Dauphine | PSL
UNIVERSITÉ PARIS

Projet de Deep Learning

Monsieur CAZENAVE Tristan

Réseau de neurones profonds pour le jeu de go : Rapport

Projet réalisé par NDIAYE Maïrame et VED Olesia

Master 2 Intelligence Artificielle, Systèmes et Données

26 février 2023

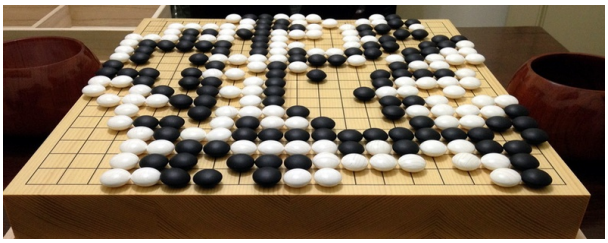
Table des matières

1 Introduction

Nous présentons dans ce rapport le travail que nous avons réalisé pour le projet de Deep Learning. Le but du projet est de construire et entrainer un réseau de neurones pour jouer au jeu de Go.

1.1 Présentation du projet

1.1.1 Présentation du jeu de Go



Le jeu de Go est un jeu traditionnel stratégique Chinois, dans lequel deux joueurs placent des pierres noires et blanches sur les intersections d'un plateau carré appelé le goban composé de 19x19 intersections. Le but du jeu est de capturer des territoires en encerclant les pierres de l'adversaire tout en protégeant ses propres pierres. Un joueur est vainqueur s'il a le plus grand territoire.

1.1.2 Spécificité du jeu en informatique

La grande complexité du jeu, sa richesse stratégique et sa profondeur l'ont rendu très célèbre et intéressant pour les joueurs et les chercheurs en intelligence artificielle. En informatique, le jeu de Go est considéré comme très complexe pour diverses raisons. Premièrement, l'espace d'états est très grand en comparaison à d'autres jeux comme les échecs. Le plateau est composé de 361 positions potentielles pour chaque coup, contre 48 positions pour les échecs. Donc l'espace de recherche pour le Go est assez grand. Deuxièmement, il y a également beaucoup de mouvements possibles. Alors que dans les échecs, chaque pièce a des mouvements spécifiques, pour le Go, les joueurs peuvent placer des pierres sur n'importe quelle intersection libre du goban. Enfin, en raison de la grandeur de l'espace de recherche et du nombre de mouvements, il est difficile de développer des fonctions d'évaluation qui permettent de déterminer quelle est la meilleure position à chaque étape du jeu.

Cependant, malgré ces complexités, grâce aux progrès du Deep Learning, les algorithmes de jeu de Go ont connu des avancées significatives ces dernières années et ont réussi à battre des champions du monde.

L'un des algorithmes les plus connus est l'AlphaGO développé par DeepMind de Google. Cet algorithme utilise des réseaux de neurones profonds pour évaluer les positions de jeu et générer des coups. En 2016, AlphaGo a battu le champion du monde de Go Lee Sedol dans une série de cinq matchs.

Depuis, d'autres algorithmes ont été développés, tel que AlphaGo Zero, qui utilise une approche d'apprentissage par renforcement pour apprendre à jouer à partir de zéro, sans aucune connaissance humaine préalable du jeu. En 2017, l'algorithme a battu AlphaGo dans un match en ligne.

Il y a d'autres projets de recherche en cours pour améliorer l'efficacité et la performance des algorithmes pour le jeu de Go, comme KataGo développé par des chercheurs en intelligence artificielle de Lightvector. Cet algorithme est basé sur le deep learning et l'apprentissage par renforcement. Ce qui le différencie des autres algorithmes de Go, c'est son architecture modulaire qui permet aux développeurs de personnaliser les différents composants du programme. Étant open source, les développeurs peuvent télécharger et utiliser le code source pour leurs propres projets de recherche.

1.1.3 Description du projet

- Le nombre maximal de paramètres du réseau est de 100 000
- Les données d'entraînement proviennent de 10000 parties différentes jouées par le programme Katago
- Le programme prend en entrée 31 plans de taille 19x19
- Les données de sorties sont la politique (un vecteur de taille 361 avec 1 pour le mouvement joué et 0 pour les autres mouvements) et la valeur (Proche de 1 si blanc gagne, sinon proche de 0)
- Les bibliothèques Tensorflow et Keras sont utilisées pour le projet

2 Évolution d'idées de models

Nous pouvons distinguer trois étapes du développment de notre projet pour le jeu de Go :

2.1 Découverte du projet

Pour développer notre modèle de jeu de Go, nous avons commencé par explorer les différentes techniques de Deep Learning enseignées pendant le cours. Nous avons utilisé des stratégies de régularisation telles que l'augmentation du nombre de neurones dans la couche Dense de la tête de valeur, ainsi que l'utilisation de la fonction d'activation ELU en remplacement de la fonction ReLU pour observer l'effet de la sortie négative. Nous avons également ajouté des couches Conv2D avec des nombres de filtres variables et des tailles de noyau de 3 et 4.

Dans un deuxième temps, nous avons mis en place une nouvelle structure de classe pour initialiser notre modèle et une fonction pour ajouter des couches Conv2D, BatchNormalization et MaxPooling. Malgré ces améliorations, notre modèle n'a pas pu obtenir un

bon classement.

C'est pourquoi nous avons décidé de nous tourner vers l'état de l'art afin d'améliorer les performances de notre modèle.

2.2 Residual Networks

Ensuite, nous avons exploré les réseaux résiduels (ResNets). Dans le domaine du jeu de Go sur ordinateur, les ResNets ont été largement utilisés pour améliorer la puissance des moteurs de jeu. Les réseaux neuronaux utilisés dans AlphaGo Zero et AlphaZero de DeepMind sont spécifiquement basés sur des architectures ResNet.Ces architectures ResNet se composent d'une série de blocs résiduels, chacun comprenant plusieurs couches de convolution suivies de normalisation en lots et d'activation ReLU. En ajoutant les sorties de chaque bloc résiduel à l'entrée du bloc, une voie de raccourci est créée pour les gradients, ce qui facilite l'optimisation des réseaux profonds.Dans l'ensemble, les ResNets se sont avérés efficaces pour améliorer les performances des réseaux neuronaux dans diverses tâches, y compris le jeu de Go sur ordinateur.[3]

2.3 Mobile Networks

A la fin nous avons exploré l'implémentation partielle puis complète de l'architecture *Mobile Network* de Google et plus précisiment MobileNetV2. MobileNetV2 est une architecture spécifique qui intègre des convolutions séparables en *profondeur*, un type de couche de convolution qui factorise l'opération de convolution en une convolution en profondeur (appliquant un seul filtre à chaque canal d'entrée) suivie d'une convolution ponctuelle (appliquant un filtre de taille 1x1 pour combiner les sorties de la convolution en profondeur). [2] Cette factorisation réduit le coût de calcul de l'opération de convolution tout en maintenant la précision. En mettant en œuvre MobileNetV2 pour l'informatique de Go, certaines modifications ont été apportées à l'architecture d'origine, telles que l'utilisation de la fonction d'activation ELU au lieu de ReLU et la modification du nombre d'époques et de la taille de lot à 150 et 64, respectivement. De plus, le nombre de filtres a été augmenté de 32 à 192, et une variable "trunk" a été ajoutée avec une valeur de 32. Ces changements ont été apportés dans le but d'améliorer les performances du modèle MobileNetV2 pour la tâche de jouer au Go.

2.4 Comparaison

ResNet et MobileNet sont des architectures populaires en apprentissage profond pour le jeu de Go sur ordinateur. ResNet utilise des connexions résiduelles pour résoudre les problèmes de gradient dans les réseaux profonds, tandis que MobileNet est conçu pour être léger et efficace sur le plan computationnel pour les appareils à ressources limitées. Les deux architectures ont montré des résultats prometteurs pour améliorer la puissance des moteurs de jeu de Go sur ordinateur, mais ResNet convient mieux aux réseaux neuronaux complexes et profonds, tandis que MobileNet est plus adapté aux modèles légers et efficaces.[5] Or, nous sommes contraintes à avoir au plus 100 000 paramètres.

Nous avons donc choisi de continuer avec l'architecture *Mobile Network* qui s'adapte le plus à notre contexte.

3 Notre meilleur modèle

Pour développer notre modèle, nous avons effectué une revue de la littérature afin de prendre en compte les dernières avancées dans l'état de l'art en matière d'amélioration des réseaux de neurones profonds. Nous avons ainsi utilisé plusieurs techniques proposées dans l'article [7] tel que l'activation Swish, le Cosine Anealing...

3.1 Activation Swish

La fonction d'activation *Swish* est une fonction mathématique récemment proposée pour les réseaux de neurones profonds. Elle est définie par la formule suivante :

$$x \cdot sigmoid(x)$$

Dans de nombreuses situations, l'activation Swish s'est révélée plus performante que l'activation ReLU (Rectified Linear Unit). Contrairement à ReLU, qui met à zéro les valeurs négatives, Swish est une fonction continue et différentiable qui fournit une sortie douce et régulière. Cette propriété permet à l'optimiseur de converger plus facilement et plus rapidement vers le minimum global de la fonction de coût.

De plus, des expériences ont montré que l'activation Swish améliore les performances de classification d'images et accélère l'entraînement des modèles de deep learning.

3.2 Cosine Annealing

Le taux d'apprentissage est un paramètre très important dans l'apprentissage d'un réseau de neurone. Il détermine la vitesse avec laquelle les poids du réseau sont mis-à-jour. S'il est trop élevé cela peut entraîner la divergence du modèle et s'il est trop faible cela peut ralentir considérablement l'apprentissage.

Le *cosine annealing* est une technique d'optimisation du taux d'apprentissage. Cette méthode consiste à faire décroître le taux d'apprentissage de manière linéaire jusqu'à un point donné, puis utiliser une fonction cosinus pour réduire progressivement le taux d'apprentissage à zéro. Plus précisément, le taux d'apprentissage est modifié en utilisant la formule suivante à chaque époque :

$$\eta_t = \eta_{\min} + \frac{1}{2}(\eta_{\max} - \eta_{\min}) \left(1 + \cos\left(\frac{T_{\text{cur}}}{T_i}\pi\right)\right)$$

où η_t est le taux d'apprentissage à l'époque t, $\eta_{\min}$ et $\eta_{\max}$ sont respectivement les taux d'apprentissage minimum et maximum, T_{cur} est l'époque actuelle et T_i est le nombre total

d'époques.

La fonction cosinus permet ainsi une diminution progressive du taux d'apprentissage, ce qui peut aider à éviter les problèmes de surapprentissage et de convergence trop rapide.

3.3 Modèle final

Après avoir mené plusieurs recherches sur les architectures et les paramètres optimaux pour notre modèle, nous avons finalement sélectionné les choix suivants pour maximiser la capture d'informations et les performances du modèle :

- Notre architecture repose sur l'utilisation de 30 blocs de Mobile Network, avec 64 filtres d'expansion et 16 filtres trunk. Nous avons créé cette architecture en nous basant sur le code source de Keras [6] [4].
- Nous avons préféré la fonction d'activation Swish à la ReLU car elle offre de meilleures performances pour notre modèle
- Pour la mise à jour du taux d'apprentissage, nous avons utilisé la technique de Cosine Annealing. Nous avons ainsi fixé le taux d'apprentissage à varier entre 0.001 et 0.0005. Nous avons également utilisé l'optimisation Adam pour améliorer davantage les performances de notre modèle
- Nous avons entraîné notre modèle en utilisant une taille de batch de 64 et sur 1000 époques

Avec cette configuration nous obtenons un modèle composé de *98033 paramètres*.

Vous retrouvez en Annexe A le code d'implémentation du modèle.

3.4 Résultat

Pour évaluer les performances de notre modèle, nous utilisons deux métriques couramment utilisées dans le domaine de l'apprentissage automatique : la *categorical accuracy* pour évaluer la prédiction de la politique et la *MSE (Mean Squared Error)* pour évaluer la valeur donnée par le modèle. La categorical accuracy mesure la proportion de prédictions correctes pour la classification des actions prises par l'algorithme tandis que la MSE mesure l'écart moyen entre les valeurs prédites et les valeurs réelles attendues.

Nous avons tracé les courbes d'évolution en validation de ces deux métriques au fil des époques de l'entraînement de notre modèle.

Comme le montrent les graphiques, la courbe d'accuracy croît progressivement au fil des époques pour atteindre une valeur finale de **0.43**. Cela indique que le modèle est capable de prédire correctement la politique à suivre dans un jeu donné avec une précision acceptable. En revanche, la courbe de la MSE décroit au fur et à mesure des époques pour atteindre une valeur finale de **0.0745**. Cela indique que le modèle est capable de prédire les valeurs attendues avec une faible erreur moyenne, ce qui est un bon indicateur de sa qualité de prédiction.

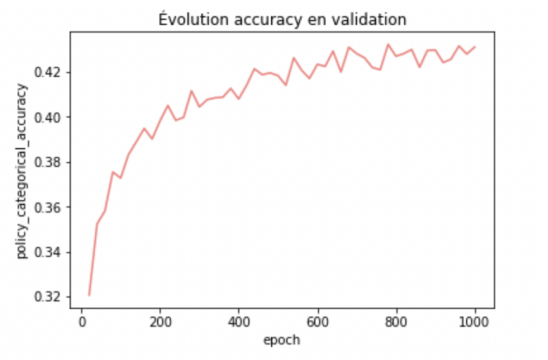


FIGURE 1 – Categorical accuracy

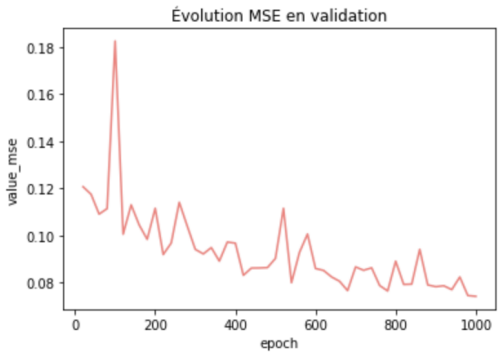


FIGURE 2 – MSE

3.5 Architecture MixConv

Nous avons cherché à améliorer notre modèle en remplaçant l'architecture MobileNetwork par une architecture *MixConv* présenté dans [7]. Cette dernière s'inspire de MobileNetV2 et introduit une nouvelle couche de convolution combinant des convolutions en profondeur et des convolutions classiques. Cette combinaison permet de mélanger plusieurs noyaux de tailles différentes dans une même convolution, offrant ainsi une solution pour pallier les limites de la taille unique du noyau dans l'architecture MobileNetwork. En conséquence, l'architecture MixConv peut potentiellement améliorer les performances en termes de précision de classification.
Nous avons tenté de modifier notre modèle en ajoutant une couche MixConv avec deux tailles de noyaux 3x3 et 5x5, mais cette modification a entraîné une baisse de performance. Par conséquent, nous avons décidé de conserver notre modèle sans cette couche de convolution mixte.

4 Améliorations possibles

Nous aurions aimé évaluer les performances de MobileNetV3 dans le cadre du jeu de Go. Cette architecture utilise exclusivement des convolutions séparables en profondeur et inclut de nouvelles fonctionnalités comme les couches de goulot linéaires et les blocs de compression-excitation, contrairement à MobileNetV2 qui combine une couche de convolution standard et une couche de convolution séparable en profondeur.[1]

5 Conclusion

En conclusion, nous avons opté pour l'architecture Mobile Network comme base pour notre modèle de deep learning. Cette architecture utilise des convolutions en profondeur pour capturer des caractéristiques complexes et abstraites, tout en réduisant le nombre de paramètres nécessaires. Nous avons également choisi d'utiliser la fonction d'activation Swish plutôt que ReLU pour améliorer les performances et la vitesse d'entraînement du modèle. Enfin, nous avons utilisé la technique du cosine annealing pour optimiser le taux d'apprentissage et éviter les problèmes de surapprentissage et de convergence trop rapide.

Nous avons compris l'importance de l'état de l'art. Faire l'état de l'art avant un projet est crucial pour comprendre les recherches et projets similaires dans le domaine. Cela aide à déterminer les meilleures pratiques et méthodes efficaces pour le projet, situer le projet dans le contexte plus large du domaine de recherche et concevoir un projet qui apporte une valeur ajoutée.

Références

[1] Grace Chu Liang-Chieh Chen Bo Chen Mingxing Tan Weijun Wang Yukun Zhu Ruoming Pang Vijay Vasudevan Quoc V. Le Hartwig Adam Andrew Howard, Mark Sandler. Searching for mobilenetv3. 2019.

[2] Tristan Cazenave. Mobile networks for computer go.

[3] Tristan Cazenave. Residual networks for computer go.

[4] F. Chollet. *Deep Learning with Python.* Manning 2017.

[5] Fransiska. The differences between inception, resnet, and mobilenet.

[6] F. Chollet *et al.* "keras,". 2019.

[7] Mathurin Videau Tristan Cazenave, Julien Sentuc. *Cosine Annealing, Mixnet and Swish Activation for Computer Go.* Advances in Computer Games 2021.

Implémentation du modèle

```
import tensorflow as tf
import tensorflow.keras as keras
import numpy as np
from tensorflow.keras import layers
from tensorflow.keras import regularizers
from tensorflow.keras import backend as K
import math
import gc
import golois

class CosineAnnealingScheduler(keras.callbacks.Callback):
  def __init__(self, T_max, eta_max, eta_min=0):
      super(CosineAnnealingScheduler, self).__init__()
      self.T_max = T_max
      self.eta_max = eta_max
      self.eta_min = eta_min

  def on_epoch_begin(self, epoch, logs=None):
      lr = self.eta_min + (self.eta_max - self.eta_min) * (1 +
         math.cos(math.pi * epoch / self.T_max)) / 2
      K.set_value(self.model.optimizer.lr, lr)


T_max = 1000
eta_max = 0.001
eta_min = 0.0005
cosine_annealing = CosineAnnealingScheduler(T_max, eta_max,
   eta_min)


planes = 31
moves = 361
N = 10000

epochs = 1000
batch = 64
blocks=30
filters = 64
trunk = 16

input_data = np.random.randint(2, size=(N, 19, 19, planes))
input_data = input_data.astype ('float32')

policy = np.random.randint(moves, size=(N,))
policy = keras.utils.to_categorical (policy)

value = np.random.randint(2, size=(N,))
```

```
value = value.astype ('float32')

end = np.random.randint(2, size=(N, 19, 19, 2))
end = end.astype ('float32')

groups = np.zeros((N, 19, 19, 1))
groups = groups.astype ('float32')

print ("getValidation", flush = True)
golois.getValidation (input_data, policy, value, end)


def bottleneck_block(x, expand=filters, squeeze=trunk):
  m = layers.Conv2D(expand, (1,1),
     kernel_regularizer=regularizers.l2(0.0001), use_bias =
     False)(x)
  m = layers.BatchNormalization()(m)
  m = layers.Activation("swish")(m)
  m = layers.DepthwiseConv2D((3,3), padding="same",
     kernel_regularizer=regularizers.l2(0.0001), use_bias =
     False)(m)
  m = layers.BatchNormalization()(m)
  m = layers.Activation("swish")(m)
  m = layers.Conv2D(squeeze, (1,1),
     kernel_regularizer=regularizers.l2(0.0001), use_bias =
     False)(m)
  m = layers.BatchNormalization()(m)
  return layers.Add()([m, x])


def getModel ():
  input = keras.Input(shape=(19, 19, 31), name="board")
  x = layers.Conv2D(trunk, 1, activation='swish', padding='same',
     kernel_regularizer=regularizers.l2(0.0001))(input)
  x = layers.BatchNormalization()(x)
  for i in range (blocks):
    x = bottleneck_block (x, filters, trunk)

  policy_head = layers.Conv2D(1, 1, activation="swish",
     padding="same", use_bias = False,
     kernel_regularizer=regularizers.l2(0.0001))(x)
  policy_head = layers.Flatten()(policy_head)
  policy_head = layers.Activation("softmax",
     name="policy")(policy_head)

  value_head = layers.Conv2D(1, 3, activation='swish',
     padding='same', use_bias = False,
     kernel_regularizer=regularizers.l2(0.0001))(x)
```

```
  value_head = layers.Conv2D(1, 1, activation='swish',
     padding='same', use_bias = False,
     kernel_regularizer=regularizers.l2(0.0001))(value_head)
  value_head = layers.GlobalAveragePooling2D()(x)
  value_head = layers.Dense(80, activation="swish",
     kernel_regularizer=regularizers.l2(0.0001))(value_head)
  value_head = layers.Dense(1, activation="sigmoid",
     name="value",
     kernel_regularizer=regularizers.l2(0.0001))(value_head)

  model = keras.Model(inputs=input, outputs=[policy_head,
     value_head])
  return model


model = getModel()
model.summary ()

model.compile(optimizer=keras.optimizers.Adam(learning_rate=0.001),
              loss={'policy': 'categorical_crossentropy',
                 'value': 'binary_crossentropy'},
              loss_weights={'policy' : 1.0, 'value' : 1.0},
              metrics={'policy': 'categorical_accuracy', 'value':
                 'mse'})

for i in range(1, epochs + 1):
    print('epoch ' + str(i))
    golois.getBatch(input_data, policy, value, end, groups, i * N)
    history = model.fit(input_data,
                        {'policy': policy, 'value': value},
                           epochs=1,
                           batch_size=batch,callbacks=[cosine_annealing])
    if (i % 5 == 0):
        gc.collect()
    if (i % 20 == 0):
        golois.getValidation(input_data, policy, value, end)
        val = model.evaluate(input_data,
                             [policy, value], verbose=0,
                                batch_size=batch)
        print("val =", val)
        model.save('NDIAYE_VED.h5')
```