IASD Deep learning

Corentin Guérendel & Vincent Henric

08/01/2020

1 Architecture du modèle

La philosophie de l'architecture de notre modèle a été de copier celle utilisée par AlphaGo Zero [1] en le restreignant au nombre de paramètres requis (1 000 000). Tout d'abord, notre réseau se compose d'un "bloc convolutionnel" composé de trois couches convolutionnelles :

- avec respectivement : 38, 64 et 128 filtres
- de noyau (3,3) à chaque fois
- d'une Batch normalisation
- d'une fonction d'activation ReLu.

A la fin de ce bloc, un dropout à 0.2 est réalisé. Nous avons ensuite deux "blocs résiduels" composés chacun de :

- une couche convolutionnelle de noyau (3,3) avec 128 filtres
- d'une Batch normalisation
- d'une fonction d'activation ReLu
- une autre couche convolutionnelle de noyau (3,3) avec 128 filtres
- d'une Batch normalisation
- d'une fonction d'activation ReLu.

Entre les blocs résiduels ainsi qu'après le deuxième bloc résiduel, un dropout à 0.4 est réalisé afin de prévénir l'overfitting. L'output de ces deux blocs résiduels est ensuite séparé en deux "têtes" afin d'approximer la policy et la value. Concernant la tête de la policy, nous avons :

- une couche convolutionnelle de noyau (1,1) avec 2 filtres
- d'une Batch normalisation
- d'une fonction d'activation ReLu
- une couche fully connected (361 neurones) avec une fonction d'activation sofmax.

La tête de la value est composée de :

- une couche convolutionnelle de noyau (3,3) avec 32 filtres
- d'une fonction d'activation ReLu
- une couche convolutionnelle de noyau (1,1) avec 1 filtre

- d'une fonction d'activation ReLu
- deux couches d'average spatial pooling (padding = (1,1) et stride = 2)
- une couche convolutionnelle de noyau (1,1) avec 1 filtre
- une couche fully connected (avec 256 neurones)
- d'une fonction d'activation ReLu
- une couche fully connected (avec 256 neurones)
- une dernière couche avec neurone et une fonction d'activation tanh.

Le Spatial Average Pooling, comme montré dans cet article [2], est assez intéressant afin d'évaluer au mieux la probabilité de gagner.

Nous avons également rajouté une contrainte sur les noyaux des couches de convolution, à savoir nous avons imposés des symmétries pour cette matrice. Les symétries utilisées sont les mêmes que celle du plateau du jeu de Go, à savoir:

- symétries axiales selon les deux axes
- symétrie selon les deux diagonales

Ainsi, pour un noyau de dimension $(3, 3)$, il est de la forme:

$$\begin{pmatrix} a & b & a \\ b & c & b \\ a & b & a \end{pmatrix}$$

Cette idée a eu un énorme impact sur les performances.

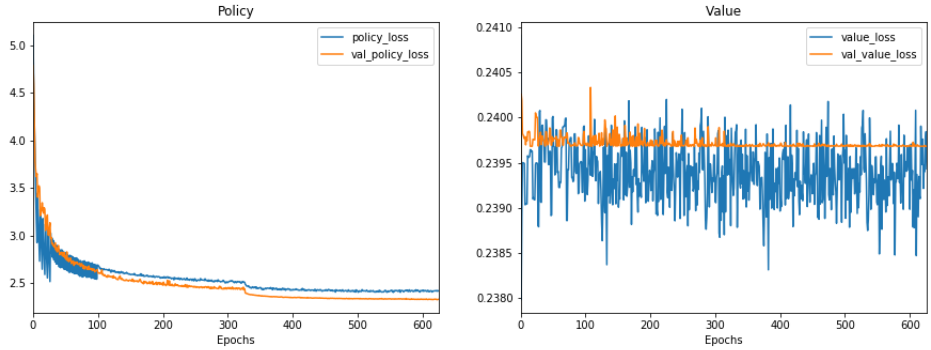
2 Démarche

Nous avons d'abord essayé d'obtenir une architecture de réseau prometteuse. Pour chaque réseau essayé, nous avons lancé l'apprentissage sur 20 epochs. Nous avons enregistré et ensuite comparé les résultats sur le jeu de données de validation, et avons passé plus de temps sur les architectures les plus prometteuses.

En ce qui concerne l'optimisation de l'apprentissage, nous avons bien sûr constaté qu'utiliser des batchs dynamiques permettaient d'augmenter nettement les performances. En utilisant uniquement les données du fichier .npy, le réseau surapprend rapidement. Nous avons mis en place une procédure nous permettant de choisir simplement la fréquence de rechargement d'un batch dans le cadre des batch dynamiques. Pour le réseau final, nous avons appelé la fonction getBatch toutes les 4 epochs pour les 25 premières epochs, puis toutes les 2 epochs jusqu'à la 100ème, puis toutes les epochs. En effet, au fur et à mesure de l'apprentissage, le réseau aura tendance à surapprendre plus vite si on laisse un batch sur plusieurs epochs. Pour limiter les appels à getBatch (qui prennent du temps), nous les avons donc laissé en place pendant plusieurs epochs au début, et diminué la fréquence progressivement. Les fichiers .npy fournis ont alors fait office de jeu de données de validation (tous les modèles ont donc pu être comparés sur le même jeu de données de validation).

Nous avons utilisé l'optimiseur Adam. Aux environs d'une loss de 2.76, l'apprentissage avait du mal à progresser. Nous avons alors abaissé le learning rate. Nous sommes passés de la configuration initiale de Adam ($lr = 0.001$) à $lr = 0.0002$. Cela nous a permis de poursuivre l'apprentissage. Nous en avons également profité pour mettre des poids différents pour les deux sorties du réseau, afin de rééquilibrer les valeurs de loss respectives. Nous avons mis un poids

de 0.4 pour la policy, et 5 pour la value. Cela permettait d'obtenir une loss pondérée de 1.01 en policy, 1.20 en value. Ce faisant, nous avons pu améliorer les performances du réseau.



The learning path for our model

3 Résultats

Une fois l'apprentissage terminé, nous avons calculé la précision et la loss sur un nouveau jeu de donné appelé par la fonction getBatch, qui fait donc office de jeu de données de test.

Les performances obtenues sont les suivantes:

- loss: 2.576
- policy loss: 2.337
- value loss: 0.2388
- policy accuracy: 0.409
- value accuracy: 0.606

References

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[2] T. Cazenave, *Spatial Average Pooling for Computer Go*, 06 2019, pp. 119–126.