J'ai modifié le nombre d'epoch de 200 à 20 pour réduire le temps d'entraînement.

Les valeurs peuvent différer avec un même modèle car j'ai exécuté certains entraînements sur mon pc portable avec le CPU et d'autres sur mon GPU Nvidia.

Calculate and Compare the results of provided MLP and LSTM baseline

Pour les modèles MLP et LSTM de base, j'ai obtenu les résultats suivants après 20 epochs d'entraînement.

```
# Modèle de base MLP
conf["board_size"]=8
conf["path_save"]="save_models"
conf['epoch']=20
conf["earlyStopping"]=20
conf["len_inpout_seq"]=1
conf["dropout"]=0.1

Batch size = 1000

Optimizer: Adam, learning rate: 0.001

Nombre de poids entraînables MLP: 33088

best DEV: Accuracy : 14.97%
Fin entrainement MLP sur 20 epoch en (109.52468705177307, 'sc')
```

```
# Modèle de base LSTM
conf["board_size"]=8
conf["path_save"]="save_models"
conf['epoch']=20
conf["earlyStopping"]=20
conf["len_inpout_seq"]=len_samples
conf["LSTM_conf"]={}
conf["LSTM_conf"]["hidden_dim"]=256
conf["dropout"]=0.1

Batch size = 1000

Optimizer: Adam, learning rate: 0.005

Nombre de poids entraînables LSTM: 346176

best DEV: Accuracy : 22.073333333333334%
```

```
Fin entrainement LSTM sur 20 epoch en (403.4299657344818, 'sc')
```

On remarque que le modèle LSTM performe mieux que le modèle MLP de base mais demande un temps d'entraînement 4x plus long (on a 67% de performance en plus pour un temps d'entraînement 4x plus long). On aussi un nombre de paramètres entraînables beaucoup plus élevé pour le LSTM (346176 contre 33088 pour le MLP), ce qui explique en partie le temps d'entraînement plus long.

Exécution de game.py

J'ai testé les deux modèles de base en les faisant jouer des parties l'un contre l'autre, LSTM gagne à chaque fois.

Optimizing the architecture of MLP (reporting the number of trainable weights is necessary)

Choix du nombre de couches

Je vais faire des MLP avec les couches suivantes pour voir la différence. Cependant, plus il y a de couches plus le temps de calcul augmente.

```
[64],               # 1 couche
[128],               # 1 couche
[256, 128],          # 2 couches
[512, 256, 128],     # 3 couches
[512, 256, 128, 64] # 4 couches
```

Les paramètres du modèle sont les suivants pour chaque architecture testée:

```
conf["board_size"]=8
conf["path_save"]="save_models"
conf['epoch']=20
conf["earlyStopping"]=20
conf["len_inpout_seq"]=1
conf["dropout"]=0.1

Batch size = 1000

Optimizer: Adam, learning rate: 0.001
```

Couches (entrainement sur 20 epochs): - [64] : 12480 poids, temps d'entraînement 101sc, Accuracy dev 12.38% - [128] : 33088 poids, temps d'entraînement 109sc, Accuracy dev 14.97% - **[256, 128]** : 57792 poids, temps d'entraînement 151sc, Accuracy dev 15.53% - [512, 256, 128] : 205760 poids, temps d'entraînement 333sc, Accuracy dev 13.78% - [512, 256, 128, 64] : 209920 poids, temps d'entraînement 429sc, Accuracy dev 10.43%

D'après les données ci-dessus, on remarque que l'augmentation du nombre de couches améliore les performances du modèle MLP jusqu'à une certaine limite. En effet, le modèle avec deux couches ([256, 128]) offre la meilleure précision (15.53%) par rapport aux autres architectures testées. Cependant, au-delà de deux couches, les performances diminuent, ce qui peut être attribué à un surapprentissage ou à une complexité excessive du modèle. De plus, le temps d'entraînement augmente significativement avec le nombre de couches, ce qui peut ne pas être justifié par les gains de performance. Ainsi, pour ce jeu de données et cette tâche spécifique, une architecture MLP avec deux couches semble être le compromis optimal entre complexité et performance. J'ai aussi essayé de réduire le nombre de neurones par couche, mais les performances ont diminué et le temps de calcul n'a pas beaucoup changé.

Changement des paramètres d'entraînement

Maintenant que j'ai choisi l'architecture [256, 128] pour le MLP, je vais modifier les batch size, optimizer et le learning rate pour voir si je peux améliorer les performances. J'ai aussi changé "earlyStopping" à 5 pour réduire le temps d'entraînement.

Batch Size

- 128: 57792 poids, Temps d'entraînement 498sc, Temps pas epoch 16sc, Accuracy dev 14.07%
- 1000: 57792 poids, Temps d'entraînement 122sc, Temps pas epoch 3sc, Accuracy dev 14.54%
- 1500: 57792 poids, Temps d'entraînement 98sc, Temps pas epoch 2sc, Accuracy dev 14.59%
- **2000**: 57792 poids, Temps d'entraînement 79sc, Temps pas epoch 1sc, Accuracy dev 14.69%
- 3000: 57792 poids, Temps d'entraînement 80sc, Temps pas epoch 1sc, Accuracy dev 14.5%
- 5000: 57792 poids, Temps d'entraînement 68sc, Temps pas epoch 1sc, Accuracy dev 14%
- 15000: 57792 poids, Temps d'entraînement 74sc, Temps pas epoch 1sc, Accuracy dev 11.11%

On remarque que l'augmentation du batch size réduit le temps d'entraînement par epoch, mais au-delà de 2000, les performances commencent à diminuer. Le batch size optimal semble être 2000 pour cet entraînement.

Optimizer et Learning Rate

J'ai testé les optimizers suivants avec un learning rate ayant les valeurs suivantes [0.0001, 0.001, 0.01, 0.1] et un dropout de 0.1. J'ai testé les combinaisons suivantes: - Adam x [0.0001, 0.001, 0.01, 0.1] - Adagrad x [0.0001, 0.001, 0.01, 0.1] - SGD x [0.0001, 0.001, 0.01, 0.1]

Pour chaque combinaison, le nombre de poids est de 57792 et le batch size est de 2000.

Résultats (entrainement sur 20 epochs): Adam:

```
Recalculing the best DEV: WAcc : 10.703333333333333%
Fin entrainement MLP_256_128 sur 20 epoch en (78.66891503334045, 'sc') | Paramètres: Learnir

Recalculing the best DEV: WAcc : 14.766666666666667%
Fin entrainement MLP_256_128 sur 20 epoch en (121.54966473579407, 'sc') | Paramètres: Learni

Recalculing the best DEV: WAcc : 9.133333333333333%
Fin entrainement MLP_256_128 sur 7 epoch en (43.20238661766052, 'sc') | Paramètres: Learning

Recalculing the best DEV: WAcc : 2.5233333333333334%
Fin entrainement MLP_256_128 sur 11 epoch en (64.8180239200592, 'sc') | Paramètres: Learning
```

Pour Adam, le meilleur learning rate est 0.001 avec une accuracy de 13.38%. On remarque que pour un learning rate trop élevé (0.1), les performances chutent drastiquement.

Adagrad:

```
Recalculing the best DEV: WAcc : 1.68%
Fin entrainement MLP_256_128 sur 7 epoch en (28.50636625289917, 'sc') | Paramètres: Learning

Recalculing the best DEV: WAcc : 8.426666666666668%
Fin entrainement MLP_256_128 sur 20 epoch en (116.77517032623291, 'sc') | Paramètres: Learni

Recalculing the best DEV: WAcc : 13.106666666666666%
Fin entrainement MLP_256_128 sur 20 epoch en (119.46998238563538, 'sc') | Paramètres: Learni

Recalculing the best DEV: WAcc : 5.083333333333333%
Fin entrainement MLP_256_128 sur 20 epoch en (119.97065329551697, 'sc') | Paramètres: Learni
```

Pour Adagrad, les performances sont médiocres quel que soit le learning rate, avec une accuracy maximale de 1.7%.

SGD:

```
Recalculing the best DEV: WAcc : 1.6833333333333331%
Fin entrainement MLP_256_128 sur 7 epoch en (27.083165168762207, 'sc') | Paramètres: Learnir

Recalculing the best DEV: WAcc : 1.6633333333333333%
Fin entrainement MLP_256_128 sur 7 epoch en (41.83458495140076, 'sc') | Paramètres: Learning

Recalculing the best DEV: WAcc : 1.6%
Fin entrainement MLP_256_128 sur 20 epoch en (119.97952938079834, 'sc') | Paramètres: Learni
```

```
Recalculing the best DEV: WAcc : 2.736666666666667%
Fin entrainement MLP_256_128 sur 20 epoch en (119.93522930145264, 'sc') | Paramètres: Learni
```

Pour SGD, les performances sont également très faibles quel que soit le learning rate, avec une accuracy maximale de 1.68%.

Conclusion

D'après les résultats obtenus, l'optimizer Adam avec un learning rate de 0.001 offre les meilleures performances pour le modèle MLP avec une architecture [256, 128]. Les autres optimizers, Adagrad et SGD, ne parviennent pas à atteindre des performances comparables, quel que soit le learning rate utilisé. Pour la suite des optimisations, je vais donc utiliser Adam avec un learning rate de 0.001, un batch size de 2000 et un dropout de 0.1.

Vérification des performances finales (vérification de l'impact de l'augmentation du nombre d'epochs)

J'ai relancé l'entraînement du modèle MLP avec les meilleurs paramètres trouvés (architecture [256, 128], optimizer Adam, learning rate 0.001, dropout 0.1, batch size 2000) pour vérifier les performances. J'ai exécuté l'entraînement sur 200 epochs avec earlyStopping à 20.

```
Recalculing the best DEV: WAcc : 16.74666666666667%
Fin entrainement MLP_256_128 sur 200 epoch en (829.7159850597382, 'sc') | Paramètres: Learni
```

On obtient une accuracy de 16.75% sur le dev set, ce qui confirme que les paramètres choisis sont efficaces pour améliorer les performances du modèle MLP par rapport à la version de base (14.97%). Le gain est cependant assez modeste malgré l'augmentation significative du temps d'entraînement (829sc contre 109sc pour la version de base).

Optimizing the architecture of LSTM (reporting the number of trainable weights is necessary)

Un LSTM peut être un LSTM avec hidden state ou un LSTM avec output sequence.

On testera les deux types sur 20 epochs.

Pour le moment, on aura un batch size de 1000, un optimizer Adam, un learning rate de 0.005 et un dropout de 0.1.

Voilà les paramètres à optimiser pour le LSTM: - nombre de couches/hidden_dim - type de LSTM (hidden state ou output sequence)

Changement des paramètres d'entraînement

Choix du nombre de couches

Hidden State LSTM Je vais faire des LSTM avec les couches suivantes pour voir la différence. Cependant, plus il y a de couches plus le temps de calcul augmente.

```
[64],               # 1 couche
[256],               # 1 couche
[512, 256],          # 2 couches
[512, 256, 128],     # 3 couches
```

Couches (entrainement sur 20 epochs): - [64] : 41536 poids, temps d'entraînement 143sc, Accuracy dev 35.05% - [256] : 362560 poids, temps d'entraînement 168sc, Accuracy dev 34.26% - [512, 256] : 2005056 poids, temps d'entraînement 185sc, Accuracy dev 32.41% - [512, 256, 128] : 2186304 poids, temps d'entraînement 169sc, Accuracy dev 28.59%

On peut voir qu'augmenter le nombre de couches n'améliore pas les performances du modèle LSTM avec hidden state. Le meilleur modèle est celui avec une seule couche de 64 neurones, qui offre une accuracy de 35.05% sur le dev set. De plus, le temps d'entraînement n'augmente pas de manière significative avec l'ajout de couches supplémentaires, ce qui suggère que la complexité accrue du modèle ne se traduit pas par une amélioration des performances pour cette tâche spécifique.

Output Sequence LSTM Je vais tester les mêmes architectures que pour le Hidden State LSTM.

```
[64],               # 1 couche
[256],               # 1 couche
[512, 256],          # 2 couches
[512, 256, 128],     # 3 couches
```

Couches (entrainement sur 20 epochs): - [64] : 37440 poids, temps d'entraînement 123sc, Accuracy dev 18.46% - [256] : 346176 poids, temps d'entraînement 177sc, Accuracy dev 22.65% - [512, 256] : 1988672 poids, temps d'entraînement 198sc, Accuracy dev 20.12% - [512, 256, 128] : 2178112 poids, temps d'entraînement 207sc, Accuracy dev 11.6%

On peut voir qu'augmenter le nombre de couches n'améliore pas les performances du modèle LSTM avec output sequence. Le meilleur modèle est celui avec une seule couche de 256 neurones, qui offre une accuracy de 22.65% sur le dev set. De plus, le temps d'entraînement augmente légèrement avec l'ajout de couches supplémentaires, mais cela ne se traduit pas par une amélioration des performances pour cette tâche spécifique.

Comparaison des deux types de LSTM On remarque que le LSTM avec hidden state performe mieux que le LSTM avec output sequence pour toutes les

architectures testées. Le meilleur modèle global est le LSTM avec hidden state et une seule couche de 64 neurones, qui offre une accuracy de 35.05% sur le dev set, contre 22.65% pour le meilleur modèle LSTM avec output sequence (une seule couche de 256 neurones). Cela suggère que pour cette tâche spécifique, le LSTM avec hidden state est plus adapté que le LSTM avec output sequence. Pour la suite de l'optimisation, je vais me concentrer sur le LSTM avec hidden state.

Batch Size

- 128: 41536 poids, Temps d'entraînement 642sc, Temps pas epoch 20sc, Accuracy dev 36.30%
- 500: 41536 poids, Temps d'entraînement 215sc, Temps pas epoch 6sc, Accuracy dev 36.96%
- 1000: 41536 poids, Temps d'entraînement 123sc, Temps pas epoch 3sc, Accuracy dev 35.05%
- 1500: 41536 poids, Temps d'entraînement 123sc, Temps pas epoch 3sc, Accuracy dev 33.41%
- 2000: 41536 poids, Temps d'entraînement 116sc, Temps pas epoch 2sc, Accuracy dev 32.82%
- 3000: 41536 poids, Temps d'entraînement 93sc, Temps pas epoch 2sc, Accuracy dev 31.14%
- 5000: 41536 poids, Temps d'entraînement 86sc, Temps pas epoch 1sc, Accuracy dev 27.66%
- 15000: 41536 poids, Temps d'entraînement 80sc, Temps pas epoch 1sc, Accuracy dev 19.86%

Concernant le batch size, on remarque que le batch size de 500 offre le meilleur compromis entre temps d'entraînement et performance, avec une accuracy de 36.96% sur le dev set. Les batch sizes plus petits (128) augmentent considérablement le temps d'entraînement sans amélioration significative des performances, tandis que les batch sizes plus grands (2000 et plus) entraînent une diminution des performances en échange d'un temps d'entraînement considérablement réduit.

On va donc choisir un batch size de 500 pour la suite des optimisations.

Optimizer et Learning Rate

J'ai testé les optimizers suivants avec un learning rate ayant les valeurs suivantes [0.0001, 0.005, 0.001, 0.05, 0.01] et un dropout de 0.1. J'ai testé les combinaisons suivantes: - Adam x [0.0001, 0.005, 0.001, 0.05, 0.01] - Adagrad x [0.0001, 0.005, 0.001, 0.05, 0.01] - SGD x [0.0001, 0.005, 0.001, 0.05, 0.01]

Pour chaque combinaison, le nombre de poids est de 41536 (LSTM hidden state avec une couche de 64 neurones) et le batch size est de 500.

J'ai mis "earlyStopping" à 5 pour réduire le temps d'entraînement.

Résultats (entrainement sur 20 epochs): Adam:

```
Recalculing the best DEV: WAcc : 14.08%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (167.4393014907837, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 35.31333333333333%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (191.59716272354126, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 28.18%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (227.24093437194824, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 28.376666666666665%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (227.55478191375732, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 35.27666666666667%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (229.6387701034546, 'sc') | Paramètres:
```

Pour Adam, le meilleur learning rate est 0.005 avec une accuracy de 35.31%. On remarque que les learning rates de 0.001 et 0.01 offrent également de bonnes performances, tandis que le learning rate de 0.0001 est nettement inférieur.

Adagrad:

```
Recalculing the best DEV: WAcc : 3.8066666666666666%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (212.83428311347961, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 15.406666666666666%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (188.96928548812866, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 9.46%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (169.83400201797485, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 29.91333333333333%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (168.66772079467773, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 19.173333333333336%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (170.89515042304993, 'sc') | Paramètres:
```

Pour Adagrad, le meilleur learning rate est 0.05 avec une accuracy de 29.91%. Cependant, les performances restent inférieures à celles obtenues avec l'optimizer Adam.

SGD:

```
Recalculing the best DEV: WAcc : 1.32%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (167.41878414154053, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 5.333333333333334%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (167.16283893585205, 'sc') | Paramètres:
```

```
Recalculing the best DEV: WAcc : 2.1033333333333335%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (166.51175570487976, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 12.690000000000001%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (167.47926211357117, 'sc') | Paramètres:

Recalculing the best DEV: WAcc : 7.33%
Fin entrainement LSTMHiddenState_64 sur 20 epoch en (167.09464812278748, 'sc') | Paramètres:
```

Pour SGD, les performances sont très faibles quel que soit le learning rate, avec une accuracy maximale de 12.69%.

Conclusion

Les deux optimizers Adam et Adagrad offrent des performances raisonnables, avec des accuracies maximales respectives de 35.31% et 29.91%. Cependant, Adam semble être plus stable et performant dans l'ensemble des tests. En revanche, l'optimizer SGD ne parvient pas à atteindre des performances comparables, avec une accuracy maximale de seulement 12.69%. Pour la suite des optimisations, je vais donc utiliser Adam avec un learning rate de 0.005, un batch size de 500 et un dropout de 0.1.

Vérification des performances finales (vérification de l'impact de l'augmentation du nombre d'epochs)

J'ai relancé l'entraînement du modèle LSTM avec les meilleurs paramètres trouvés (LSTM hidden state avec une couche de 64 neurones, optimizer Adam, learning rate 0.005, dropout 0.1, batch size 500) pour vérifier les performances. J'ai exécuté l'entraînement sur 200 epochs avec earlyStopping à 20.

```
Recalculing the best DEV: WAcc : 36.49333333333333%
Fin entrainement LSTMHiddenState_64 sur 64 epoch en (530.9663951396942, 'sc') | Paramètres:
```

On obtient une accuracy de 36.49% sur le dev set, ce qui confirme que les paramètres choisis sont efficaces pour améliorer les performances du modèle LSTM par rapport à la version de base (22.07%). Le gain est significatif malgré l'augmentation du temps d'entraînement (530sc contre 403sc pour la version de base).

Exécution de game.py

J'ai testé le modèle LSTM optimisé en le faisant jouer des parties contre le modèle MLP optimisé. J'ai lancé 10 parties. Sur les 10, le modèle LSTM a gagné les 10 parties. Il est donc clairement supérieur au modèle MLP optimisé. J'ai répété l'expérience sur 10 parties supplémentaires et le modèle LSTM a encore gagné les 10 parties.

```
Parties:
1-{'save_models_LSTMHiddenState_64model_43.pt': 10}
2-{'save_models_LSTMHiddenState_64model_43.pt': 10}
```

On a donc un taux de victoire de 100% pour le modèle LSTM optimisé contre le modèle MLP optimisé sur les 20 parties jouées.

J'ai ensuite fait jouer le modèle LSTM optimisé contre lui-même, on a un taux de victoire de 50%.

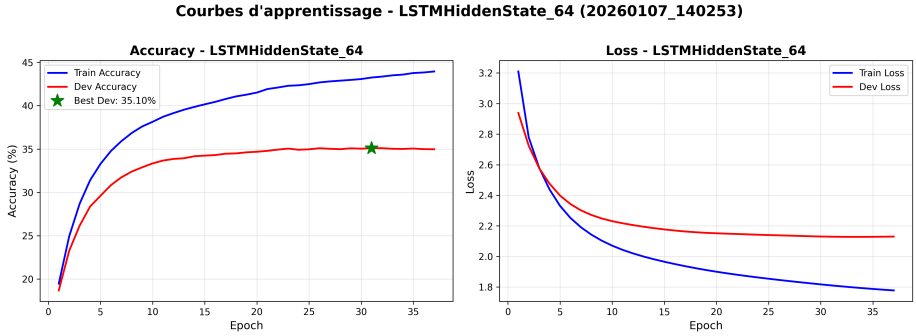
```
Parties:
1-{'save_models_LSTMHiddenState_64model_43.pt': 5, 'save_models_LSTMHiddenState_64model_41.p
2-{'save_models_LSTMHiddenState_64model_43.pt': 5, 'save_models_LSTMHiddenState_64model_41.p
```

Overfitting & Optimisations supplémentaires

Pour ces optimisations, je vais utiliser uniquement l'optimizer Adam car c'est celui qui a donné les meilleurs résultats lors des tests précédents, cela me permettra de réduire le temps d'entraînement.

LSTM optimisé J'ai aussi fait faire des parties entre le modèle LSTM optimisé et le modèle LSTM de base. Le modèle de base a gagné toutes les parties alors qu'il a un score bien moins bon. Je pense qu'il y a de l'overfitting sur le modèle optimisé, car il a été entraîné plus longtemps et a pu mieux mémoriser les parties du dataset d'entraînement.

D'après l'image suivante, on peut voir qu'il y a de l'overfitting car l'accuracy sur le dev set stagne alors que l'accuracy sur le train set continue d'augmenter. C'est pareil pour la loss, on voit que la loss sur le dev set stagne alors que la loss sur le train set continue de diminuer.



Pour améliorer ça, je vais réentrainer le model LSTM optimisé avec plus de dropout (0.1 et 0.2) et moins d'epochs (20).

Pour ce modèle, on a 41536 poids entraînables.

- Dropout = 0.1

```
Recalculing the best DEV: WAcc : 35.42666666666667%
Fin entrainement LSTMHiddenState_Dropout_64 sur 20 epoch en (179.8138403892517, 'sc') | Para
```

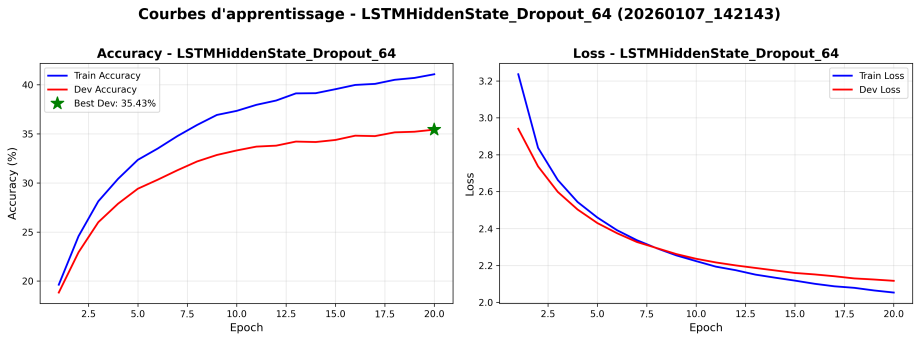


Figure 1: Courbes d'apprentissage LSTM optimisé - Dropout = 0.1

- Dropout = 0.2

```
Recalculing the best DEV: WAcc : 34.026666666666664%
Fin entrainement LSTMHiddenState_Dropout_64 sur 20 epoch en (180.72124028205872, 'sc') | Par
```

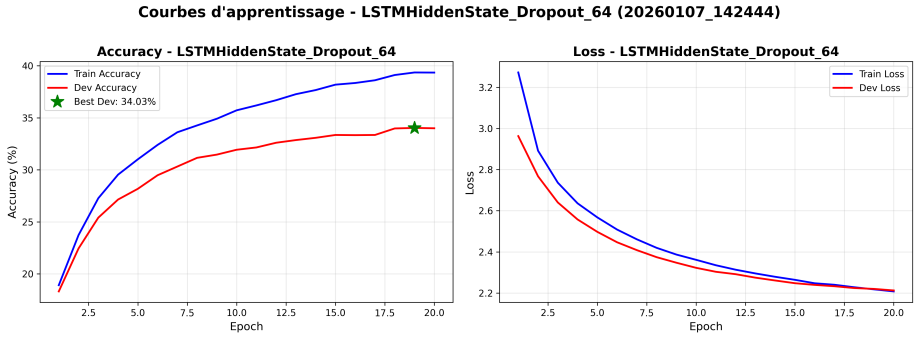


Figure 2: Courbes d'apprentissage LSTM optimisé - Dropout = 0.2

On peut voir que grâce à l'ajout du dropout, la différence de loss entre train et dev s'est réduite ce qui indique une réduction de l'overfitting. En ajoutant une fonction d'activation comme ReLU ou tanh, on pourrait peut-être encore réduire l'overfitting.

- Dropout = 0.1 + ReLU

```
Recalculing the best DEV: WAcc : 31.169999999999998%
Fin entrainement LSTMHiddenState_Dropout_Relu_64 sur 20 epoch en (242.9840669631958, 'sc') |
```

- Dropout = 0.2 + ReLU

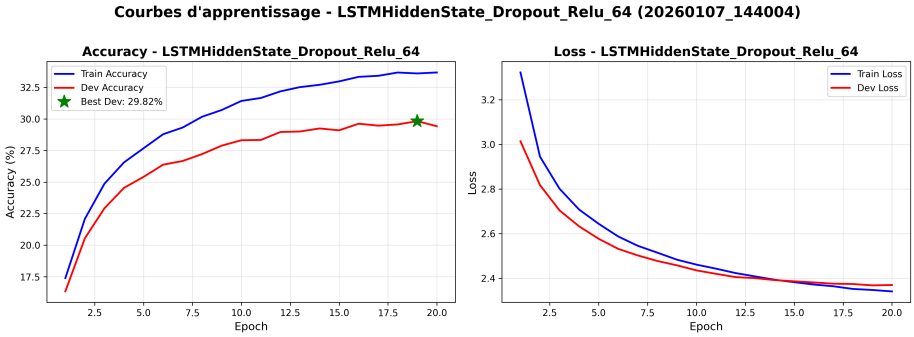


Figure 3: Courbes d'apprentissage LSTM optimisé - Dropout = 0.1 + ReLU

```
Recalculing the best DEV: WAcc : 29.823333333333334%
Fin entrainement LSTMHiddenState_Dropout_Relu_64 sur 20 epoch en (248.544508934021, 'sc') |
```

Courbes d'apprentissage LSTM optimisé - Dropout = 0.2 + ReLU

Figure 4: Courbes d'apprentissage LSTM optimisé - Dropout = 0.2 + ReLU

- Dropout = 0.1 + tanh

```
Recalculing the best DEV: WAcc : 34.94%
Fin entrainement LSTMHiddenState_Dropout_Tanh_64 sur 20 epoch en (246.82054924964905, 'sc')
```

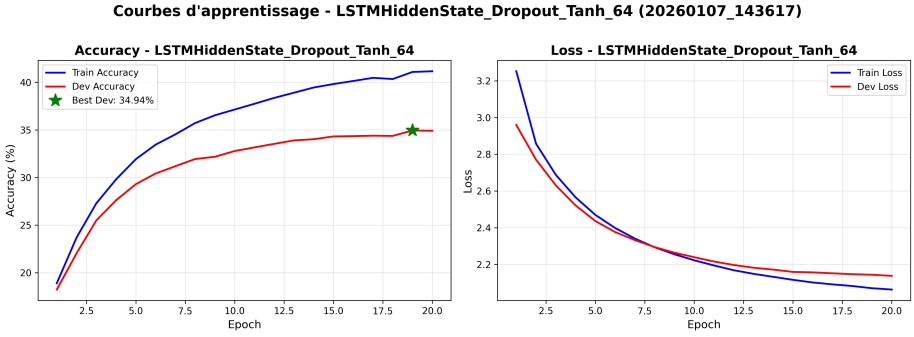


Figure 5: Courbes d'apprentissage LSTM optimisé - Dropout = 0.1 + tanh

- Dropout = 0.2 + tanh

```
Recalculing the best DEV: WAcc : 34.35%
Fin entrainement LSTMHiddenState_Dropout_Tanh_64 sur 20 epoch en (243.60504722595215, 'sc')
```

Plus tôt j'avais retiré la partie "softmax" à la fin du modèle LSTM, je vais la remettre pour voir si ça améliore les performances.

- Dropout = 0.1 + Relu + softmax

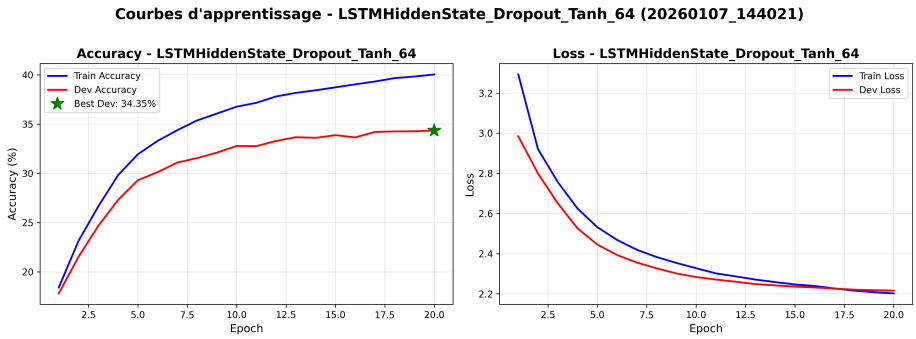


Figure 6: Courbes d'apprentissage LSTM optimisé - Dropout = 0.2 + tanh

```
Recalculing the best DEV: WAcc : 19.666666666666664%
Fin entrainement LSTMHiddenState_Dropout_Relu_Softmax_64 sur 20 epoch en (266.0434787273407
```

- Dropout = 0.2 + Relu + softmax

```
Recalculing the best DEV: WAcc : 20.676666666666666%
Fin entrainement LSTMHiddenState_Dropout_Relu_Softmax_64 sur 20 epoch en (275.09586095809937
```

- Dropout = 0.1 + Tanh + softmax

```
Recalculing the best DEV: WAcc : 22.096666666666668%
Fin entrainement LSTMHiddenState_Dropout_Tanh_Softmax_64 sur 20 epoch en (275.36101055145264
```

- Dropout = 0.2 + Tanh + softmax

```
Recalculing the best DEV: WAcc : 21.51%
Fin entrainement LSTMHiddenState_Dropout_Tanh_Softmax_64 sur 20 epoch en (274, 'sc') | Paran
```

J'avais retiré le softmax pour le LSTMHiddenState. En le rajoutant, on voit que les scores diminuent.

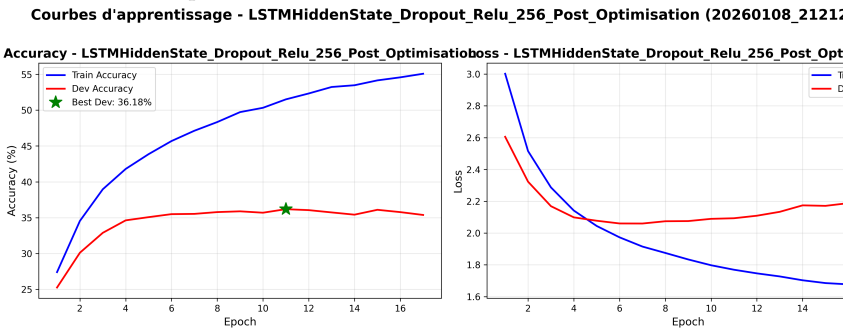
Je vais retenter un entraînement du modèle LSTM optimisé avec un dropout de 0.2 mais sans softmax en testant plusieurs configurations de couches.

Tableau des résultats du grid search sur le LSTM optimisé avec les différentes architectures testées: | Nom du modèle | Optimizer | Dropout | Fonction d'activation | Temps d'entraînement (sc) | Batch size | Best learning Rate | Best Epoch | Accuracy train (%) | Accuracy dev (%) | Loss train | Loss dev | Date | Différence Accuracy train-dev (%) | Différence Loss dev-train (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| LSTMHiddenState_Dropout_Relu_Gridsearch_64 | Adam | 0.2 | Relu | 620.73 | 128 | 0.005 | 20 | 34.83 | 30.8 | 2.3281 | 2.32 | 2026-01-08 10:04:42 | 4.03 | 0.0081 | |
| LSTMHiddenState_Dropout_Relu_Gridsearch_256 | Adam | 0.2 | Relu | 542.9 | 128 | 0.005 | 11 | 54.35 | 36.73 | 1.615 | 2.1172 | 2026-01-08 11:05:16 | 17.62 | 0.5022 | |
| LSTMHiddenState_Dropout_Relu_Gridsearch_512_256 | Adam | 0.3 | Relu |

400.2 | 256 | 0.005 | 14 | 75.46 | 35.29 | 0.9344 | 2.9477 | 2026-01-08 19:45:23 | 40.17 | 2.0133 | | LSTMHiddenState_Dropout_Relu_Gridsearch_512_256_128 | Adam | 0.2 | Relu | 974.52 | 64 | 0.001 | 6 | 49.7 | 28.71 | 1.8241 | 2.6724 | 2026-01-07 23:40:28 | 20.99 | 0.8483 | | LSTMHiddenState_Dropout_Tanh_Gridsearch_64 | Adam | 0.2 | Tanh | 615.17 | 128 | 0.005 | 18 | 39.66 | 33.98 | 2.2305 | 2.221 | 2026-01-08 14:17:49 | 5.68 | 0.0095 | | LSTMHiddenState_Dropout_Tanh_Gridsearch_256 | Adam | 0.3 | Tanh | 1026.71 | 64 | 0.005 | 11 | 52.16 | 37.3 | 1.8576 | 2.1413 | 2026-01-08 05:14:27 | 14.86 | 0.2837 | | LSTMHiddenState_Dropout_Tanh_Gridsearch_512_256 | Adam | 0.3 | Tanh | 716.91 | 64 | 0.001 | 4 | 53.21 | 34.75 | 1.85 | 2.2339 | 2026-01-08 06:41:48 | 18.46 | 0.3839 | | LSTMHiddenState_Dropout_Tanh_Gridsearch_512_256_128 | Adam | 0.3 | Tanh | 888.6 | 64 | 0.001 | 5 | 47.99 | 29.3 | 1.9892 | 2.6123 | 2026-01-08 08:48:17 | 18.69 | 0.6231 |

À mon avis, le modèle LSTM optimisé avec l'architecture [256] et la fonction d'activation ReLU est le meilleur compromis entre performance et temps d'entraînement. Cependant, on observe toujours une différence significative entre l'accuracy sur le train set et le dev set, ce qui suggère qu'il y a encore de l'overfitting. Pour réduire davantage l'overfitting, il serait intéressant d'explorer des techniques supplémentaires telles que la régularisation L2 ou l'augmentation des données.

J'ai tracé la courbe d'apprentissage du modèle LSTM optimisé avec l'architecture [256] et la fonction d'activation ReLU.



On remarque qu'il y a toujours beaucoup d'overfitting. La différence entre l'accuracy sur le train set et le dev set est encore importante, ce qui indique que le modèle a du mal à généraliser aux données non vues. De plus, la loss sur le dev set stagne tandis que la loss sur le train set continue de diminuer, ce qui est un signe classique d'overfitting.

MLP optimisé Comme le LSTM optimisé souffrait d'overfitting, j'ai fait la même analyse sur le MLP optimisé. Le MLP optimisé a l'air d'avoir un comportement plus stable en partie, il ne perd pas contre le modèle LSTM de base comme le LSTM optimisé.

J'ai décidé de faire un grid search sur différentes modèles de MLP pour voir si je peux améliorer ses performances. J'ai décidé, par contrainte de temps, de ne pas

tester le batch size de 64 car ça augmente beaucoup trop le temps d'entraînement (ça a retiré 108 expériences à exécuter).

Tableau des résultats du grid search sur le MLP optimisé avec les différentes architectures testées:

Nom du modèle	Optimizer	Dropout	Fonction d'activation	Temps d'entraînement (sc)	Batch size	Best learning Rate	Best Epoch	Accuracy train (%)	Accuracy dev (%)	Loss train	Loss dev	Date	Différence Accuracy train-dev (%)	Différence Loss dev-train (%)
MLP_64_Dropout_Gridsearch_Relu	Adam	0.2	Relu	922.85	128	0.001	18	24.26	23.53	2.9396	2.722	2026-01-07 21:03:53	0.73	0.2176
MLP_256_Dropout_Gridsearch_Relu	Adam	0.4	Relu	525.66	128	0.001	20	38.39	35.22	2.203	1.9754	2026-01-07 23:45:59	3.17	0.2276
MLP_256_128_Dropout_Gridsearch_Relu	Adam	0.4	Relu	718.36	128	0.001	20	34.13	30.67	2.5184	2.2755	2026-01-08 01:06:41	3.46	0.2429
MLP_512_256_Dropout_Gridsearch_Relu	Adam	0.2	Relu	709.16	128	0.001	20	43.85	35.84	2.1466	1.9854	2026-01-08 02:34:16	8.01	0.1612
MLP_512_256_128_Dropout_Gridsearch_Relu	Adam	0.4	Relu	899.94	128	0.001	20	38.09	31.75	2.4348	2.3264	2026-01-08 04:05:46	6.34	0.1084
MLP_512_256_128_64_Dropout_Gridsearch_Relu	Adam	0.4	Relu	1082.85	128	0.001	19	32.17	27.63	2.6893	2.5827	2026-01-08 05:51:02	4.54	0.1066
MLP_64_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	528.29	128	0.005	19	23.19	22.42	2.9964	2.8018	2026-01-08 08:06:02	0.77	0.1946
MLP_256_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	523.34	128	0.001	20	34.49	31.71	2.5426	2.2591	2026-01-08 09:15:09	2.78	0.2835
MLP_256_128_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	698.05	128	0.001	20	29.42	26.99	2.7871	2.5296	2026-01-08 10:35:42	2.43	0.2575
MLP_512_256_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	702.4	128	0.001	20	36.91	32.31	2.4729	2.201	2026-01-08 12:16:11	4.6	0.2719
MLP_512_256_128_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	875.99	128	0.001	20	31.94	28.04	2.6999	2.4422	2026-01-08 14:04:29	3.9	0.2577
MLP_512_256_128_64_Dropout_Gridsearch_Tanh	Adam	0.4	Tanh	1056.46	128	0.001	20	25.4	22.8	2.9199	2.7366	2026-01-08 16:01:31	2.6	0.1833

Selon moi, le meilleur modèle est le MLP_512_256_Dropout_Gridsearch_Relu car il a la meilleure accuracy sur le dev set (35.84%) et une différence accuracy train-dev de 8.01% ce qui est acceptable.

On remarque que le MLP optimisé avec l'architecture [512, 256] et la fonction d'activation ReLU a un comportement plus stable que le LSTM optimisé. La différence entre l'accuracy sur le train set et le dev set est moindre, ce qui indique que le modèle généralise mieux aux données non vues. Cependant, il y a encore une certaine différence, suggérant qu'il pourrait y avoir un léger overfitting.

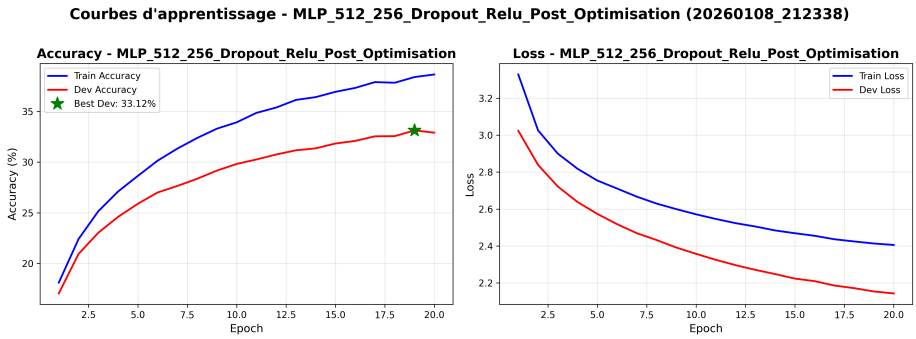


Figure 7: Courbes d'apprentissage MLP optimisé - Architecture [512, 256] + ReLU

Test des modèles optimisés en partie Les modèles ont toujours des performances en parties qui sont pas en accord avec leurs performances sur le dev set. Je pense que malgré le fait que j'ai réduit l'overfitting, il en reste encore un peu. Par exemple, le modèle LSTM optimisé et le modèle MLP optimisé ont des accuracies respectives de 36.73% et 35.84% sur le dev set, mais lors des parties jouées, ils ont gagné 50/10 parties chacun à chaque fois.

Je vais donc passer au CNN pour voir si je peux obtenir de meilleures performances et je ferais l'augmentation des données après ça.

Conclusion de l'optimisation

Après avoir optimisé les modèles MLP et LSTM, j'ai constaté que le modèle LSTM optimisé avec l'architecture [256] et la fonction d'activation ReLU offre les meilleures performances, avec une accuracy de 36.73% sur le dev set. Cependant, il souffre encore d'overfitting, comme en témoigne la différence significative entre l'accuracy sur le train set et le dev set. Le modèle MLP optimisé avec l'architecture [512, 256] et la fonction d'activation ReLU présente également de bonnes performances, avec une accuracy de 35.84% sur le dev set, et un comportement plus stable en termes de généralisation. Cependant, les performances en parties jouées ne sont pas entièrement cohérentes avec les performances sur le dev set.

CNN TODO

CNN-LSTM TODO

Transformer TODO

Augmentation des données TODO

Transformations

Pour augmenter les données, j'ai modifié les classes de Dataset dans data.py pour faire des opérations simples sur les parties à leur chargement: - Rotations (90, 180, 270 degrés) - Symétrie Le dataset a été démultiplié grâce à ces transformations (il y a actuellement 1440240 samples dans le train set contre 180030 avant augmentation).

J'ai donc réentraîné les modèles optimisés LSTM et MLP avec ce nouveau dataset augmenté.

LSTM optimisé avec données augmentées

```
Recalculing the best DEV: WAcc : 45.63458333333333%
Fin entrainement LSTMHiddenState_Dropout_Relu_256_Post_Optimisation_DataAugmentation_20epoch
```

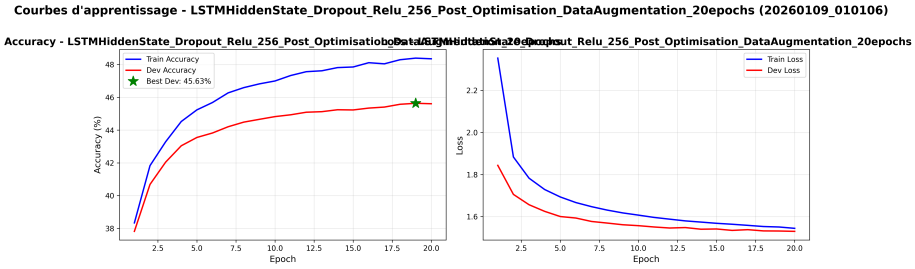


Figure 8: Courbes d'apprentissage LSTM optimisé avec données augmentées

En analysant la courbe d'apprentissage du LSTM optimisé avec les données augmentées, on peut observer que l'accuracy sur le dev set a considérablement augmenté pour atteindre 45.63%, ce qui est une amélioration significative par rapport à l'entraînement précédent sans augmentation des données (36.73%). De plus, l'overfitting semble avoir été réduit, car la différence entre l'accuracy sur le train set et le dev set est moindre (~6%). Cependant, le temps d'entraînement a également augmenté de manière significative (1677sc contre 542sc avant augmentation des données).

On peut remarquer que la loss sur dev et sur train sont très proches (1.3% d'écart), ce qui indique que le modèle généralise bien.

Vu le gain en performance général, cela semble être un compromis acceptable. Je vais sauvegarder ce modèle et le faire jouer des parties contre le MLP optimisé avec données augmentées (Nom du modèle: **save_models_LSTMHiddenState_Dropout_Relu_256_Post_Optimisation_DataAugmentation_**

MLP optimisé avec données augmentées

```
Recalculing the best DEV: WAcc : 38.403333333333336%
Fin entrainement MLP_512_256_Dropout_Relu_Post_Optimisation sur 20 epoch en (3188.796848535
```

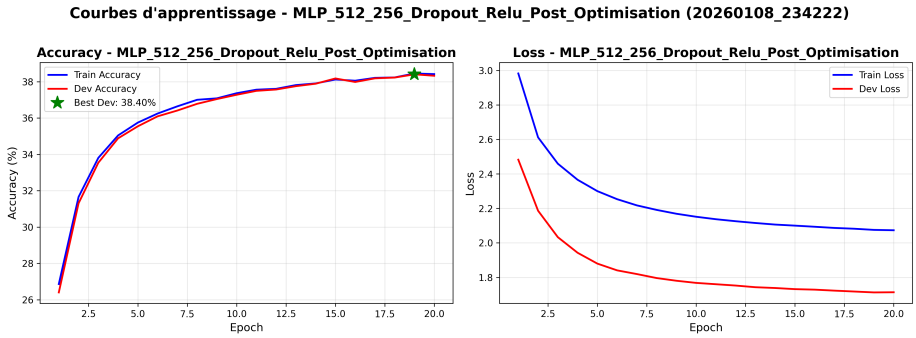


Figure 9: Courbes d'apprentissage LSTM optimisé avec données augmentées

Grâce à la courbe d'apprentissage, on peut remarquer que l'overfitting est quasiment nul sur cet entrainement avec un très bon score de 38% par rapport à avant. Cependant, le temps d'entraînement a énormément augmenté (3188sc contre 709sc avant augmentation des données). Mais vu le gain en précision, c'est plutôt correct.

Comme j'ai remarqué que la courbe n'avait pas terminé de monter, j'ai décidé de lancer l'entrainement avec plus d'epochs (50):

```
Recalculing the best DEV: WAcc : 46.385%
Fin entrainement MLP_512_256_Dropout_Relu_Post_Optimisation_DataAugmentation_50epochs sur 5
```

Dans ce graphique, on voit que l'accuracy sur le dev set a continué d'augmenter pour atteindre 46.38% après 50 epochs, ce qui est une amélioration significative par rapport aux entraînements précédents. De plus, l'overfitting reste très faible, indiquant que le modèle généralise plutôt bien. Lors ce cet entrainement, le temps d'entraînement a encore augmenté (4120sc contre 3188sc pour 20 epochs), mais le gain en performance semble justifier ce coût supplémentaire. De plus, on peut remarquer que l'overfitting a fortement diminué par rapport à l'entrainement avec 20 epochs (9% d'écart entre train et dev contre 21% avant). Je vais donc garder ce modèle MLP optimisé avec données augmentées (Nom du modèle: **save_models_MLP_512_256_Dropout_Relu_Post_Optimisation_DataAugmentation_50epoch** pour le faire jouer des parties contre le LSTM optimisé avec données augmentées.

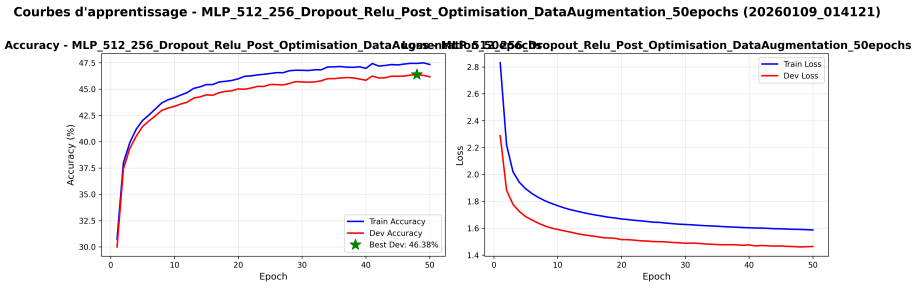


Figure 10: Courbes d'apprentissage LSTM optimisé avec données augmentées - 50 epochs

Génération de parties TODO

Pour augmenter les données disponibles, je vais passer par l'implémentation de la confrontation entre deux modèles. En faisant jouer deux modèles l'un contre l'autre, on peut générer de nouvelles parties qui pourront être utilisées pour entraîner les modèles.

Utilisation de toutes les données et non pas seulement les parties gagnées TODO

TODO

CNN optimisé vs LSTM optimisé TODO TODO TODO

Entrainement final TODO TODO TODO

Pour l'entrainement final, j'ai entrainé le modèle TODO optimisé avec données augmentées sur 100 epochs pour voir si je peux encore améliorer les performances. J'ai regroupé les données de test/dev dans le train set augmenté pour avoir plus de données d'entrainement.

`TODO`

League Othello TODO - à héberger en local

J'ai hébergé la League Othello en local pour faire s'affronter les différents modèles optimisés entre eux.

Fais des tableaux de scores pour chaque modèle avec le nombre de victoires et le score moyen.

Voici les résultats des différents affrontements:

1er affrontement (Uniquement avec MLP et LSTM optimisés avec données augmentées):

Rang	Modèle	Victoires	Score Moyen
1	MLP_optimise_model_48_1767919116.7898524_0.463855prct_jit.pt	18	13
2	Othello_Overlord.pt	16	9
3	Stable_Genius.pt	16	8
4	Convolution_Conqueror.pt	14	5
5	Edge_Hog.pt	14	5
6	Chaotic_Neutral.pt	14	-2
7	Multilayer_Menace.pt	13	2
8	RL_Qlearner_the_Confused.pkl	12	-2
9	Midgame_Magician.pt	11	1
10	The_CoinFlipper.pt	11	-2
11	Random	10	-5
12	The_Tile_Donor.pt	5	-13
13	LSTM_optimise_model_19_1767916779.1295853_0.456345833333333334prct_torchscript.pt	3	-13

On peut remarquer que le modèle MLP optimisé avec données augmentées est très performant dans cette league, si bien qu'il termine premier du classement. Les très mauvaises performances du modèle LSTM s'expliquent par le fait que, lors de l'export, une conversion de l'entrée doit être faite (le modèle attend quelque chose de taille 64 et on lui donne une matrice 8x8), donc cette conversion fait perdre les poids et rend le modèle inefficace. Par contrainte de temps je n'ai pas pu corriger ce problème, donc je vais exclure le modèle LSTM des prochains affrontements.

Game.py

En faisant jouer plusieurs modèles les uns contre les autres, j'ai remarqué que les parties effectuées étaient toujours strictement les mêmes pour deux modèles données. J'ai donc modifié un peu la fonction `find_best_move` pour ajouter un peu d'aléatoire dans le choix des coups à jouer, comme dans une partie réelle.

Utilisation GPU/CPU

Lors de l'entrainement, mon GPU est faiblement utilisé (7%). Le CPU lui a des piques à 100% mais en moyenne il est à 15%. Je pense qu'il y a un problème de lenteur au chargement des données. Quand j'augmente le batch size, le temps d'entrainement diminue mais la précision du modèle est impactée négativement.

Je pense que le goulot d'étranglement vient du CPU qui n'arrive pas à fournir les données assez rapidement au GPU. En passant de LSTM/MLP à CNN, l'utilisation du GPU a augmenté à 50%, donc les modèles LSTM et MLP sont probablement trop simples pour exploiter pleinement la puissance du GPU.

Conclusion générale TODO

Optimisation

Modèles

Fonctions d'activation

Optimizers

Augmentation des données