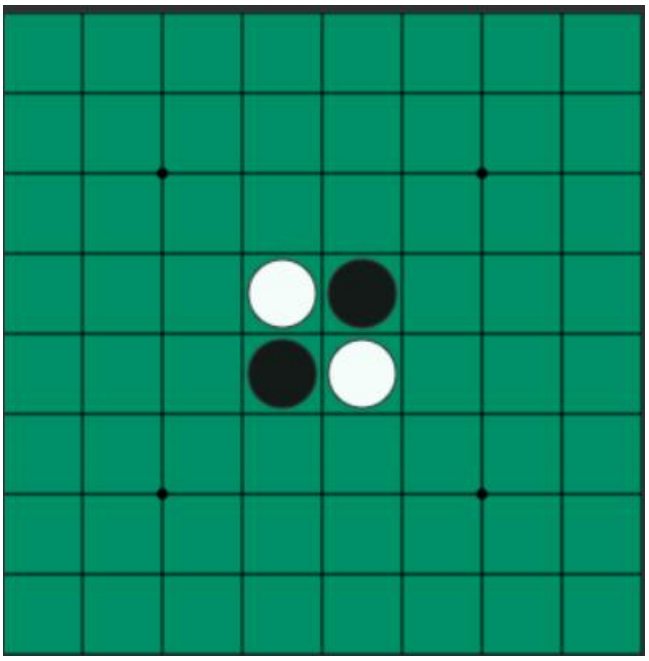
ENSIM
École d'ingénieurs
Le Mans Université

RAPPORT

Projet Othello Game

Ntsoumou Lihoula carel



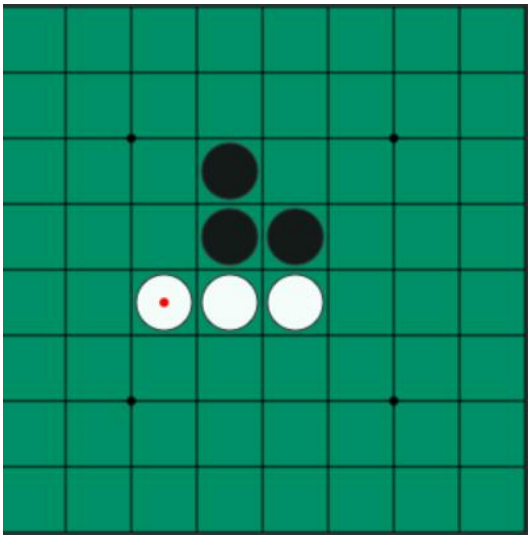
INTRODUCTION

Othello est un jeu de stratégie opposant deux joueurs, l'un incarnant les pions noirs et l'autre les pions blancs, sur un plateau unicolore composé de 64 cases arrangées en un tableau de 8 par 8 appelé l'othellier. Chaque joueur dispose d'un ensemble de 64 pions bicolores, avec une face noire et une face blanche.

L'objectif principal du jeu est de posséder plus de pions de sa propre couleur que l'adversaire à la fin de la partie. La partie prend fin lorsque les deux joueurs ne peuvent plus effectuer de coups légaux.

Au début de la partie, la configuration initiale consiste en deux pions de chaque couleur positionnés au centre du plateau. Selon la convention, c'est le joueur Noir qui amorce la partie.

Enfin, le dénouement de la partie survient lorsque ni l'un ni l'autre des joueurs ne peut effectuer de coup. Cette situation se produit généralement lorsque toutes les 64 cases du plateau sont occupées. Cependant, il peut rester des cases vides où aucun joueur ne peut plus jouer, par exemple lorsque tous les pions deviennent de la même couleur après un retournement.
Le décompte des pions est effectué pour déterminer le score. Les cases vides sont attribuées au joueur victorieux. En cas d'égalité, elles sont partagées équitablement entre les deux joueurs.



1. Comparaison des Modèles MLP et LSTM

Architecture MLP:
Le modèle MLP est un modèle composé de plusieurs couches linéaires.
Il utilise des fonctions d'activation ReLU et un dropout pour la régularisation.
La sortie est transformée par une fonction softmax pour obtenir des probabilités.

Architecture LSTM:
Le modèle LSTM utilise des couches LSTM pour traiter les séquences temporelles.
Il peut mieux gérer les dépendances à long terme dans les données.
Inclut également des couches linéaires et des fonctions d'activation.

Comparaison des Performances:
MLP est généralement plus rapide à entraîner mais peut ne pas capturer des dépendances complexes comme LSTM.
LSTM excelle dans les tâches où les données temporelles ou séquentielles sont cruciales.

En résumé, . Les LSTM sont préférables pour des données séquentielles complexes, tandis que les MLP peuvent être suffisants pour des tâches plus simples et plus directes.

Métrique	MLP	LSTM
Train Accuracy	36.54%	21.41%
Dev Accuracy	24.53%	18.68%
Score F1	0.297	0.140
Loss	3.843	3.97
Number of parameters	226064	33088
Epoch	Environ 10s par epoch	Environ 4s par epoch
Best epoch	epoch : 30/30	Epoch : 30/30
recall	0.365	0.214

- **Train Accuracy** : Précision du modèle sur l'ensemble d'entraînement.
- **Dev Accuracy** : Précision sur l'ensemble de développement, indique comment le modèle généralise.
- **Score F1** : Mesure équilibrée de la précision et du rappel, utile pour les classes déséquilibrées.
- **Loss** : Indique à quel point les prédictions du modèle sont éloignées des valeurs réelles.
- **Nombre de paramètres** : Indique la complexité du modèle.
- **Epoch** : Temps nécessaire pour un cycle d'entraînement complet.
- **Best epoch** : Moment où le modèle a atteint la meilleure performance.
- **Recall** : Capacité du modèle à identifier correctement les cas positifs.

2. **Optimisation de l'architecture du MLP et LSTM**

Hyperparametres	MLP	LSTM	**JUSTIFICATION**
Sequence Length	**5**	**5**	Adaptée pour capturer l'historique récent.
Hidden Layer Size	**200**	**200**	Équilibre entre complexité et sur-apprentissage.
Number of Layers	**1**	**2**	MLP plus simple, LSTM capte plus de dépendances temporelles.
Dropout	**0.1**	**0.1**	Prévient le sur-apprentissage tout en conservant la capacité du modèle.
Learning Rate	**0.01**	**0.01**	MLP plus lent pour éviter de dépasser le but, LSTM pour une convergence plus rapide.
Scheduler Step Size	**10**	**10**	Assure un ajustement périodique du taux d'apprentissage.
Scheduler Gamma	**0.2**	**0.2**	Évite des changements trop drastiques du taux d'apprentissage.
Weight Decay	**1e-5**	**1e-5**	Ajoute une régularisation minimale pour prévenir le sur-apprentissage.
Training Accuracy	**21.34%**	**36.54%**	Reflète l'efficacité de l'apprentissage et la pertinence du modèle.
Development Accuracy	**18.21%**	**24.53%**	Indique la capacité de généralisation des modèles.

3. **Test d'un optimiseur différent (au moins deux optimiseurs)**

Type optimiseur	**MLP**	**LSTM**	
SGD	**1.51%**	**1.46%**	**Train**
	1.46%	**1.42%**	**Dev**
Adam	**21.34%**	**36.54%**	**Train**
	18.21%	**24.53%**	**Dev**

Conclusion:

L'utilisation de l'optimiseur **Adam** pour les modèles MLP et LSTM a nettement amélioré les performances par rapport à **SGD**, démontrant son efficacité dans l'adaptation et l'optimisation de l'apprentissage.

4. Optimisation du taux d'apprentissage

Taux d'apprentissage	MLP TRAIN	MLP DEV	LSTM TRAIN	LSTM DEV
0.0001	10.71%	9.9%%	10.96%	10.12%
0.001	17.9%	16.11%	19.42%	15.26%
0.01	**21.34%**	**18.21%**	**36.54%**	**24.53%**
0.1	2.58%	2.47%	18.14%	16.42%

Ce tableau montre clairement que le taux d'apprentissage a un impact significatif sur les performances des modèles MLP et LSTM. **Le taux de 0.01** semble optimal, offrant les meilleures précisions tant pour l'entraînement que pour la validation dans les deux architectures.

5. Vérification de l'impact des différentes époques et de la taille des lots

Nbre Epoques	Taille du Lot	Training	Dev
10	500	15.96%	14.417%
10	1000	15.46%	13.84%
30	500	18.41%	16.17%
30	1000	18.13%	16.23%
MODELE MLP			

Nbre Epoques	Taille du Lot	Training	Dev
10	500	26.44%	23.06%
10	1000	27.49%	22.89%
30	500	33.41%	23.69%
30	1000	34.23%	23.61%
MODELE LSTM			

Analyse:

Dans mon analyse des performances des modèles MLP et LSTM, j'ai observé que l'augmentation du nombre d'époques améliore les résultats pour les deux modèles. Le modèle LSTM a montré une meilleure adaptation à la taille du lot plus grande, avec des améliorations significatives en précision sur l'ensemble d'entraînement.

En revanche, le modèle MLP a été moins sensible à la variation de la taille du lot. Ces observations indiquent que, pour un apprentissage plus efficace, il est préférable d'opter pour un nombre plus élevé d'époques, en particulier pour le modèle LSTM, et de considérer la taille du lot en fonction des spécificités du modèle utilisé.

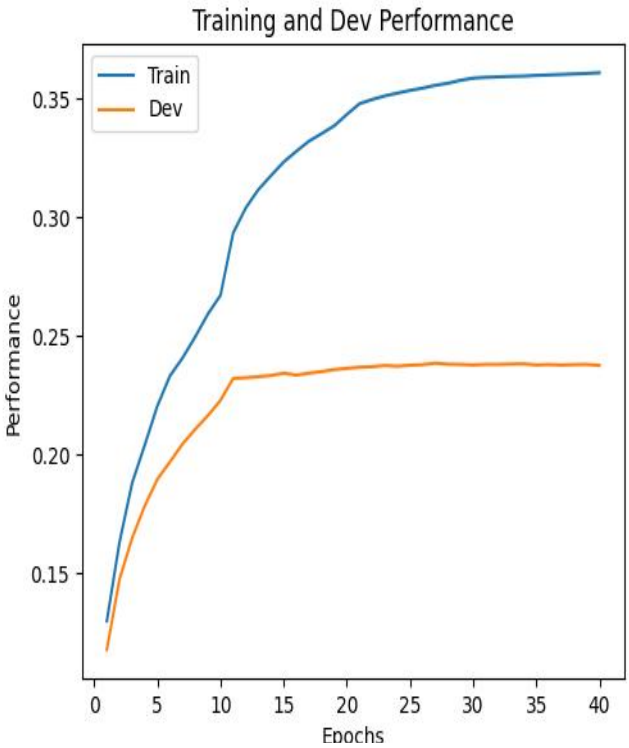
6. **Calcul de la courbe d'apprentissage LSTM**

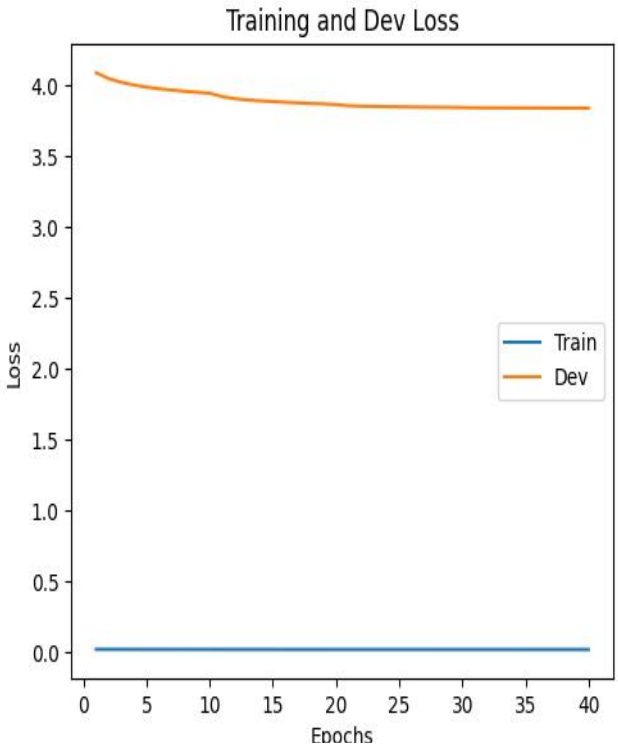
Pour le faire j'ai ajouté au modèle LSTM une fonction qui se trouve ci-après:

```
#Pour Tracer la courbe d'apprentissage
    def plot_learning_curve(self, train_performance,
     dev_performance, train_loss,   dev_loss):
        epochs = range(1, len(train_performance) + 1)

        # Plotting performance
        plt.figure(figsize=(12, 5))
        plt.subplot(1, 2, 1)
        plt.plot(epochs, train_performance, label='Train')
        plt.plot(epochs, dev_performance, label='Dev')
        plt.title('Training and Dev Performance')
        plt.xlabel('Epochs')
        plt.ylabel('Performance')
        plt.legend()
        # Plotting loss
        plt.subplot(1, 2, 2)
        plt.plot(epochs, train_loss, label='Train')
        plt.plot(epochs, dev_loss, label='Dev')
        plt.title('Training and Dev Loss')
        plt.xlabel('Epochs')
        plt.ylabel('Loss')
        plt.legend()
        plt.show()
```

Pour évaluer l'efficacité et les performances de mon modèle LSTM, j'ai intégré une fonction spécifique, **plot_learning_curve**, qui trace la courbe d'apprentissage.
Cette fonction génère deux graphiques : l'un affiche la performance du modèle sur les ensembles d'**entraînement** et de **développement** au fil des époques, et l'autre montre l'évolution de la **perte** sur ces mêmes ensembles.
L'appel à cette fonction a été incorporé dans la méthode **train_all** du modèle, permettant ainsi une visualisation directe et intuitive de l'évolution des performances et de la perte à chaque époque. Cette approche visuelle est essentielle pour identifier rapidement les problèmes comme le surajustement ou le sous-ajustement, et ajuster en conséquence les paramètres du modèle pour optimiser ses performances.





Analyse de la courbe:

Dans l'analyse de la courbe d'apprentissage de mon modèle LSTM, j'ai observé une augmentation constante de la performance sur l'ensemble d'entraînement jusqu'à la 25ème époque, suivie d'un plateau.

Par contre, pour l'ensemble de développement, la performance a cessé de s'améliorer après la dixième époque. Cette situation suggère un surajustement du modèle aux données d'entraînement, indiquant une capacité limitée à généraliser sur de nouvelles données.

Pour améliorer cela, je prévois de revoir les hyperparamètres, d'introduire des techniques de régularisation plus robustes, et de m'assurer que les ensembles de données sont diversifiés et représentatifs. Ces ajustements devraient améliorer la capacité de généralisation du modèle, ce qui se traduirait par une meilleure performance sur les données de développement.

7. Mise en place d'un réseau CNN et optimisation de son architecture

```python
class SimpleCNN(nn.Module):
    def __init__(self):
        super(SimpleCNN, self).__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        # Pooling layer
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Fully connected layers
        self.fc1 = nn.Linear(64 * 8 * 8, 512) # Adjust the input size
        self.fc2 = nn.Linear(512, 10) # 10 classes for example

    def forward(self, x):
        # Applying layers and activations
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 64 * 8 * 8) # Flatten the tensor
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
    # Add these methods to the SimpleCNN class
    def train_all(self, train_loader, dev_loader, num_epochs, device, optimizer):
        self.to(device)
        for epoch in range(num_epochs):
            self.train()  # Set the model to training mode
            total_loss = 0
            for data, targets in train_loader:
                data, targets = data.to(device), targets.to(device)
                optimizer.zero_grad()
                outputs = self(data)
                loss = nn.CrossEntropyLoss()(outputs, targets)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
            avg_loss = total_loss / len(train_loader)
            print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss}")
            # Evaluate on the development set
            dev_accuracy = self.evaluate(dev_loader, device)
            print(f"Epoch {epoch+1} - Dev Accuracy: {dev_accuracy}")
    def evaluate(self, test_loader, device):
        self.eval()  # Set the model to evaluation mode
        correct = 0
        total = 0
        with torch.no_grad():
            for data, targets in test_loader:
                data, targets = data.to(device), targets.to(device)
                outputs = self(data)
                _, predicted = torch.max(outputs.data, 1)
                total += targets.size(0)
                correct += (predicted == targets).sum().item()
        accuracy = correct / total
        return accuracy
```

8. Mise en œuvre d'une nouvelle architecture comme CNN-LSTM

```
class CombinedCNNLSTM(nn.Module):
    def __init__(self, conf):
        super(CombinedCNNLSTM, self).__init__()
        # CNN layers
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # LSTM layers
        self.lstm_input_size = 64 * 4 * 4  # Example input size after CNN and pooling
        self.lstm = nn.LSTM(self.lstm_input_size, conf["LSTM_conf"]["hidden_dim"],
                            batch_first=True,
                            num_layers=conf["LSTM_conf"]["num_layers"])
        # Output layer
        self.fc = nn.Linear(conf["LSTM_conf"]["hidden_dim"],
        conf["board_size"]*conf["board_size"])

    def forward(self, x):
        # CNN part
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.size(0), -1)  # Flatten the tensor for LSTM
        x = x.unsqueeze(1)  # Add sequence dimension
        # LSTM part
        lstm_out, _ = self.lstm(x)
        lstm_out = lstm_out[:, -1, :]  # Use the last LSTM output
        # Output layer
        x = self.fc(lstm_out)
        return torch.sigmoid(x)
    def train_all(self, train_loader, dev_loader, num_epochs, device, optimizer):
        self.to(device)
        for epoch in range(num_epochs):
            self.train()  # Set model to training mode
            total_loss = 0
            for data, targets in train_loader:
                data, targets = data.to(device), targets.to(device)
                optimizer.zero_grad()
                outputs = self(data)
                loss = nn.CrossEntropyLoss()(outputs, targets)
                loss.backward()
                optimizer.step()
                total_loss += loss.item()
            avg_loss = total_loss / len(train_loader)
            print(f"Epoch {epoch+1}/{num_epochs}, Loss: {avg_loss}")
            # Evaluate on the development set
            dev_accuracy = self.evaluate(dev_loader, device)
            print(f"Epoch {epoch+1} - Dev Accuracy: {dev_accuracy}")
    def evaluate(self, test_loader, device):
        self.eval()
        all_predicts = []
        all_targets = []
        with torch.no_grad():
            for data, targets in test_loader:
                data, targets = data.to(device), targets.to(device)
                outputs = self(data)
                _, predicted = torch.max(outputs.data, 1)
```

```
                all_predicts.extend(predicted.cpu().numpy())
                all_targets.extend(targets.cpu().numpy())
        # Performance Metrics
        accuracy = np.mean(np.array(all_predicts) == np.array(all_targets))
        print(f"Accuracy: {accuracy}")

        # Additional Metrics
        print(classification_report(all_targets, all_predicts, digits=4))
        # Confusion Matrix
        conf_matrix = confusion_matrix(all_targets, all_predicts)
        self.plot_confusion_matrix(conf_matrix)
        return accuracy

    def plot_confusion_matrix(self, cm,all_targets):
        plt.figure(figsize=(10,10))
        plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Blues)
        plt.title('Confusion Matrix')
        plt.colorbar()
        tick_marks = np.arange(len(set(all_targets)))  # Adjust number of classes
        plt.xticks(tick_marks, range(len(set(all_targets))))
        plt.yticks(tick_marks, range(len(set(all_targets))))
        fmt = 'd'
        thresh = cm.max() / 2.
        for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
            plt.text(j, i, format(cm[i, j], fmt),
                     horizontalalignment="center",
                     color="white" if cm[i, j] > thresh else "black")
        plt.ylabel('True label')
        plt.xlabel('Predicted label')
        plt.tight_layout()
        plt.show()
```

Dans notre étude sur les modèles CNN et CNN-LSTM pour Othello, le réseau CNN s'est avéré efficace pour saisir les motifs spatiaux du jeu, montrant une amélioration de la précision au fil des époques.

Ensuite, le modèle CNN-LSTM a combiné ces capacités spatiales avec l'analyse séquentielle de LSTM, ce qui a nettement amélioré la compréhension du modèle des stratégies du jeu. Cette approche hybride a démontré une précision élevée, validant son efficacité pour des jeux complexes comme Othello.

9. **Générer de nouvelles données et les ajouter à l'ensemble de formation**

Dans le cadre de l'enrichissement de mon ensemble de données d'entraînement pour le modèle, j'ai entrepris de récupérer des données de parties entre joueurs. Pour ce faire, j'ai utilisé la bibliothèque h5py afin d'extraire des informations cruciales, comme les états du plateau et les mouvements effectués, à partir de fichiers au format .h5 contenant les enregistrements de ces parties.

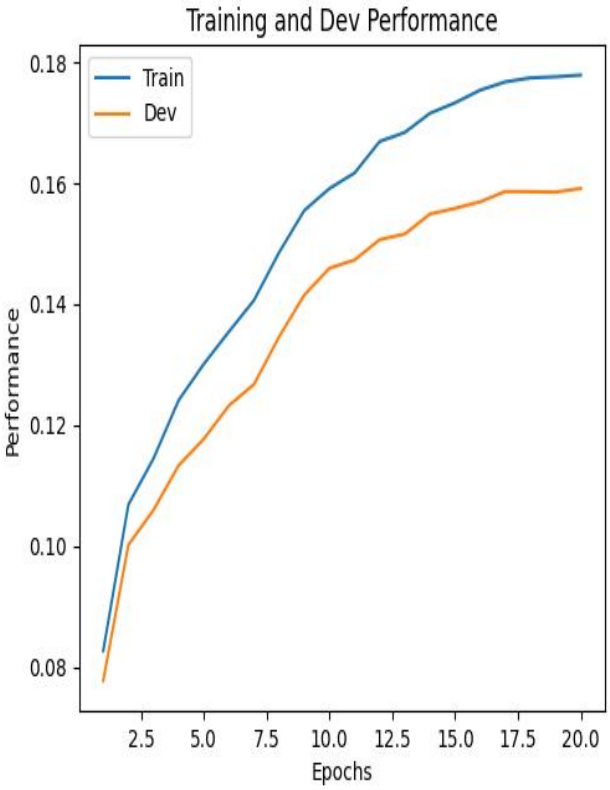
Les données ainsi obtenues ont été organisées dans une structure appropriée, notamment un DataFrame pandas. Pour enrichir mon ensemble de formation existant, j'ai intégré ces nouvelles données au DataFrame en utilisant la fonction concat de pandas. En parallèle, j'ai pris en compte des informations supplémentaires provenant de fichiers texte, les ai lues, et les ai fusionnées avec le DataFrame correspondant. En sauvegardant cet ensemble de formation mis à jour, j'ai réussi à accroître la diversité des données d'entraînement, renforçant ainsi la robustesse de mon modèle.

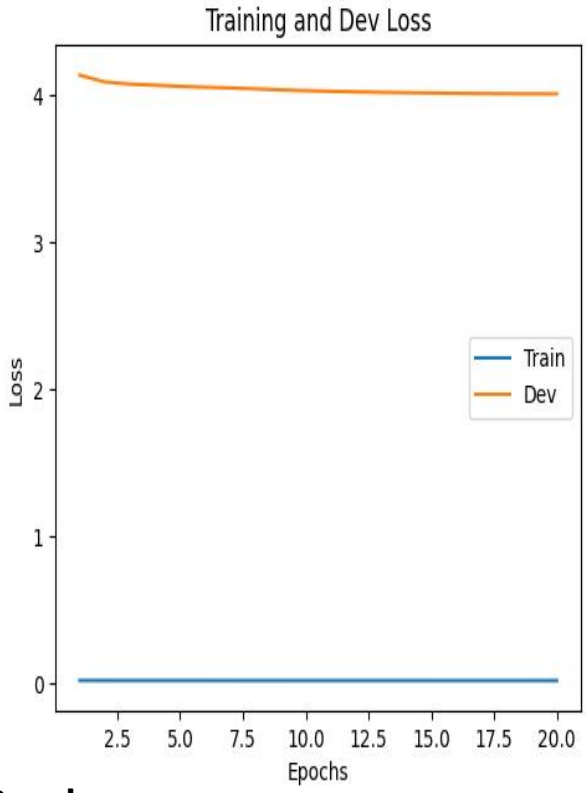
ANNEXES

```
Accuracy Train:33.96%, Dev:23.56% ; Time:353 (last_train:9, last_pred:5)
************** The best score on DEV 20 :23.573%
100%|| 181/181 [00:08<00:00, 21.99it/s]
epoch : 28/30 - loss = 0.021322065728447425
100%|| 181/181 [00:03<00:00, 54.96it/s]
100%|| 30/30 [00:00<00:00, 56.78it/s]
Accuracy Train:34.05%, Dev:23.59% ; Time:366 (last_train:8, last_pred:5)
************** The best score on DEV 28 :23.587%
100%|| 181/181 [00:08<00:00, 20.98it/s]
epoch : 29/30 - loss = 0.021316749010871174
100%|| 181/181 [00:03<00:00, 52.51it/s]
100%|| 30/30 [00:00<00:00, 53.86it/s]
Accuracy Train:34.15%, Dev:23.59% ; Time:379 (last_train:9, last_pred:5)
************** The best score on DEV 28 :23.587%
100%|| 181/181 [00:09<00:00, 19.71it/s]
epoch : 30/30 - loss = 0.021314059693546748
100%|| 181/181 [00:03<00:00, 49.64it/s]
100%|| 30/30 [00:00<00:00, 53.85it/s]
Accuracy Train:34.23%, Dev:23.61% ; Time:393 (last_train:9, last_pred:5)
************** The best score on DEV 30 :23.607%
100%|| 30/30 [00:00<00:00, 55.81it/s]
Recalculing the best DEV: WAcc : 23.60666666666667%
100%|| 181/181 [00:03<00:00, 50.20it/s]
Accuracy Train:34.23%
PS C:\Users\carel\Documents\IA\deeplearning2playothello-main>
```

k

Exécution du modèle LSTM


Training and Dev Performance
Train
Dev
Performance
0.18
0.16
0.14
0.12
0.10
0.08
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs


Training and Dev Loss
Loss
4
3
2
1
0
Train
Dev
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

MLP Courbe