentissage, voici deux exemples issus de ce projet. Chaque image montre l'évolution de la loss et de l'accuracy sur l'entraînement et la validation.

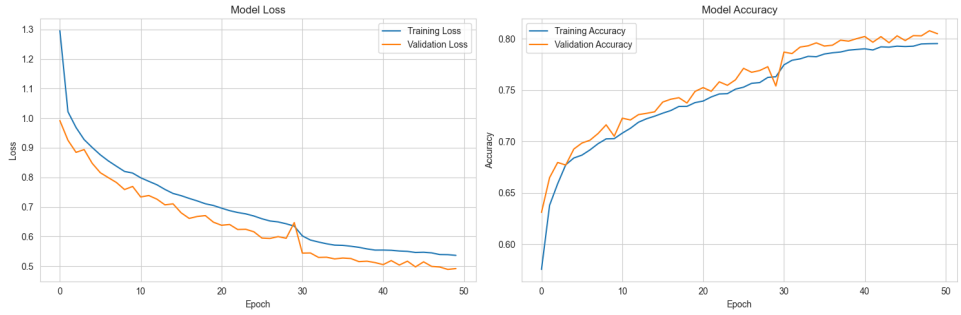

Figure 1: Courbes d'apprentissage du modèle entraîné le 2025-06-04. On observe la loss et l'accuracy sur l'entraînement et la validation.

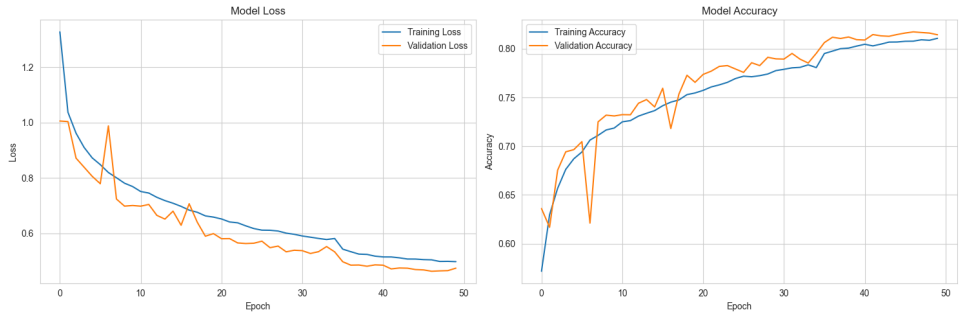

Figure 2: Courbes d'apprentissage du modèle entraîné le 2025-07-17. Comparer la stabilité et la performance des courbes de validation pour choisir le meilleur modèle.

7 Questions complémentaires sur le backend IA

7.1 Comment le modèle est-il sauvegardé et rechargé ?

Le modèle entraîné est sauvegardé dans un dossier dédié (par exemple `data/models/ids_model_2025060` sous forme de fichiers TensorFlow/Keras. Le code utilise la méthode `model.save()` pour la sauvegarde et `tf.keras.models.load_model()` pour le rechargement. Les objets de normalisation (scaler) et d'encodage des labels sont également sauvegardés pour garantir la cohérence lors des prédictions futures.

```
self.model.save(model_path)
# ...
model = tf.keras.models.load_model(model_path)
```

7.2 Comment faire une prédiction sur une nouvelle donnée ?

Pour prédire si une connexion réseau est une attaque, il faut :

1. Prétraiter la donnée (extraction et normalisation des features)
2. Appliquer le modèle pour obtenir la prédiction
3. Interpréter la sortie (classe prédite)

Cela se fait via la méthode `predict()` du modèle, après avoir transformé la donnée d'entrée au bon format.

```
predictions = model.predict(X)
predicted_class = np.argmax(predictions[0])
```

7.3 Comment sont gérés les logs de tests ?

Après chaque prédiction ou test, un log est enregistré dans un fichier JSON (`data/test_logs.json`). Chaque log contient l'entrée, la prédiction, la classe réelle (si connue), la confiance, et le statut (succès/échec). Cela permet de suivre l'historique des performances du modèle et de diagnostiquer d'éventuelles erreurs.

```
log_entry = {
    'timestamp': datetime.now().isoformat(),
    'testType': 'Validation',
    'input': str(X_val[i].flatten().tolist()),
    'prediction': int(y_pred_labels[i]),
    'confidence': float(y_pred[i].max()),
    'true_class': int(y_val_labels[i]),
    'status': 'success' if y_pred_labels[i] == y_val_labels[i] else 'fail'
}
```

7.4 Quelles sont les limites du modèle actuel ?

Le modèle dépend fortement de la qualité et de la diversité des données d'entraînement. Il peut avoir du mal à détecter des attaques très différentes de celles vues pendant l'entraînement (généralisation limitée). De plus, il n'y a pas de gestion automatique de la mise à jour du modèle en production, ni de détection d'attaques inconnues (zero-day).

7.5 Comment ajouter un nouveau type d'attaque ?

Pour ajouter un nouveau type d'attaque, il faut :

1. Ajouter la nouvelle classe dans la variable `ATTACK_TYPES` du code.
2. S'assurer que les données d'entraînement contiennent des exemples de cette attaque.
3. Réentraîner le modèle pour qu'il apprenne à la reconnaître.

7.6 Comment le backend gère-t-il la scalabilité ou la mise à jour du modèle ?

Actuellement, le backend charge le modèle en mémoire et l'utilise pour toutes les prédictions. Pour mettre à jour le modèle, il faut réentraîner et sauvegarder un nouveau modèle, puis redémarrer le service pour qu'il charge la nouvelle version. Il n'y a pas encore de mécanisme automatique de mise à jour ou de gestion de plusieurs versions en parallèle.

7.7 Comment sont gérées les erreurs ou les cas inattendus ?

Le code prévoit des blocs `try/except` pour capturer les erreurs lors des prédictions ou du chargement du modèle. En cas d'erreur, un message explicite est retourné et, dans certains cas, une prédiction simulée est renvoyée pour éviter de bloquer le système.

```
try:
    # ... prediction
except Exception as e:
    print(f"Erreur lors de la prediction : {str(e)}")
    # ... prediction simulee
```

8 Frontend : Interface utilisateur Next.js

8.1 Présentation générale

Le frontend du projet est développé avec Next.js, un framework moderne basé sur React. Il permet de créer une interface utilisateur réactive, ergonomique et facilement maintenable. Le code source se trouve dans le dossier `frontend-next/`.

8.2 Organisation et architecture

Le projet est organisé en pages (`src/app/`) et en composants réutilisables (`src/components/`). Chaque page correspond à une vue principale de l'application (dashboard, alertes, règles, paramètres, etc.).

8.3 Rôle de Next.js et des fichiers principaux

Next.js gère le routage, le rendu côté client, et l'organisation des pages. Les fichiers importants sont :

- `src/app/layout.tsx` : structure globale de l'application (sidebar, layout)
- `src/app/page.tsx` : page d'accueil (redirection)
- `src/app/dashboard/page.tsx` : tableau de bord principal
- `src/app/alerts/page.tsx` : affichage des alertes
- `src/app/settings/page.tsx` : configuration des paramètres
- `src/components/` : composants réutilisables (tableaux, graphiques, formulaires, etc.)

8.4 Communication avec le backend

Le frontend communique avec le backend Flask via des appels API HTTP (principalement avec `fetch`). Les endpoints utilisés sont par exemple :

- `/api/stats/traffic` : données réseau en temps réel
- `/api/stats/alerts` : liste des alertes
- `/api/settings` : lecture et modification des paramètres
- `/api/rules` : gestion des règles IDS

Exemple d'appel API dans un composant React :

```
useEffect(() => {
    fetch('http://localhost:5000/api/stats/alerts')
        .then(res => res.json())
        .then(setAlerts);
}, []);
```

8.5 Gestion de l'état et des effets

L'état de l'application est géré avec les hooks React `useState` et `useEffect`. Par exemple, pour stocker et mettre à jour la liste des alertes :

```
const [alerts, setAlerts] = useState<Alert[]>([]);
useEffect(() => {
    fetch('http://localhost:5000/api/stats/alerts')
        .then(res => res.json())
        .then(setAlerts);
}, []);
```

Des hooks personnalisés (ex : `useNetworkData`) sont utilisés pour centraliser la logique de récupération des données réseau.

8.6 Composants principaux

- **Dashboard** : vue d'ensemble des métriques, historique, logs de tests
- **AlertsTable** : affichage des alertes récentes
- **SettingsPage** : gestion des paramètres (seuils, modules actifs)
- **RulesPage** : gestion dynamique des règles IDS
- **ModelStats, NetworkTraffic, SystemStats** : visualisation des statistiques et graphiques

8.7 Gestion des erreurs et du chargement

Chaque appel API gère les états de chargement et d'erreur pour informer l'utilisateur :

```
const [loading, setLoading] = useState(true);
const [error, setError] = useState<string | null>(null);

useEffect(() => {
    fetch('http://localhost:5000/api/stats/alerts')
        .then(res => res.json())
        .then(data => {
            setAlerts(data);
            setLoading(false);
        })
        .catch(() => {
            setError('Erreur lors du chargement des alertes.');
```

```
            setLoading(false);
        });
}, []);
```

8.8 Bibliothèques externes utilisées

Le frontend utilise plusieurs bibliothèques pour améliorer l'expérience utilisateur :

- `react` et `next` : base du framework
- `tailwindcss` : styles et design moderne
- `recharts` : graphiques interactifs
- `react-hook-form` : gestion des formulaires
- `sonner` : notifications
- `lucide-react` : icônes

8.9 Résumé visuel de l'architecture frontend

- Sidebar (navigation)
- Pages principales (Dashboard, Alertes, Règles, Paramètres)
- Composants réutilisables (tableaux, graphiques, formulaires)

L'architecture modulaire permet de faire évoluer facilement l'interface et d'ajouter de nouvelles fonctionnalités.