TP Noté

Supervised Learning & Reinforcement Learning

AI53

PERRIN Raphaël

25 novembre 2024

Table des matières

1 Introduction

Dans le cadre de ce projet, deux problématiques distinctes ont été abordées :

- **Processus de Décision Markovien (MDP)** (Reinforcement Learning): L'objectif était de modéliser et résoudre un environnement simple constitué d'une série de flèches, chacune pouvant pointer dans deux directions distinctes. Cette modélisation a nécessité l'initialisation complète des états possibles, des actions disponibles et des fonctions de transition associées. La résolution a été réalisée à l'aide de l'algorithme Q-Iteration, permettant d'optimiser les décisions dans cet environnement.
- **Régression linéaire, réseaux de neurones, et forêt aléatoire** (Supervised Learning): Une étude comparative a été menée entre une approche classique de régression linéaire et des réseaux de neurones profonds. Cette analyse vise à évaluer les performances des deux modèles dans des scénarios de prédiction.

2 Reinforcement Learning

L'apprentissage par renforcement (Reinforcement Learning) est une méthode d'apprentissage automatique où un agent apprend à prendre des décisions en interagissant avec un environnement. L'agent effectue des actions et reçoit des récompenses ou des pénalités en fonction des résultats. Son objectif est de maximiser les récompenses cumulées sur le long terme.

2.1 Processus de Décision Markovien (MDP)

Pour appliquer concrètement cette méthode d'apprentissage automatique, nous avons développé un programme Python basé sur l'algorithme Q-Iteration. Cet algorithme permet de déterminer les meilleures décisions pour passer d'un état initial à un état final.

L'exercice proposé consistait en une série de flèches, chacune pouvant être orientée dans deux directions distinctes (haut ou bas). L'objectif est d'atteindre un état final où toutes les flèches sont orientées dans la direction opposée à leur configuration initiale.

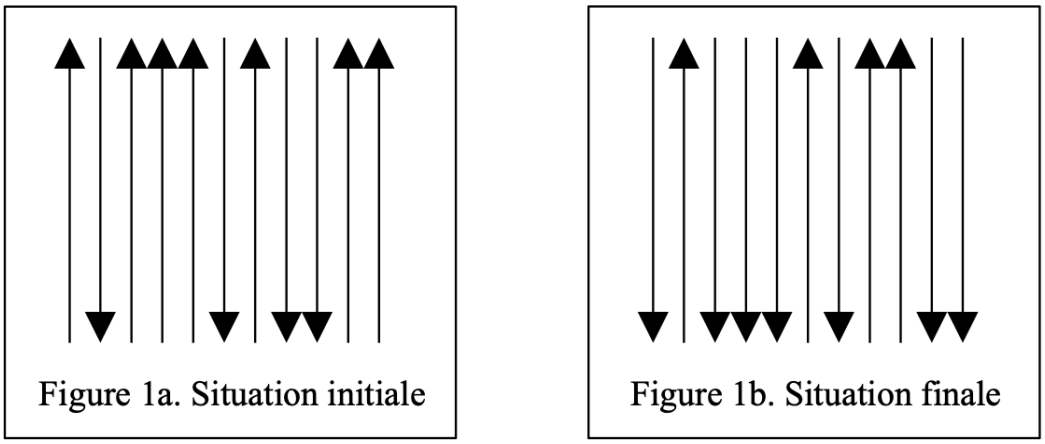

Figure 1: Etat initial et final des flèches

Pour ce faire, il a fallu définir notre MDP :

- **Etat** : Un état peut être représenté sous la forme d'une liste binaire de n éléments. En effet, une flèche pointant vers le haut peut être représentée par la valeur 1, tandis qu'une flèche pointant vers le bas peut être représentée par la valeur 0.
- **Action** : Deux actions distinctes sont disponibles dans notre énoncé, c'est à dire :
  1. Inverser l'orientation de trois flèches adjacentes ayant la même orientation.
  2. Inverser l'orientation de deux flèches adjacentes ayant des orientations opposées.
- **Transition** : La fonction de transition détermine le nouvel état en appliquant l'action sélectionnée, qu'il s'agisse de l'inversion de trois flèches adjacentes de même orientation ou de deux flèches adjacentes ayant des orientations opposées.

2.2 Création des états

Dans notre cas, nous souhaitons généraliser notre programme pour une séries de flèche de taille n. Ainsi, n^2 combinaisons d'états sont possibles dans notre ensemble, étant donné que les valeurs possibles sont 0 ou 1.

De manière générale, une Q-table utilisée dans l'algorithme QIteration est représentée sous la forme d'un tableau, où chaque ligne correspond à un état et chaque colonne à une action.

	Left	Right	Up	Down
(1,1)	0	0	0	0
(1,2)	0	0	0	0
(1,3)	0	0	0	0
(2,1)	0	0	0	0
(2,2)	0	0	0	0
(2,3)	0	0	0	0
(3,1)	0	0	0	0
(3,2)	0	0	0	0
(3,3)	0	0	0	0

Figure 2: Q-table

Pour des raisons de performance, nous choisirons de représenter la Q-table sous forme d'un dictionnaire. Dans cette structure, chaque clé représente un état unique, et chaque valeur associée est une liste contenant les actions possibles à partir de cet état. Cette représentation permet un accès rapide et efficace aux actions réalisables pour chaque état sans nécessiter une table à deux dimensions.

$$Q_{table} = \left\{ \begin{array}{l} s_1 : [a_1, a_2, a_3] \\ s_2 : [a_2, a_4] \\ s_3 : [a_1, a_4] \\ s_4 : [a_1, a_2] \end{array} \right\}$$

Dans notre programme, nous commençons simplement par remplir notre dictionnaire de Q-table par des 0 pour chaque action réalisable sur chaque état :

```
for state in range(2 ** n):
        state_list = [int(x) for x in bin(state)[2:].zfill(n)]
        actions = env.available_actions(state_list)
        q_table[tuple(state_list)] = {action: 0 for action in actions}
```

2.3 Détermination des actions possibles

Les actions réalisables ont été détaillées précédemment et peuvent être implémentées facilement par l'intermédiaire de cette fonction :

```
def available_actions(self, state):
        actions = []

        for i in range(self.n - 2):
            if len(set(state[i:i+3])) == 1:
                actions.append((1, i))

        for i in range(self.n - 1):
            if state[i] != state[i+1]:
                actions.append((2, i))

        return actions
```

La première boucle for se charge d'ajouter les actions correspondant à la règle 1 (modifier l'orientation de trois flèches adjacentes de même orientation).

La ligne if len(set(state[i:i+3])) == 1: permet de vérifier si trois flèches adjacentes ont la même direction. Les sets sont des collections qui stockent uniquement des valeurs uniques. Par conséquent, si les trois flèches possèdent la même direction, seule cette direction unique sera présente dans le set, et la longueur du set sera égale à 1. En revanche, si les flèches ont des directions différentes, le set contiendra à la fois la valeur correspondant à chaque direction (0 et 1), et sa longueur sera donc égale à 2.

La deuxième boucle for se charge de la deuxième règle. Elle vérifie simplement si l'orientation d'une flèche est inverse à celle de sa voisine.

2.4 Fonction de transition

La fonction de transition prend en entrée un état et une action, puis calcule le nouvel état en fonction de l'action réalisée.

Voici le code correspondant :

```
def transition(self, state, action):
        new_state = state.copy()

        if action[0] == 1:
            i = action[1]
            new_state[i:i+3] = [1 - x for x in new_state[i:i+3]]

        elif action[0] == 2:
            i = action[1]

            new_state[i] = 1 - state[i]
            new_state[i+1] = 1 - state[i+1]

        return new_state
```

La ligne new_state[i:i+3] = [1 - x for x in new_state[i:i+3]] inverse l'orientation des trois flèches adjacentes dans l'état. En effet, cette opération fonctionne de la manière suivante :

- Si l'orientation de chaque flèche est vers le haut (1), alors l'expression 1 - 1 donne 0, ce qui signifie que les flèches seront maintenant orientées vers le bas.
- À l'inverse, si les flèches étaient orientées vers le bas (0), alors 1 - 0 donne 1, et les flèches seront réorientées vers le haut.

2.5 Détermination de l'état final

L'état final est atteint lorsque l'orientation de toutes les flèches est inversée par rapport à leur état d'origine.

Ceci est vérifiable simplement par l'intermédiaire de cette fonction :

```
def is_goal_state(self, state, initial_state):
    return state == [1 - arrow for arrow in initial_state]
```

2.6 Implémentation du QIteration

L'objectif de la boucle while ci-dessous est d'itérer sur la table Q et de mettre à jour ses valeurs jusqu'à ce qu'elles convergent. Autrement dit, on veut s'assurer que les modifications apportées à la table Q n'entraînent plus de changements significatifs après un certain nombre d'itérations, ce qui indique que l'algorithme a trouvé une politique stable.

Voici le code en question :

```
q_table_update = {}

while True:
    delta = 0
    for state in q_table:
        actions = env.available_actions(list(state))
        for action in actions:
            old_value = q_table[state][action]
            next_state = env.transition(list(state), action)
            max_q_next = max(q_table[tuple(next_state)].values(), default=0)
            reward = 100 if env.is_goal_state(next_state, initial_state)
                else 0
            q_table_update[(state, action)] = reward + gamma * max_q_next

            delta = max(delta, abs(old_value - q_table_update[(state,
                action)]))

    for (state, action), new_value in q_table_update.items():
        q_table[state][action] = new_value

    if delta < theta:
        break
```

Nous pouvons également voir qu' un deuxième dictionnaire appelé q_table_update a été créé, qui sert à stocker temporairement les nouvelles valeurs calculées pour chaque paire (état, action) pendant l'itération. À la fin de l'itération, ce dictionnaire est utilisé pour mettre à jour les valeurs de q_table.

Cela permet de garantir que l'algorithme utilise les valeurs initiales de q_table pour calculer les nouvelles valeurs, et ne prend pas en compte les modifications apportées lors de la même itération. En

d'autres termes, lors du calcul de max_q_next (la valeur maximale des actions possibles dans l'état suivant), nous utilisons l'ancienne version de la table Q (avant toute modification de cette itération). Cela évite que les valeurs modifiées au cours de la même itération influencent les calculs des autres états et actions.

La politique de changement de chaque valeur de la Q-table suit l'algorithme que nous avons pu voir en cours :

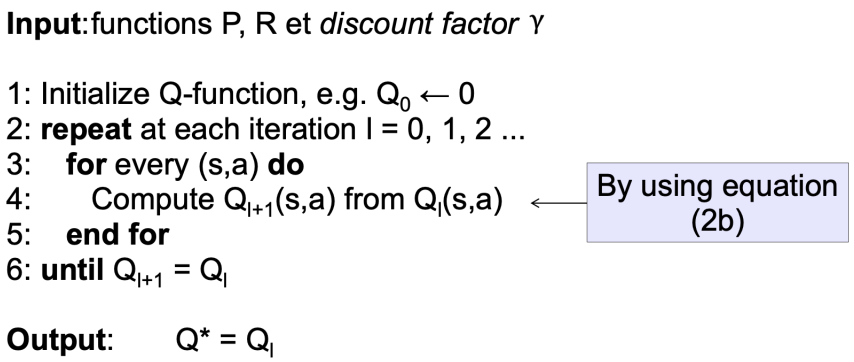

Figure 3: Algorithme QIteration

avec :

$$Q^*(s,a) = E\{r_{t+1} + \gamma max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\}$$
$$= \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma max_{a'} Q^*(s', a')] \qquad \textbf{(2b)}$$

Figure 4: Modification de la valeur Q(s,a)

2.7 Récupération de la politique optimale

A partir de notre dictionnaire Q-table, il est possible d'en extraire la politique optimale. C'est à dire qu'il est nécessaire, pour chaque état, de récupérer l'action optimale pour obtenir notre résultat final le plus rapidmeent possible.

Pour ce faire, il suffit de parcourir chaque action pour chaque état, et récupérer celle qui possède la valeur maximale. Toutes ces politiques optimales sont stockées dans un dictionnaire nommée policy.

```
def extract_optimal_policy(q_table):
    policy = {}
    for state, actions in q_table.items():
        if actions:
            best_action = max(actions, key=actions.get)
            policy[state] = best_action
    return policy
```

2.8 Affichage de la séquence d'actions

Enfin, la dernière étape consiste à affiche la séquences d'actions à réaliser pour arriver à l'état final.

Cela peut être facilement réalisé, car nous disposons déjà de toutes les informations nécessaires grâce

à la fonction extract_optimal_policy. Tant que l'état actuel n'est pas l'état final, nous récupérons l'action optimale à effectuer pour cet état, nous l'ajoutons à notre séquence d'actions, puis nous passons à l'état suivant en appliquant cette action. Ce processus se répète jusqu'à ce que l'état final soit atteint.

```
def get_action_sequence(self, initial_state, policy):
        state = initial_state
        action_sequence = []

        while not self.is_goal_state(state, initial_state):
            action = policy.get(tuple(state))
            if action is None:
                break

            action_sequence.append(action)
            next_state = self.transition(state, action)
            state = next_state

        return action_sequence
```

Ainsi, la politique optimale pour les flèches représentées dans l'exemple de l'exercice est :

```
Séquence d'actions: [(1, 2), (2, 0), (2, 5), (1, 6), (1, 8), (2, 7), (2, 6)]
```

Figure 5: Séquence optimale pour l'état initial [1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1]

(1, 2) correspond à l'action 1 sur la troisième flèche de la liste (2 car l'index commence à 0).
Ainsi de suite pour les prochaines actions.

3 Supervised Learning

L'apprentissage supervisé (Supervised Learning) est une méthode d'apprentissage utilisée dans le domaine du machine learning.

Dans le cadre de ce type d'apprentissage, l'objectif est de contraindre le réseau à adopter un comportement spécifique en ajustant les poids synaptiques, de manière à ce que les sorties du réseau correspondent aux valeurs attendues à partir des entrées fournies.

3.1 Manipulation des données

Dans cette application, nous avons utilisé la régression linéaire pour prédire la charge de chauffage à partir de plusieurs caractéristiques (comme la surface du toit, la surface des murs, etc.) contenues dans un fichier XLS. Les étapes du processus sont détaillées ci-dessous.

3.1.1 Chargement des données

Nous avons d'abord chargé le jeu de données à partir d'un fichier XLS. Ce fichier contient différentes caractéristiques des habitations ainsi que la variable cible, qui est la charge de chauffage.

Voici le code permettant de charger les données à partir du XLS grâce à pandas :

```
file_path = os.path.join('EX1', 'data', 'ENB2012_data.xlsx')
df = pd.read_excel(file_path)
```

3.1.2 Nettoyage des données

Les valeurs manquantes (NaN) ont été supprimées du jeu de données pour éviter d'introduire des biais dans les calculs de régression.

Cela se réalise par l'intermédiaire d'une fonction simple :

```
df_cleaned = df.dropna()
```

3.1.3 Préparation des données

Nous avons séparé les données en deux parties :

- X : Les variables indépendantes, c'est-à-dire toutes les colonnes excepté Y1 et Y2.
- y : La variable cible (Y1), qui est la charge de chauffage.

Voici le code en question :

```
X = df_cleaned.drop(columns=['Y1', 'Y2'])
y = df_cleaned['Y1']
```

3.1.4 Matrice de corrélation

Pour visualiser les relations entre les données et identifier celles qui sont les plus fortement corrélées, nous avons utilisé une matrice de corrélation. Celle-ci présente les données sous forme matricielle, où chaque valeur, comprise entre -1 et 1, indique la force et la direction de la relation linéaire entre deux variables.

- 1 indique une corrélation parfaite entre deux variables.
- -1 indique une corrélation parfaite et négative entre deux variables (l'augmentation d'une provoque la diminution de l'autre).
- 0 indique qu'il n'y a aucune corrélation entre deux variables.

Voici le code en question :

```
numeric_columns = [
    'X1',
    'X2',
    'X3',
    'X4',
    'X5',
    'X6',
    'X7',
    'X8',
    'Y1',
    'Y2'
]

correlation_matrix = df_cleaned[numeric_columns].corr()

plt.figure(figsize=(10, 8))

sns.heatmap(
```

```
    correlation_matrix,
    annot=True,
    cmap='coolwarm',
    fmt='.2f',
    linewidths=0.5
)
plt.title('Matrice de corrélation')
plt.show()
```

Nous obtenons la matrice de corrélation suivante :

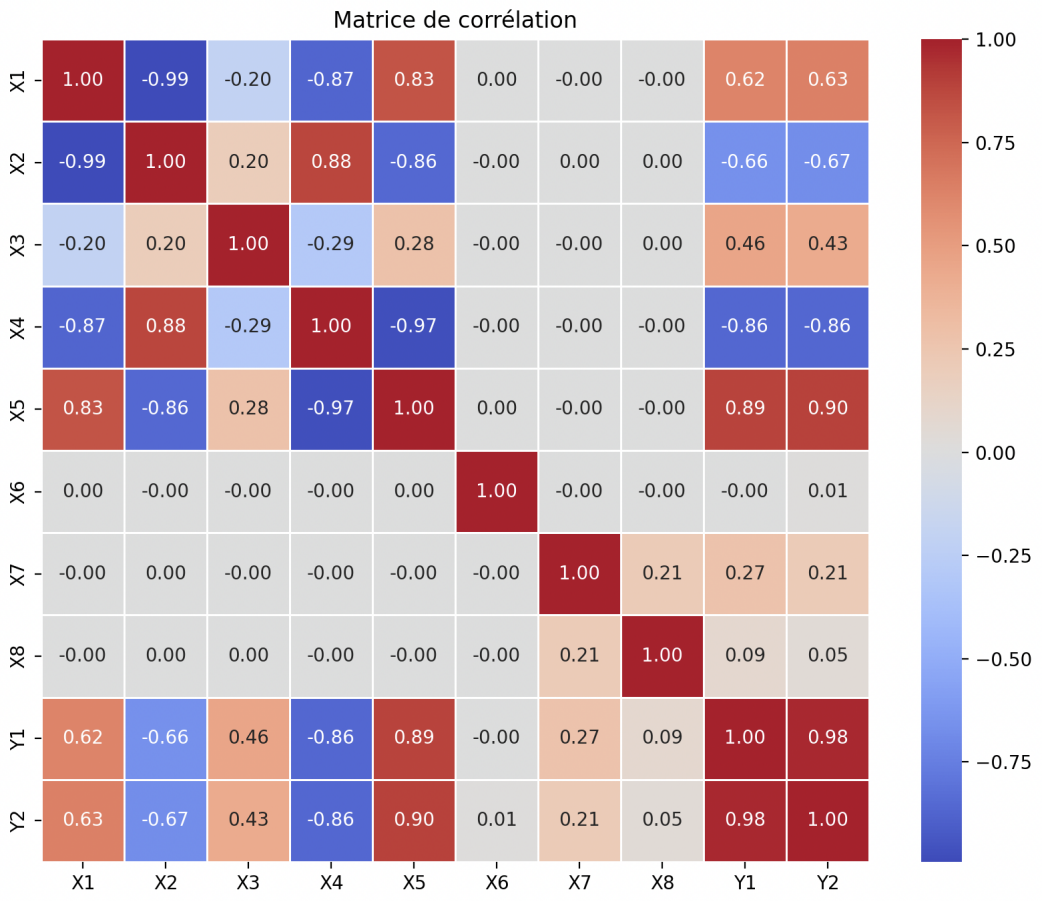

Figure 6: Matrice de corrélation

Nous pouvons voir que les données les plus corrélées sont :

- La compacité relative avec la charge de refroidissement (0.63)
- La surface avec la charge de chauffage (-0.67)
- La surface du toit avec la charge de chauffage ou de refroidissement (-0.86)
- La charge de refroidissement avec la charge de chauffage (0.98)
- etc

3.2 Régression linéaire

3.2.1 Séparation en ensembles d'entraînement et de test

Nous avons divisé les données en ensembles d'entraînement (80%) et de test (20%) pour évaluer la performance du modèle sur des données qu'il n'a pas vues pendant l'entraînement.

Ceci est réalisable simplement par l'intermédiaire de cette fonction :

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
    random_state=42)
```

3.2.2 Normalisation des données

Nous avons normalisé les données d'entrée en utilisant le StandardScaler, ce qui permet d'avoir des caractéristiques avec une moyenne de 0 et une écart-type de 1. Cela est important pour que la régression linéaire ne soit pas influencée par l'échelle des variables.

```
scaler = StandardScaler()
X_train_normalized = scaler.fit_transform(X_train)
X_test_normalized = scaler.transform(X_test)
```

3.2.3 Entraînement du modèle

Nous avons utilisé un modèle de régression linéaire (LinearRegression) pour entraîner le modèle sur les données normalisées d'entraînement.

```
linear_model = LinearRegression()
linear_model.fit(X_train_normalized, y_train)
```

3.2.4 Prédictions et évaluation

Voici le code pour prédire les valeurs :

```
y_pred_train = linear_model.predict(X_train_normalized)
y_pred_test = linear_model.predict(X_test_normalized)
```

Nous pouvons ensuite évaluer sa performance en calculant l'erreur absolue moyenne (MAE) et l'erreur quadratique moyenne (MSE) sur les deux ensembles. Cette métrique nous donne une idée de la différence moyenne entre les valeurs réelles et les valeurs prédites, avec un plus petit MAE ou MSE indiquant une meilleure précision du modèle.

3.2.5 Résultats obtenus

Voici les résultats du MAE et du MSE pour la régression linéaire :

```
Régression linéaire - MSE sur l'ensemble d'entraînement : 8.57
Régression linéaire - MSE sur l'ensemble de test : 9.32
Régression linéaire - MAE sur l'ensemble d'entraînement : 2.07
Régression linéaire - MAE sur l'ensemble de test : 2.18
```

Figure 7: Résultats MSE et MAE pour un modèle de régression linéaire

Nous observons que le MAE sur l'ensemble d'entraînement est relativement bas, ce qui est également le cas pour l'ensemble de test. Cependant, l'erreur est plus élevée pour le MSE, avec des valeurs avoisinant 10. Une erreur de 10 peut être considérée comme importante, surtout que les valeurs réelles de Y1 varient entre 10 et 40 maximum.

3.3 Deep Neuron Network

3.3.1 Création du modèle

Il a été acté de réaliser un réseau de neurones avec deux couches denses de 64 neurones avec des fonctions d'activation ReLU.

Enfin, il a été demandé d'utiliser l'optimisation Adam avec un taux d'apprentissage de 0.01.

Voici le code qui correspond à ces critères :

```
model = Sequential()

model.add(Dense(units=64, activation='relu',
    input_dim=X_train_normalized.shape[1], kernel_regularizer=l2(0.005)))
model.add(Dense(units=64, activation='relu', kernel_regularizer=l2(0.003)))


model.add(Dense(units=1, activation="linear"))

model.compile(optimizer=Adam(learning_rate=0.01), loss='mean_squared_error')

history = model.fit(X_train_normalized, y_train, epochs=100, batch_size=64,
    validation_data=(X_test_normalized, y_test), verbose=1)
```

Des régularisations ont été appliquées pour limiter l'impact des poids excessivement grands sur le modèle. Cela permet d'éviter le surapprentissage (overfitting), en aidant le modèle à mieux se généraliser et à s'adapter à de nouvelles données qu'il n'a pas encore rencontrées.

3.3.2 Résultats obtenus

Voici les courbes de validation loss et de training loss en fonction des epochs :

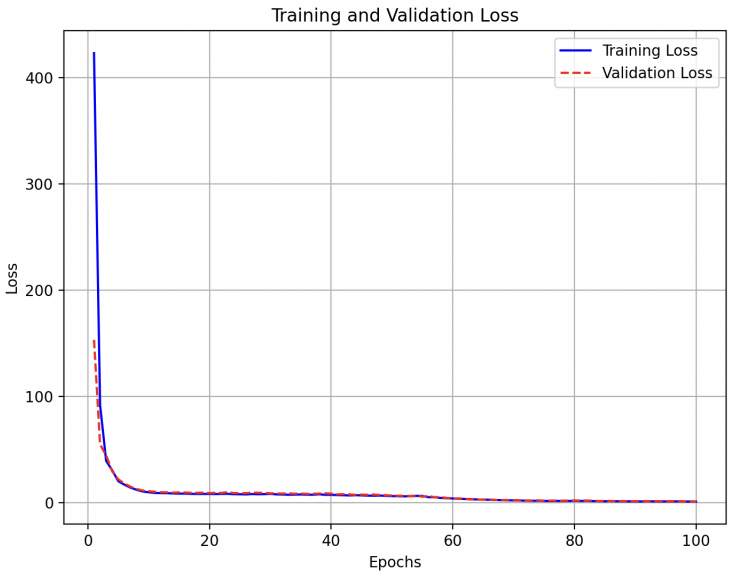

Figure 8: Evolution des courbes de perte en fonction des epochs

Nous pouvons voir que le modèle apprend très rapidement et minimise sa perte en seulement quelques epochs. Dans ce cas précis, le modèle se perfectionne davantage dans les alentours des 60 epochs. Nous pourrions considérer que 60 epochs sont suffisantes pour obtenir des résultats satisfaisant.

De plus, aucun signe de surapprentissage (overfitting) n'est détecté, puisque la courbe de la validation loss ne montre pas de hausse soudaine avec l'augmentation du nombre d'epoch.

Finalement, voici les résultats obtenus pour le MSE :

```
Réseau de neurones - MSE sur l'ensemble d'entraînement : 0.27
Réseau de neurones - MSE sur l'ensemble de test : 0.39
```

Figure 9: MSE pour le réseau de neurones

Et les résultats pour le MAE :

```
Réseau de neurones - MAE sur l'ensemble d'entraînement : 0.93
Réseau de neurones - MAE sur l'ensemble de test : 1.02
```

Figure 10: MAE pour le réseau de neurones

Encore une fois, aucune valeur d'erreur trop élevée du côté de l'ensemble de test pour le MSE ou pour le MAE n'est à noter, il n'y a donc pas de trace d'overfitting.

3.4 Random Forest Regression

Pour le dernier modèle de notre choix concernant un apprentissage supervisé, nous allons choisir la régression par forêt aléatoire (Random Forest Regression).

Chaque arbre de décision dans la forêt est entraîné sur un sous-ensemble aléatoire des données, et la prédiction finale est obtenue en faisant la moyenne des prédictions de tous les arbres. Cette approche permet de réduire le risque de surapprentissage (overfitting) et améliore la précision des prédictions en exploitant la diversité des arbres pour mieux généraliser aux nouvelles données.

3.4.1 Création du modèle

Sklearn est une bibliothèque très puissante étant donné qu'elle permet simplement d'utiliser ce modèle directement par l'intermédiaire de la classe RandomForestRegressor. Ainsi, il suffit simplement de modifier le modèle grâce à cette classe par rapport à l'ancien modèle de régression linéaire :

```
1 rf_model = RandomForestRegressor(n_estimators=100, random_state=42)
2
3 rf_model.fit(X_train_normalized, y_train)
```

Dans ce code, nous utilisons 100 arbres de décision. Le paramètre random_state=42, également utilisé dans la régression linéaire, permet de fixer la graine (seed) du générateur de nombres aléatoires, garantissant ainsi des résultats identiques à chaque exécution du programme.

3.4.2 Résultats obtenus

Voici les résultats de MAE et MSE obtenus :

```
Régression par forêt aléatoire - MAE sur l'ensemble d'entraînement : 0.12
Régression par forêt - MAE sur l'ensemble de test : 0.36
Régression par forêt aléatoire - MSE sur l'ensemble d'entraînement : 0.03
Régression par forêt - MSE sur l'ensemble de test : 0.25
```

Figure 11: MAE et MSE pour la régression par forêt aléatoire

3.5 Comparaison des résultats

Régression linéaire :

```
Régression linéaire - MSE sur l'ensemble d'entraînement : 8.57
Régression linéaire - MSE sur l'ensemble de test : 9.32
Régression linéaire - MAE sur l'ensemble d'entraînement : 2.07
Régression linéaire - MAE sur l'ensemble de test : 2.18
```

Figure 12: Résultats MSE et MAE pour un modèle de régression linéaire

Réseau de neurones :

```
Réseau de neurones - MSE sur l'ensemble d'entraînement : 0.27
Réseau de neurones - MSE sur l'ensemble de test : 0.39
```

Figure 13: MSE pour le réseau de neurones

```
Réseau de neurones - MAE sur l'ensemble d'entraînement : 0.93
Réseau de neurones - MAE sur l'ensemble de test : 1.02
```

Figure 14: MAE pour le réseau de neurones

Régression par forêt aléatoire :

```
Régression par forêt aléatoire - MAE sur l'ensemble d'entraînement : 0.12
Régression par forêt - MAE sur l'ensemble de test : 0.36
Régression par forêt aléatoire - MSE sur l'ensemble d'entraînement : 0.03
Régression par forêt - MSE sur l'ensemble de test : 0.25
```

Figure 15: MAE et MSE pour la régression par forêt aléatoire

Il est évident que la régression linéaire est le modèle le moins performant parmi ceux testés. En revanche, la régression par forêt aléatoire se révèle particulièrement efficace, avec des erreurs proches de zéro. Le modèle de réseau de neurones, bien que performant, est plus complexe et nécessite de nombreux ajustements de poids dans son processus d'apprentissage. Ainsi, la régression par forêt aléatoire semble offrir les meilleurs résultats, tout en équilibrant les performances et les besoins en ressources.

4 Conclusion

Dans un premier temps, nous avons implémenté un MDP afin de trouver une séquence d'actions optimale pour une série de flèches. Cela nous a permis de mettre en application les concepts fondamentaux du processus de décision markovien (MDP), en déterminant les états, les actions, la fonction de transition et en implémentant un algorithme de Q-iteration pour trouver la solution optimale.

Dans un second temps, nous avons développé trois modèles de machine learning dans le cadre d'apprentissages supervisés. Nous avons ainsi implémenté un modèle de régression linéaire, un réseau de neurones avec des couches denses, ainsi qu'un algorithme de régression par forêt aléatoire. Nous avons également abordé la problématique du surapprentissage et appris à y remédier grâce aux techniques de régularisation.

Enfin, nous avons constaté que, dans le cas de notre jeu de données, l'algorithme de régression par forêt aléatoire était le plus efficace.

Bibliographie

[1] Borne P., Benrejeb M., Haggège J. (2007). "Les réseaux de neurones : Présentation et applications." 2007. Editions TECHNIP.

[2] TensorFlow [EN LIGNE]. *Solve your model's overfitting and underfitting problems - Pt.1 (Coding TensorFlow)*. Consulté le 25/11/2024. Lien du site : https://www.youtube.com/watch?v=GMrTBtzJkCg&t=1s

[3] TensorFlow [EN LIGNE]. *Solve your model's overfitting and underfitting problems - Pt.2 (Coding TensorFlow)*. Consulté le 25/11/2024. Lien du site : https://www.youtube.com/results?search_query=overfitting+tensorflow

[4] AssemblyAI [EN LIGNE] *Regularization in a Neural Network | Dealing with overfitting*. Consulté le 25/11/2024. Lien du site : https://www.youtube.com/watch?v=EehRcPo1M-Q&t=628s

[5] Medium [EN LIGNE]. *Reinforcement Learning Explained Visually (Part 4): Q Learning, step-by-step*. Consulté le 25/11/2024. Lien du site : https://towardsdatascience.com/reinforcement-learning-explained-visually-part-4-q-learning-step-by-step-b65efb731d3e

[6] GeeksForGeeks [EN LIGNE]. *Random Forest Regression in Python*. Consulté le 25/11/2024. Lien du site : https://www.geeksforgeeks.org/random-forest-regression-in-python/

[7] Fabrice LAURI - AI53 Machine Learning By Practice [EN LIGNE]. *Reinforcement Learning*. Consulté le 25/11/2024. Lien du site : https://moodle.utbm.fr/pluginfile.php/233344/mod_resource/content/9/AI53-CM01-02.pdf