Rapport Projet Deep Learning :
Train a network for playing the game of Go

Maxence Philbert, Vincent Gouteux

Décembre 2019

Contents

1 Présentation du Problème et des données **1**

2 Choix du modèle : Architecture **2**
2.1 Réseaux Convolutifs 2
2.2 Réseaux Résiduels 2
2.3 Réseaux Résiduel pour le Go 3
2.3.1 Premier modèle 4
2.3.2 Second modèle 4

3 Choix du modèle : Paramètres **4**
3.1 Optimizer 5
3.1.1 Adam 5
3.1.2 RMSprop 6
3.1.3 SGD 6

4 Entraînement du modèle **7**
4.1 Batch Dynamique 7
4.2 Entraînement 7

1 Présentation du Problème et des données

Programmer un joueur de go est considéré comme un problème bien plus difficile que pour d'autres jeux, comme les échecs, en raison d'un bien plus grand nombre de combinaisons possibles, ce qui rend extrêmement complexe l'utilisation de méthodes traditionnelles telles que la recherche exhaustive. Pour apprendre à une machine a jouer à ce jeu il est donc extrêmement coûteux de calculer tout le coups possibles et de sélectionner celui qui est jugé comme le meilleur. C'est pourquoi ce jeu se prête très bien aux réseaux de *deep learning*.

Le but de ce projet est d'implémenter un réseau de neurone qui apprend à jouer au jeu de Go. La principale difficulté est que le nombre de paramètres autorisés est limité a 1 milion. L'objectif est donc de trouver l'architecture et les paramètres qui nous permettra de d'apprendre au mieux les données avec un nombre de paramètres limités.

Les données utilisées proviennent des parties simulées par le programme ELF opengo Go de Facebook. Il y a plus de 98 000 000 états différents au total dans le jeu d'entrainement. Les données d'entrée sont composées de 8 plans 19x19 (couleur à jouer, échelles, état actuel sur deux plans, deux états précédents sur quatre plans). Les cibles de sortie sont la politique (un vecteur de taille 361 avec 1,0 pour le coup joué, 0,0 pour les autres coups), la valeur (1,0 si les blancs gagnent, 0,0 si les noirs gagnent) et l'état à la fin du jeu (deux plans).

2 Choix du modèle : Architecture

Dans cette partie nous avons gardé le même échantillon d'entraînement pour effectuer tout nos tests et déterminer l'architecture qui rendait notre réseau le plus performant possible tout en respectant la contrainte de 1 million de paramètres imposée.

2.1 Réseaux Convolutifs

Dans l'analyse d'images comme pour les jeux les réseaux convolutifs sont très utilisés. Ils sont notamment très performants pour reconnaître les formes et s'avèrent efficaces pour l'apprentissage du jeu de Go. Nous avons donc dans un premier temps implémenté un simple réseau composé de couches de convolutions. Nous avons remarqué suite à nos premiers tests que ce réseau était déjà assez performant et apprenait bien les données.

Layer (type)	Output Shape	Param #	Connected to
board (InputLayer)	[(None, 19, 19, 8)]	0	
conv2d (Conv2D)	(None, 19, 19, 32)	2336	board[0][0]
conv2d_1 (Conv2D)	(None, 19, 19, 32)	9248	conv2d[0][0]
conv2d_2 (Conv2D)	(None, 19, 19, 32)	9248	conv2d_1[0][0]
conv2d_3 (Conv2D)	(None, 19, 19, 32)	9248	conv2d_2[0][0]
conv2d_4 (Conv2D)	(None, 19, 19, 32)	9248	conv2d_3[0][0]
conv2d_5 (Conv2D)	(None, 19, 19, 32)	9248	conv2d_4[0][0]
conv2d_6 (Conv2D)	(None, 19, 19, 1)	289	conv2d_5[0][0]
flatten (Flatten)	(None, 361)	0	conv2d_6[0][0]
flatten_1 (Flatten)	(None, 11552)	0	conv2d_5[0][0]
policy (Dense)	(None, 361)	130682	flatten[0][0]
value (Dense)	(None, 1)	11553	flatten_1[0][0]

Figure 1: Architecture de réseau convolutif

Le réseau implémenté était très similaire au réseau *test.h5* donné pour exemple. Nous avons simplement ajouté des couches et augmenté le nombre de paramètres pour améliorer les performances du réseau. Les résultats sont intéressants mais le réseau semble se stabiliser vers une *validation loss* aux alentours de 3.5. Ces résultats signifient que le réseau apprend mais elle est encore bien trop élevée pour considérer que le réseau pouvait jouer de façon autonome. Nous avons donc choisi de garder cette architecture de réseaux convolutifs mais en ajoutant quelques modifications pour améliorer les performances.

2.2 Réseaux Résiduels

Les réseaux résiduels se basent sur le principe de réutiliser des données des couches précedentes. Une des raisons pour sauter les couches est d'éviter le problème de la *disparition du gradient*, en réutilisant les poids d'une couche précédente. Nous avons donc utilisé la même architecture que pour le réseau précédent mais en réintroduisant a chaque étape les poids de la couche précédente

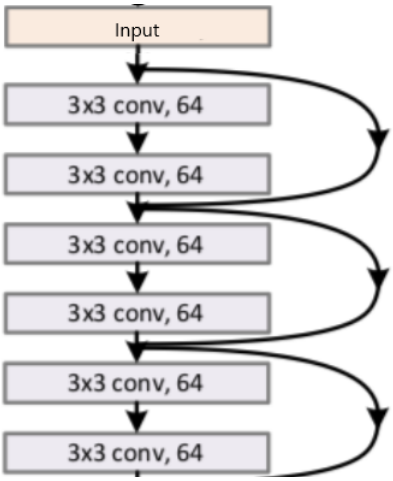


Figure 2: Architecture de réseau convolutif résiduel

Le schéma ci dessus provient de l'article suivant the-3-tricks-that-made-alphago-zero-work dont nous nous sommes inspirés pour concevoir le réseau Nous observons des résultats meilleurs que pour le réseau convolutif simple mais la valeur de la loss reste élevée et le réseaux peut encore être largement amélioré.

2.3 Réseaux Résiduel pour le Go

L'architecture finale du réseau que nous avons implémenté est une architecture qui est très similaire à celle disponible dans ce papier de recherche : Residual Networks for Computer Go dont le schéma ci dessous provient également

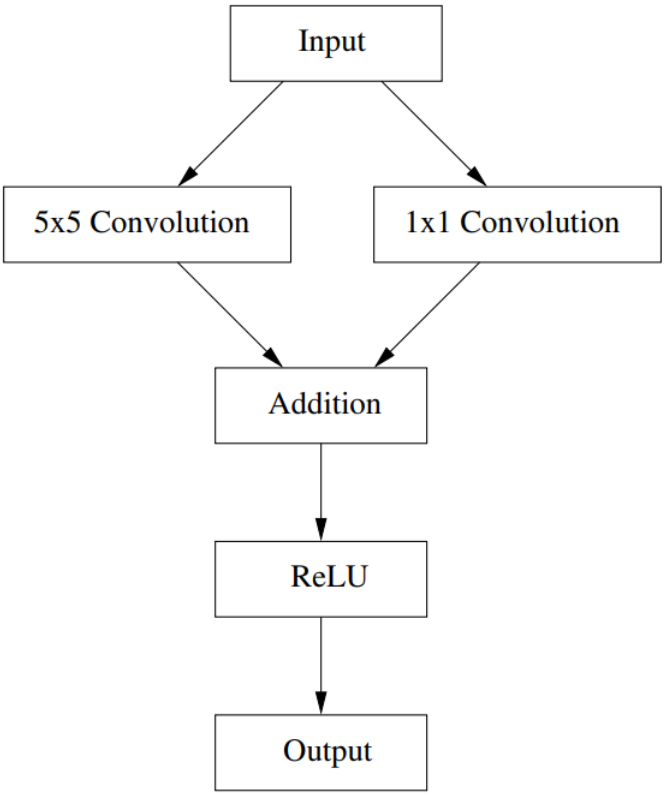


Figure 3: Architecture de réseau convolutif résiduel pour le Go

2.3.1 Premier modèle

Nous avons d'abord implémenté un premier modèle ou chaque *bloc* était de la forme suivante

```
y=x
x = layers.Conv2D(64, 5, padding='same')(x)
x = layers.Activation('relu')(x)
x = layers.Conv2D(64, 5, padding='same')(x)
y = layers.Conv2D(64, 1, padding='same')(y)
x = layers.add([x,y])
x = layers.Activation('relu')(x)
```

Figure 4: Architecture du premier réseau

Ce modèle etait performant mais se stabilisait a une loss autour de 3 ce qui est satisfaisant mais peut encore être amélioré

2.3.2 Second modèle

Nous avons ensuite implémenté un second modèle inspiré du papier de recherche ci dessus ou chaque *bloc* était de la forme suivante

```
y=x
x = layers.Conv2D(64, 5, activation='relu', padding='same')(x)
y = layers.Conv2D(64, 1, activation='relu', padding='same')(y)
x = layers.add([x,y])
```

Figure 5: Architecture du second réseau

Ce réseau bien qui soit plus simple que le premier est plus efficace et semble apprendre mieux. C'est donc celui ci que nous avons décidé de conserver pour effectuer l'entraînement final.

3 Choix du modèle : Paramètres

Nous avons remarqués qu'a partir de 12 epochs le modèle commençait a sur-apprendre les données d'entraînement. En effet la loss pour le jeu d'entraînement semble se stabiliser voir augmenter a partir d'un dizaine d'epochs et ce pour tout les hyperparametres. Nous allons donc effectuer nos tests sur 12 epochs seulement

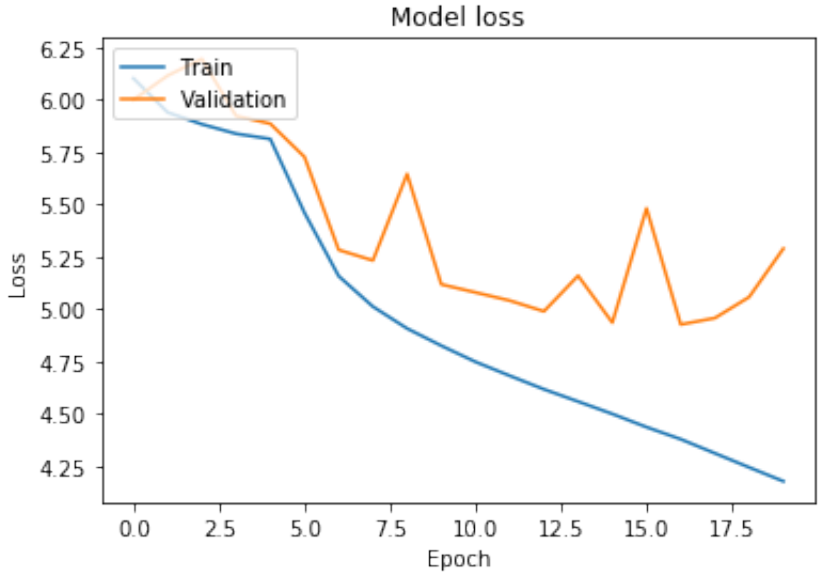


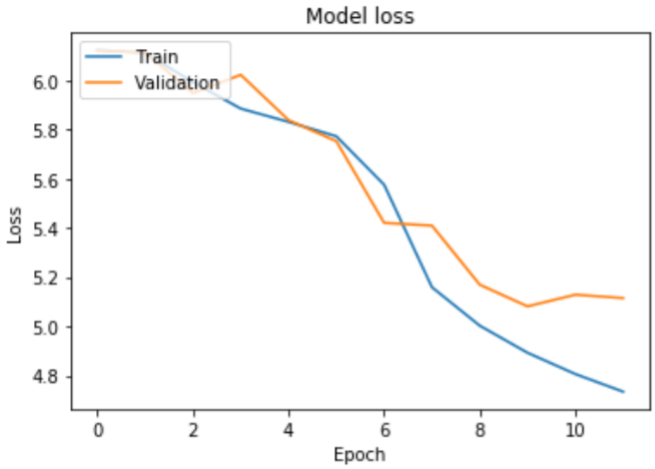
Figure 6: Overfitting au bout de 10 epochs

3.1 Optimizer

Nous allons commencer par tester les performances du réseau avec différents *optimizers* afin de sélectionner celui qui a les meilleurs résultats et de passer a l'entraînement du réseau. Pour qu les comparaisons soient cohérentes nous n'allons pas utiliser le *dynamicBatch* afin de tester les performances des réseaux sur les mêmes données. Nous allons donc effectuer les tests sur le réseau vu dans la section précédente sur 12 epochs.

3.1.1 Adam

Le premier *optimizer* que nous allons tester est le *Adam*

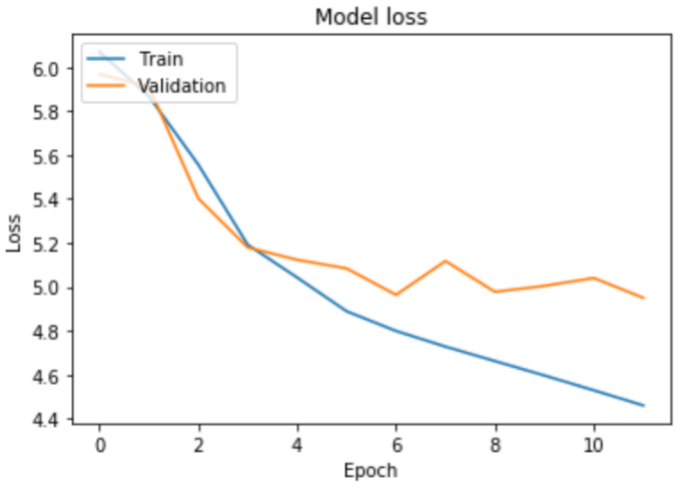



```
Train loss : 4.733926066419813 Validation Loss : 5.11450405883789
```

Figure 7: Training et Validation Loss pour Adam

Pour ce premier *optimizer* comme pour les autres nous allons nous contenter des paramètres par défauts qui souvent sont les paramètres optimaux.

3.1.2 RMSprop



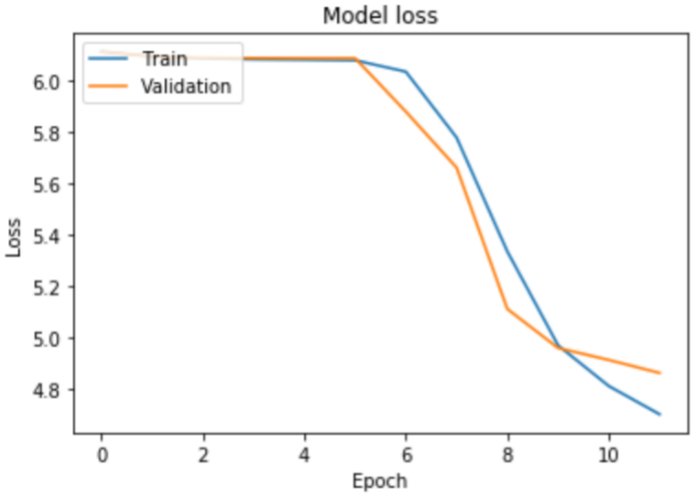

```
Train loss : 4.459726598442925 Validation Loss : 4.949700023651123
```

Figure 8: Training et Validation Loss pour RMSprop

Pour cet *optimizer* nous avons également pris les paramètres par défaut a savoir *learning rate* = 0.001 rho =0.9

3.1.3 SGD

Le dernier *optimizer* que nous allons tester est le *stochastic gradient descent*




```
Train loss : 4.698284914737277 Validation Loss : 4.859768920898437
```

Figure 9: Training et Validation Loss pour SGD

Après plusieurs test il semble que le *learning rate* optimal soit de 0.1, pour des valeurs supérieurs la descente de gradient fait des pas trop grands et a du mal à faire baisser la fonction de perte

Bien que le modèle ne soit pas très lourd il est assez coûteux à entraîner c'est pourquoi tester les differents *optimizers* peut prendre un peu de temps. Si de plus nous devons faire des grid searchs ou des tests pour tester l'erreur du modèle pour différents paramètres le temps de calcul devient exponentiel c'est pourquoi il est très compliqué de trouver les paramètres optimaux au problème et que nous contenterons des paramètres par défauts qui donne des résultats.

4 Entraînement du modèle

Au vu des résultats obtenus, bien que les résultats semblent assez similaires il semblerait que le *optimizer* SGD présente le meilleur résultat. Nous allons donc conserver ce dernier pour l'entraînement du réseau.

Les différentes configurations de parties sont extrêmement nombreuses c'est pourquoi il faut entraîner le réseau sur énormément d'exemples pour apprendre au mieux. Nous allons utiliser la fonction *dynamicBatch* qui nous permet de sélectionner aléatoirement 100 000 exemples sur les 109M disponibles nous allons donc conserver le réseau mais boucler sur les batchs pour entraîner sur le plus d'exemples possibles. Afin de ne pas sur-apprendre nous allons limiter les epochs a 5 par batch.

4.1 Batch Dynamique

Cette fonction est l'élément clé de notre entraînement. En effet pour comparer les réseaux entre eux il suffisait de les tester sur un jeu d'entraînement quelconque. Mais les possibilités du jeu de go sont tellement nombreuses que le potentiel jeu d'entraînement est énorme. Nous ne pouvons pas entraîner un réseau de cette taille avec de très gros jeux d'entraînements, il est donc nécessaire de sélectionner aléatoirement des *batchs* parmi toutes les parties disponibles pour entraîner le réseau.

La fonction *dynamicBatch* implémentée en C++ permet de choisir aléatoirement N états dans l'ensemble de 109 millions d'états disponibles et donc pour avoir le réseau le plus performante possible nous allons énormément utiliser cette fonction.

4.2 Entraînement

Nous avons remarqué qu'il était plus efficace d'entraîner le réseau en *2 étapes* (voire meme 3). La première étape d'entraînement consistait à entraîner le réseau sur plusieurs epochs pour que les réseau apprenne *grossièrement*. En effet sur les premières phases d'entraînement les poids étant initialisés aléatoirement nous pouvons entraîner sur plusieurs epochs sans pour autant faire du sur-apprentissage. Nous avons donc entraîné le réseau successivement sur 8 epochs pour 10 batchs différents de tailles 100 000.

Nous remarquons que quand la loss atteint une valeur intéressante (aux alentours de 3) nous avons des phénomènes d'overfittings très nets. On remarque à l'aide de ce graphique que le réseau met quelques temps à commencer à apprendre ce qui est sûrement du au fait que les poids sont initialisés aléatoirement. Le réseau apprend ensuite très bien sur plusieurs batchs avant de se stabiliser et commencer a sur-apprendre. Nous stoppons donc la première phase d'apprentissage ici, lorsque la loss semble se stabiliser atour de 3 et que pour chaque nouveau batch nous commençons a overfitter dès la seconde epoch.
Nous voyons que pour les 2 premiers batchs la loss ne descend pas beaucoup, puis sur les deux suivants le réseau semble apprendre énormément. Ensuite on observe la courbe bleu en *dent de scie* ce qui nous permet de très bien voir limpact du changement de batch. A partir de 6 batchs les loss ne semblent plus trop diminuer et pour chaque batch quand la training loss baisse la validationloss augmente donc nous sommes entrain d'overfitter le model.

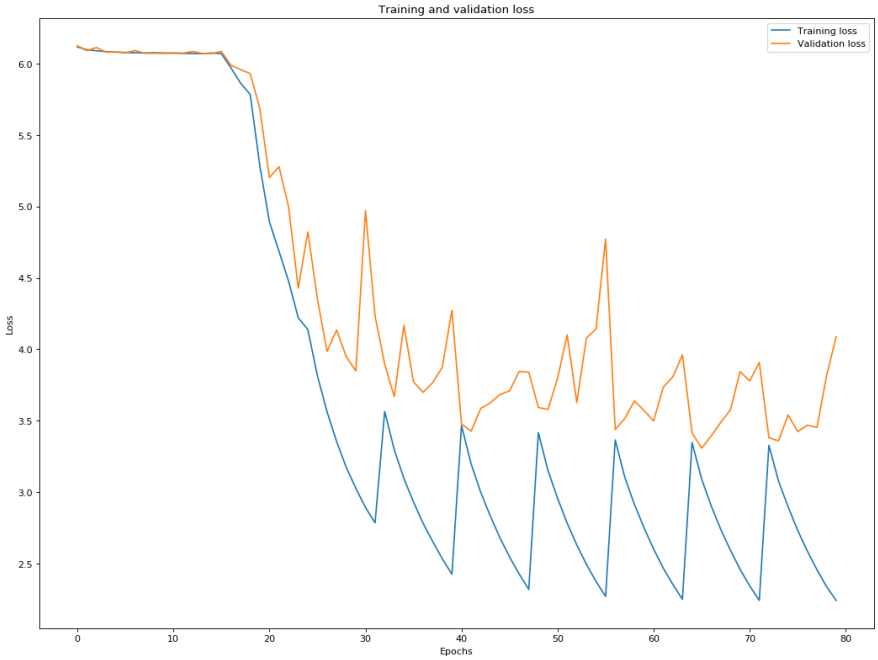


Figure 10: Training et Validation Loss (10 batchs 8 epoch par batch)

Nous allons donc commencer une seconde phase d'entraînement ou nous allons entraîner le réseau seulement sur 1 epoch et changer de batch a chaque fois. Ceci permet de ne pas sur-apprendre les données et de couvrir un maximum d'exemples possibles. On observe que la loss diminue encore légèrement et semble se stabiliser vers 2.7. On remarque a l'aide du graphique que cet entraînement est très performant et fait bien baisser la loss. nous avons poussé l'entraînement le plus possible et au bout d'un nombre élevé de batch nous atteignons une loss qui semble être la plus faible pour ce réseau.

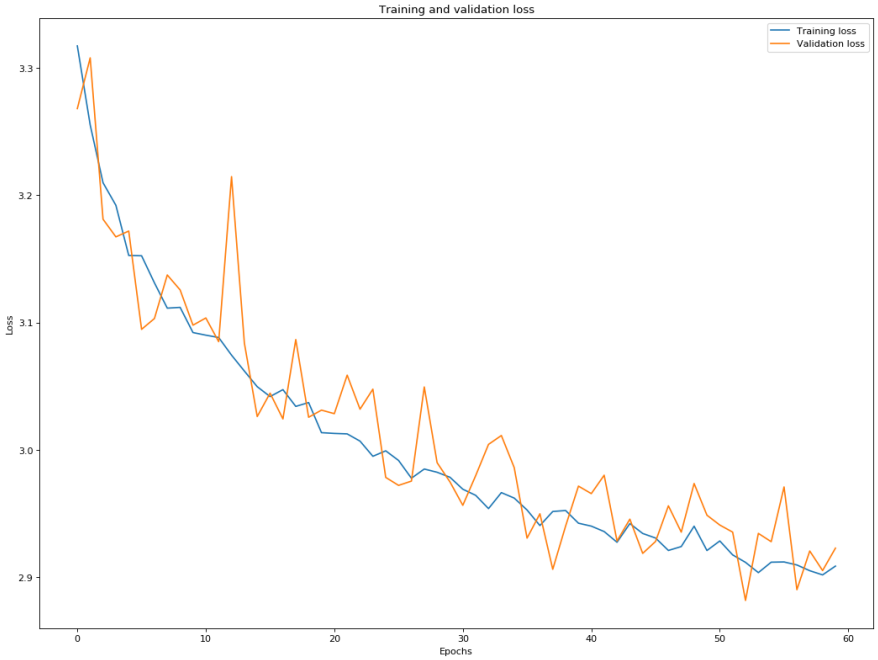


Figure 11: Training et Validation Loss (60 batchs 1 epoch par batch)

Finalement nous avons remarqué qu'en procédant a une dernière phase d'entraînement nous pouvions encore augmenter les performance du model, en faisait encore légèrement baisser la validation

loss. Cette dernière phase consiste à ré entraîner le réseau cette fois ci sur 3-4 epochs mais avec un learning rate bien plus faible. Nous avons choisi 0.002 qui nous permettait de continuer a apprendre mais était assez faible pour ne pas overfitter. Après plusieurs évaluations du modèle la loss finale que nous obtenons est très légèrement inférieure a 2.6, un résultat satisfaisant mais qui pourrait encore être amélioré avec un entraînement plus conséquent, la différence serait assez négligeable car il semblerai que nous avons atteint les *limites* pour ce type de modèle.

Conclusion

Le code a été nettoyé et nous n'avons gardé que l'essentiel. La plus grosse partie du travail a été de se documenter et de tester les différentes architectures possibles ainsi que les hyperparametres. Une fois trouvé l'architecture que nous considérons comme optimale (bien que le réseau puisse être encore amélioré) nous avons procédé à l'entraînement du réseau. La partie entraînement était bien évidemment plus simple et moins intéressante mais nous avons quand même du jouer sur les epochs et les jeux d'entraînement pour optimiser d'entraînement. Cette partie nous a notamment appris à entraîner efficacement un réseau. Le nombre de paramètres entraînables étant limité les résultats le sont aussi. D'après les nombreux articles et papiers de recherches que nous avons lus il semble néanmoins que cette architecture de réseau convolutif résiduel semble être celle qui se prête le mieux à l'apprentissage du jeu de Go.