Objectif

L'objectif de ce projet est de réaliser puis d'entrainer un réseau de neurones profond capable de jouer au GO. La performance du modèle sera évaluée par l'organisation de « compétitions » entre les solutions des différents élèves de la promotion IASD4.

Afin de s'assurer de conditions équitables entre tous les modèles, ceux-ci ne devront pas avoir plus de 100 milles paramètres. Ils devront être sauvegardés au format h5 de Keras. Il est demandé par ailleurs que le réseau proposé ait les mêmes têtes de sortie.

Principales consignes

Le tenseur d'entrée est composé de 31 plans de 19x19. En sortie le réseau doit comprendre deux têtes appelées « value » et « policy ». La tête « policy » sera un vecteur de taille 361 (19*19) encodant 1.0 pour les coups joués et 0.0 pour les autres coups. La tête « value » donnera une valeur comprise entre 0.0 et 1.0 ; valeur donnée par la recherche arborescente de Monte-Carlo représentant la probabilité pour les points blancs de gagner.

Le corps du réseau est à définir tant en termes d'architecture (choix des couches, profondeur ou largeur) qu'en termes de valeurs hyperparamètres pour l'entrainement.

Méthodologie

- La définition du meilleur modèle pose en **1^{ier} lieu le choix des critères d'évaluation des performances**. Puisqu'il y a deux têtes, l'évaluation devra tenir compte des métriques de chacune d'elle.

  Pour la tête « policy », la « loss » fonction utilisée est la « categorical cross entropy » et la métrique est la « categorical accuracy ». Je rechercherai donc à maximiser cette « accuracy ».

  Pour la tête « value », la « loss » fonction utilisée est la « binary cross entropy » et la métrique est la « mse ». Je rechercherai donc à minimiser cette erreur.

Et pour l'évaluation globale, je m'intéresserai à la « loss » globale des modèles, ce qui posera la question de la pondération des « loss » de la « policy » et de la « value » dans la « loss » globale.

- **Concernant l'architecture du modèle** (dans le respect de la limite des 100K paramètres), j'ai envisagé trois axes de prospection :
  - Nature et combinaison des couches (dense, convolutive, activation, « pooling » etc.) ;
  - Nombre successif de couches (que j'appellerai 'profondeur') ;
  - Taille des 'tensors' en sortie des couches convolutives (nombre de plans, taille des 'kernel') que je me permettrais d'appeler 'largeur'.

- Une fois l'architecture définie, la question du **choix des hyperparamètres d'entrainement** se posera avec le choix entre-autre :
  - De l'optimiseur ('SGD','ADAM'…)
  - Des valeurs du 'Learning Rate', du nombre d'époques, la taille du 'batchsize'…,
  - D'une éventuelle régularisation, de la différenciation des « weigth-loss ».

Compte-tenu du nombre possibilités et après quelques essais exploratoires, je me suis fixé le **cadre suivant pour prospecter les différentes possibilités** :

- Étape 0 : évaluation d'un modèle de référence dit 'base line'.
- Étape 1 : définition d'un 1ier ensemble d'hyperparamètres 'optimisés' avant « tunning » final, afin de distinguer l'impact des hyperparamètres de celui de l'architecture.
- Étape 2 : évaluation de modèles en jouant soit sur le corps commun et/ou les couches cachées dédiées à chacune des têtes.
- Étape 3 : entrainement du meilleur modèle sur un nombre important d'époques et « fine tunning » de certains hyperparamètres.
- Étape 'bonus' : bien que cette étape puisse faire partie de l'étape 2, je me suis proposé pour un modèle utilisant les mêmes types de couches convolutives, de comparer les performances entre un modèle profond (nombre de couches successives) versus un modèle large (nombre de plans des couches convolutives) à iso nombre de paramètres.

- Et **dernier point concernant le suivi des performances en généralisation,** je me suis posé la question de la meilleure façon de procéder.

Comme j'ai souhaité suivre l'évolution des valeurs de « l'accuracy » et de la « mse » en test afin détecter tout éventuel « overfitting », j'ai splitté le jeu de train[1] en deux parties (80% pour le train) lors du « fit » du modèle (et du coup sans intervenir sur les données issues du « GetValidation »). J'utiliserai l'option « cvs_logger » du paramètre « callbacks » pour récupérer ces valeurs.

```
history = model.fit(input_data,
                    {'policy': policy, 'value': value},validation_split=0.2,
                    epochs=1, batch_size=batch,verbose=1,callbacks=[csv_logger])
```

Et pour éviter de trop réduire la taille du jeu de données réellement utilisé pour le train, j'ai augmenté le nombre de données à 30 000 (paramètre N) et joué sur le « Garbage Collect » afin d'éviter les problèmes de saturation de RAM sur Google Collab…sachant qu'une plus grande taille du jeu de données à chaque itération, améliore la qualité de l'entrainement.

[1] Dans le code de référence, la fonction 'GetValidation' permet d'obtenir le jeu d'évaluation en généralisation et la fonction 'GetBatch' permet d'obtenir le jeu de train.

Performances du modèle de référence

Pour le modèle de référence, je suis parti du 'Mobile Net' fourni dans l'énoncé en conservant ses hyperparamètres, dont les principales caractéristiques sont pour rappel :

```
#optimizer SGD avec Learning_rate=0.0005 et Momentum=0.9
#loss_weights={'policy' : 1.0, 'value' : 1.0}
#regularisation de 1 e-4 sur la tête policy , 1e-3 sur la tête value.
#taille de l'échantillon par epoch N=10K
#Batch size=128
#epoch =100
```

Avec 5 boucles du bloc principal (« Inverted Residual Block ») et les couches natives dédiées aux deux têtes, le modèle comprend 99,7K paramètres.

Après 100 époques, on obtient les valeurs suivantes en évaluation comme 'base line' de référence :

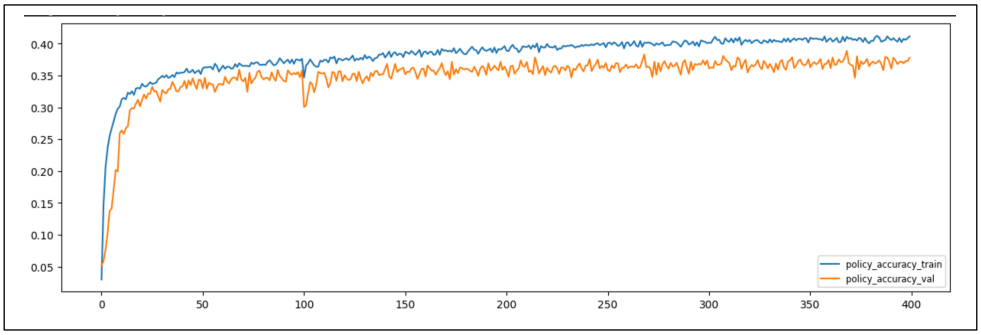
'loss' du modèle	'accuracy policy'	'mse value'
3,4934	0,3406	0,1183

Attention : à ce stade, il n'y a pas de pondération des 'loss' de 'policy' et 'value' dans la 'loss' globale

Choix d'un 1ier ensemble d'hyperparamètres 'optimisés' (avant 'tunning')

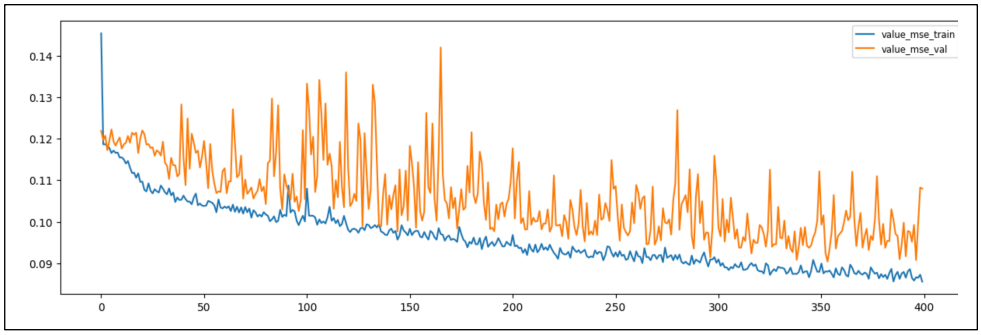
Le 1ier constat lors d'essais préliminaires a été que la **limite de 100K paramètres ne permet pas « d'overfitter »**. En effet, on constate qu'après plusieurs centaines d'époques, les estimations en généralisation (réalisé sur les 20% du jeu pour le test) ne se dégrade pas comme le montre les deux graphes suivants réalisés sur le modèle 'Mobile Net' de référence.

Il apparait du coup inutile de mettre en œuvre de la régularisation ou du 'drop out'.



'accuracy policy' (train et test sur le 'GetBatch')

Reste une surprise non expliquée (et constatée lors de différents essais) concernant les grandes variations de l'estimation en généralisation sur la « mse value ». Cela est peut-être lié à la façon dont le 'getBatch' génère les jeux de données (ou une erreur de manipulation non détectée).



'mse value' (train et test sur le 'GetBatch')

Le deuxième constat est que la « mse value » s'améliore nettement moins vite que la « policy accuracy ». Du coup il m'est apparu intéressant de modifier le poids de la « loss value » dans la « loss globale ». Après plusieurs essais, j'ai opté pour une « weighted loss » de 1 pour la « categorical crossentropy » de « policy » et de 9 pour la « binary crossentropy » pour la « value ».

```
model.compile(optimizer=opt,
              loss={'policy': 'categorical_crossentropy', 'value': 'binary_crossentropy'},
              loss_weights={'policy' : 1.0, 'value' : 9.0},
              metrics={'policy': 'categorical_accuracy', 'value': 'mse'})
```

Pour le choix des autres hyperparamètres, je me suis inspiré des conseils trouvés dans livre [4] avec :

De façon générale : if …"your loss is stuck..it is always a problem with the configuration of the gradient descent process: your choise of optimizer, the distribution of initial values in the weights of your models, your LR or your batch size ..all these parameters are inter-dependents as such it is usually sufficient to tune the LR and the batch size while keeping the rest of the para constant".

Et plus spécifiquement concernant :

- Le "batch-size": "Increasing the batch size …with more samples will lead to gradients that are more informative and less noisy (lower variance)" => il m'a semblé intéressant d'augmenter cette valeur de 128 à 1024 (après plusieurs essais non concluants avec des valeurs intermédiaires).

- Le 'Learning Rate': "Lowering or increasing the LR…a LR to high may lead to updates that vastly overshoot a proper fit, …too low may make the training so slow that it appears to

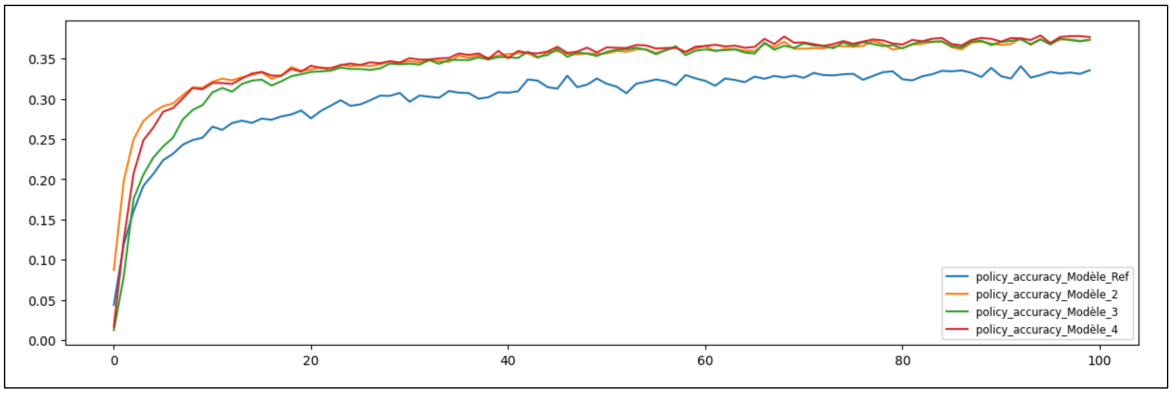
stall" => il m'a semblé utile d'adopter une méthode avec un 'Learning Rate' dégressif au fil des époques [2].

- L'optimiseur : "Choise of optimizer… depends of distribution of initial values in the weights of your model" => il semblé intéressant de tester un autre optimiseur avec "Adam'

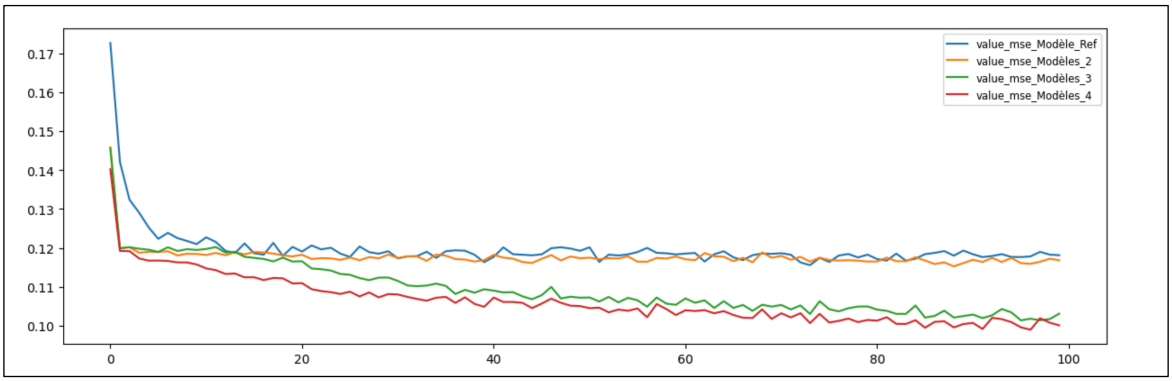
Voici ci-dessous les résultats issus d'essais comparatifs sur le modèle de référence 'Mobile Net' illustrant l'impact des différents des hyperparamètres évoqués sur la « policy accuracy » et « mse value » :

Modèle testé	Valeur des principaux hyperparamètres
Modèle 1 = 'Mobile Net' initial	SGD (LR=0.0005, momentum=0.9) /loss_weights={'policy' : 1.0, 'value' : 1.0}/regul de 1e-4 sur policy et 1e-3 sur value /N=10K /Batch size=128/ Epoch =100
Modèle 2 avec modification de :	Modèle 1 avec modification de N=30K et Batch size=1024
Modèle 3 avec modification de :	Modèle 2 avec régularisation de 1e-4 sur policy et loss_weight s{'policy' : 1.0, 'value' : 9.0}
Modèle 4 avec modification de :	Modèle 3 optimizeur ADAM et LR=0,01 sur les 100 premières époques

Illustrations avec les courbes obtenues sur le train



'Policy Accuracy' (train et test sur le 'GetBatch')



'Value MSE' (train et test sur le 'GetBatch')

[2] Avec successivement 0.01 , 0.008, 0.006, 0.004, 0.002, 0.001 sur les 100 premières époques, puis les 100 suivantes etc. Détail technique, la modification dynamique du 'Learning Rate' ('LearningRateScheduler' et 'ReduceLROnPlateau' du paramètre 'call_backs' du 'fit') ne fonctionne que s'il y a plusieurs époques dans le 'fit'. J'ai opté pour la sauvegarde et une recompilation du modèle toutes les 100 époques avec un 'LR' modifié. Il y a également la possibilité d'utiliser la méthode « set-value » de la bibliothèque « back-end » de Keras pour modifier le LR.

La conclusion est que le **choix de l'optimiseur a été déterminant** en améliorant nettement la « policy accuracy » et la « mse value » (courbe en rouge appelée modèle 4).

Notons que la **pondération de « loss value » dans la « loss » a eu un effet positif** sur la réduction de la « mse value » (courbe en vert modèle 3).

Performance du meilleur modèle

Le modèle qui a donné les meilleurs scores est issu d'une variante du Mobile Net avec dans :

- Le corps commun au sein de « l'Inverted Residual Block », l'ajout d'une couche d'excitation et « squeeze » en s'inspirant de l'article [3];
- La branche dédiée à la tête « value », l'ajout d'une couche « AvgPool2D » en s'inspirant de l'article [2].

Le modèle final dispose de 98 256 paramètres.

Description du sous-bloc de Squeeze et d'excitation

```
#artcile sur Squeeze and Excitation
def SE_block(t,filters,ratio=16):
  se_shape = (1,1,filters)
  se = GlobalAveragePooling2D()(t)
  se = Reshape(se_shape)(se)
  se = layers.Dense(filters//ratio,activation='relu',use_bias = False)(se)
  se = layers.Dense(filters,activation='sigmoid',use_bias = False)(se)
  x = layers.Multiply()([t,se])
  return x

```

Focus sur l'excitation et squeeze

Ce sous-bloc a été rajouté à la fin du bloc principal de « l'Interveted Residual Block » comme le montre le code ci-dessous :

```
def bottleneck_block_bis(x, expand=6*filters, squeeze=filters):
  m = layers.Conv2D(expand, (1,1), use_bias = False)(x)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.DepthwiseConv2D((3,3), padding='same', use_bias = False)(m)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.Conv2D(squeeze, (1,1), use_bias = False)(m)
  m = layers.BatchNormalization()(m)

  m=SE_block(m,filters)
  return layers.Add()([m, x]) #ici on rajoute l'entrée du bloc à l'entré
```

Inverted Residual Block modifié

Et une couche « AvgPool2D » a été ajoutée aux couches cachées dédiées à la tête « value »

```
#value_head=ZeroPadding2D(padding=(1,1))(x)
value_head=AvgPool2D(pool_size=(2,2),strides=2,padding='same')(x)
value_head=ZeroPadding2D(padding=(1,1))(value_head)
value_head=AvgPool2D(pool_size=(2,2),strides=2,padding='same')(value_head)
value_head = layers.Conv2D(1, 1, activation='relu', padding='same', use_bias = False)(value_head)
value_head = layers.Flatten()(value_head)

value_head = layers.Dense(50, activation='relu')(value_head)
value_head = layers.Dense(9, activation='relu')(value_head)
value_head = layers.Dense(1, activation='sigmoid', name='value')(value_head)
```

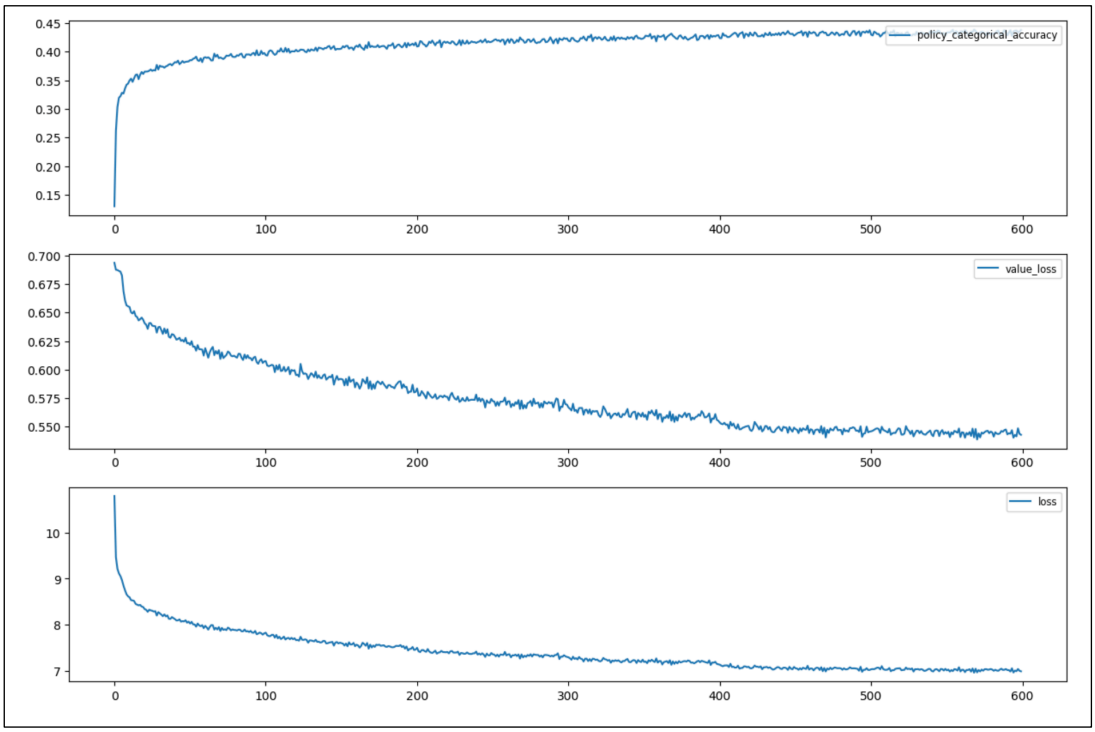
Modification des couches dédiées à la tête Value

J'ai conservé les hyperparamètres 'optimisés' mais en réduisant le 'Learning Rate' de 0.02 toutes les 100 époques (comme décrit dans le paragraphe méthode/choix des hyperparamètres).

En termes de performance après 600 époques, les résultats suivants sur le jeu de validation (issu du « GetValidation ») sont :

« Loss » du modèle	« Policy accuracy »	« Value_mse »
7.1769	0.4307	0.06598

Pour illustration, voici les courbes d'évolution en fonction du nombre d'époques de « policy accuracy », de la « mse value » et la de « loss » globale sur le train (avec une pondération 1 pour la « policy » et 9 pour la « value »).



Comparaison de l'impact de travailler en largeur (nombre de plans) par rapport à la profondeur du modèle (à iso nombre de paramètres)

Le principe adopté a été de comparer la performance de deux modèles dont les types couches sont les mêmes, à la différence près que dans un cas on empile les couches (« profondeur ») et dans l'autre on augmente le nombre de plans dans les couches convolutives. **La motivation m'a semblé double :**

1) De vérifier si l'augmentation du nombre de plans dans les couches convolutives permet de capturer plus de 'caractéristiques' …sous la contrainte des 100K paramètres à ne pas dépasser.
2) De vérifier si l'empilement du nombre de couches, réduit la 'vitesse' d'apprentissage du modèle (qui au final pour un même nombre d'époques, doit se traduire par des valeurs de la « policy accuracy » et de la « mse value » moins bonnes.

Deux essais ont été réalisés sur :

- Le modèle de référence 'Mobile Net' ;
- Le meilleur modèle.

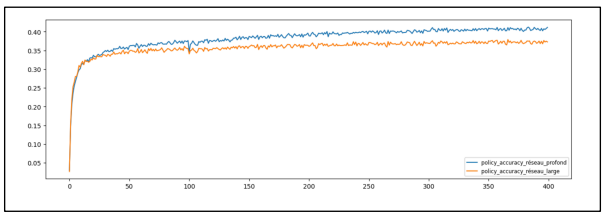
Compte-tenu de la contrainte des 100K paramètres, **le modèle de référence a été modifié pour ne conserver que 2 'Residual Inverted Blocks' avec 54 plans** en entrée des couches convolutives (contre 5 blocks et 32 plans dans le modèle de référence).

Le meilleur modèle a été modifié pour atteindre 48 plans en entrée des couches convolutives pour atteindre 103K (au lieu des 98,3K dans sa version la plus profonde).

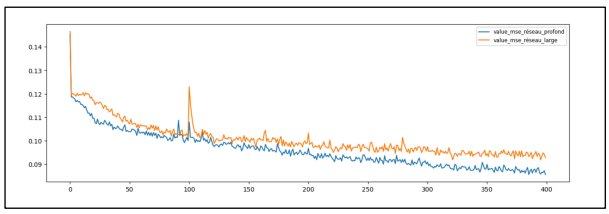
En termes de résultats, j'ai pu constater que les modèles les plus profonds proposaient les meilleurs performance (observations sur le train), même si la différence n'est pas exceptionnelle.

	Comparaison valeur réseau 'profond' versus 'large'		
	loss	Policy accuracy	Mse value
Modèle de référence	8.735 vs 8.6361	0,4071 vs 0,3713	0,0939 vs 0,1053
Meilleur modèle	7.4025 vs	0.4248 vs 0,414	0.0730 vs 0,0783

Résultats pour le modèle de référence

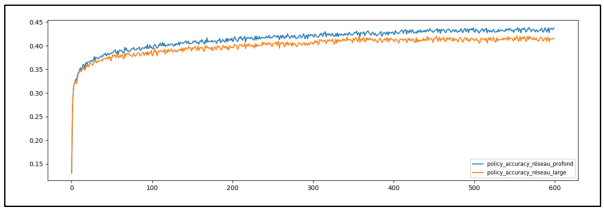


« Policy accuracy »

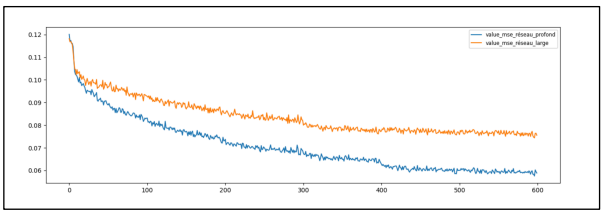


« Value mse »

Résultats pour le meilleur modèle



« Policy accuracy »



« Value mse »

Ces résultats m'ont surpris, car je pensais qu'un plus grand nombre de plans dans les couches de convolution permettrait de capturer plus de caractéristiques et donc au final donner une meilleure performance du modèle. Mais cela n'est pas ici le cas. L'explication provient peut-être de la limite de 100K paramètres.

En termes de vitesse d'apprentissage, il n'y a aucune différence notable sauf peut-être pour la « mse value » avec un résultat contre-intuitif (le réseau le plus profond a une pente d'amélioration plus importante). Il est possible que le faible nombre de paramètres en soit également la cause.

Conclusion

Ma principale difficulté a été (et reste) d'anticiper l'impact d'une couche sur la performance du modèle.

> En effet en-dehors quelques principes très généraux comme l'utilité de la « Batch Normalisation » ou de certaines couches d'activation (comme « sigmoid » ou encore « softmax »), je dois avouer n'avoir rien trouver de mieux que de réaliser des essais systématiques et de lire des articles de référence sur le sujet.

En revanche, le travail sur les hyperparamètres a été plus facile à structurer, la combinatoire étant plus facile à encadrer.

Je regrette la contrainte des 100K paramètres qui n'a pas permis de jouer sur les hyperparamètres afin de trouver l'optimum entre « overfitting » et « underfitting », même si je me doute que les ressources nécessaires pour atteindre ce seuil de « l'overfitting » nécessite de grosses capacités de traitement.

Sinon les deux points les plus notables de cet exercice sont de ma compréhension :

- L'importance du choix de l'architecture (en comparaison de l'augmentation « brute » du nombre de paramètres)
- L'importance de l'entrainement (notamment choix de l'optimiseur, pondération des loss).

ANNEXES

Performances des autres modèles étudies

Ces modèles ont été étudiés avec les paramètres 'optimisés' (sauf exception) **avec 100 époques**.

1) Performances du «Shuffle Net» avec l'utilisation du « GlobaAveragePooling » pour les couches cachées dédiée à la tête « value »

Le modèle proposé comprend 99,3K paramètres avec 14 répétitions du block principal du « Shuffle Net » et l'ajout d'un « GlobalAveragePooling2D » en entrée des couches dédiées à la tête « value ». A noter que pour des raisons de limitations de mémoire RAM sous le notebook Google Collab, j'ai dû réduire le « batchsize » à '256'.

Description des couches cachées dédiées à la tête « Value »

```
#new tetes -> plante
x = layers.BatchNormalization()(x)

#couches de sortie policy
policy_head = layers.Conv2D(1, 1, activation='relu', padding='same',kernel_regularizer=regularizers.l2(0.001))(
policy_head = layers.Flatten()(policy_head)
policy_head = layers.Activation('softmax', name='policy')(policy_head)

#couches de sortie value
value_head = layers.GlobalAveragePooling2D()(x)
value_head = layers.Dense(50, activation='relu')(value_head)
value_head = layers.Dense(1, activation='sigmoid', name='value')(value_head)

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

model.summary ()
```

Ce modèle donne de meilleurs résultats que le modèle de référence (avec les hyperparamètres initiaux)

Modèles	« Loss » du modèle	« Policy accuracy »	« Value_mse »
Shuffle Net + AvgGP	8.1329	0.3717	0.0897
Mobile Net	3,4934	0.3406	0,1183

2) Performances de la variante du « Mobile Net » avec un nouveau corps de couches cachées communes comme indiqué dans l'article [1] (modèle 3)

Le modèle proposé comprend 110K paramètres.

Il m'a semblé intéressant de le conserver comme une illustration d'un cas où un nombre élevé de paramètres ne garantit de meilleures performances.

Sur ce modèle j'ai modifié le corps du bloc principal « Mobile Net » par l'architecture proposé dans l'article [1] tout en conservant les couches dédiées aux deux têtes. Le principe est d'avoir deux couches de convolution en parallèle (chacune avec des « strides » spécifiques), de les sommer avant d'appliquer une activation 'Relu' au tenseur résultant (modèle 3 décrit ci-après).

```python
def bottleneck_block(x,filters=32):
  m = layers.Conv2D(filters, (1,1))(x)
  m = layers.BatchNormalization()(m)

  n = layers.Conv2D(filters, (5,5),padding='same')(x)
  n = layers.BatchNormalization()(n)

  output=layers.Add()([m,n]) #ici on rajoute l'entrée du bloc à l'entrée du bloc
  output=layers.Activation('relu')(output)

  return output
```

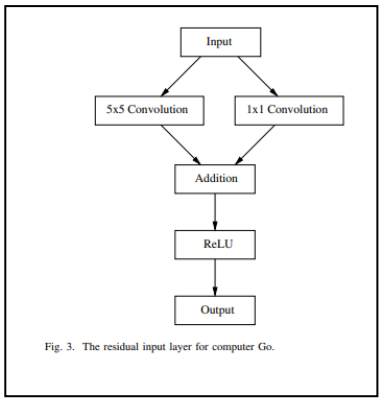


Fig. 3. The residual input layer for computer Go.

The residual layer used for image classification adds the input of the layer to the output of the layer. It uses two convolutional layers before the addition. The ReLU layers are put after the first convolutional layer and after the addition. The residual layer is shown in figure 2. We will experiment with this kind of residual layers for our Go networks.

The input layer of our Go networks is also residual. It uses a 5×5 convolutional layer in parallel to a 1×1 convolutional layer and adds the outputs of the two layers before the ReLU layer. It is depicted in figure 3.

The output layer of the network is a 3×3 convolutional layer with one output plane followed by a SoftMax. All the hidden layers use 256 feature planes and 3×3 filters.

We define the number of layers of a network as the number of convolutional layers. So a 28 layers network has 28 convolutional layers corresponding to 14 layers depicted in figure 2.

Ce modèle plus compliqué reste moins performant que le modèle de précédent qui se contentait d'ajouter une couche « GlobaAveragePooling) au modèle de référence .

Modèles	« Loss » du modèle	« Policy accuracy »	« Value_mse »
3	8.5288	0.3598	0.0995
Shuffle Net + AvgGP	8.1329	0.3717	0.0897
Mobile Net	3,4934	0.3406	0,1183

3) Performances de la variante du Mobile Net avec l'utilisation du «Spatial Average Pooling » en entrée des couches cachées dédiées à la tête 'value' comme préconisé dans l'article [2] (modèle 4)

Le modèle proposé comprend 97,48K paramètres avec 6 répétitions du bloc principal du 'Mobile Net' et la modification des couches cachées dédiées à la tête « value » à travers l'utilisation du « Spatial Average Pooling » (modèle 4).

Je me suis intéressé à cette article [2] car il propose une réponse à la faible amélioration de la 'value' que j'ai pu constater dès les premiers essais. Après applications des modifications proposées dans l'article, on obtient l'architecture suivante pour les couches dédiées à la tête « value » :

```
#couches de sortie policy
policy_head = layers.Conv2D(1, 1, activation='relu', padding='same',kernel_regularizer=regularizers.l2(0.001))(x)
policy_head = layers.Flatten()(policy_head)
policy_head = layers.Activation('softmax', name='policy')(policy_head)

#value_head=ZeroPadding2D(padding=(1,1))(x)
value_head=AvgPool2D(pool_size=(2,2),strides=2,padding='same')(x)
value_head=ZeroPadding2D(padding=(1,1))(value_head)
value_head=AvgPool2D(pool_size=(2,2),strides=2,padding='same')(value_head)
value_head = layers.Conv2D(1, 1, activation='relu', padding='same', use_bias = False)(value_head)
value_head = layers.Flatten()(value_head)

value_head = layers.Dense(50, activation='relu')(value_head)
value_head = layers.Dense(9, activation='relu')(value_head)
value_head = layers.Dense(1, activation='sigmoid', name='value')(value_head)

```

En termes d'explication, j'ai compris que le « Spatial Average Pooling » facilite la propagation de l'information de gagner au cours des différentes époques, en prenant la valeur moyenne sur une fenêtre d'entrée (dont la taille est définie par le paramètre « pool_size ») pour chaque plan d'entrée de la « value » (chaque plan représentant la probabilité pour chaque nœud du damier d'être noir à la fin de la partie).

The Spatial Average Pooling is meaningful for a value network since such a net works outputs a winning probability that is related to the estimated score of the board If neurons in the various planes represent the probability of an intersection to be Black territory in the end, averaging such probabilities gives the winning probability. So using Spatial Average Pooling layers can push the value network to represent probabilities o ownership for the different parts of the board and help the training process.

We used Spatial Average Pooling in the last layers of Golois value network with a size 2x2 and a stride of 2 as in the table 1 example.

When applying Spatial Average Pooling with a size 2x2 and a stride of 2 to 19x19 planes, we add a padding of one around the 19x19 plane. Therefore the resulting planes are 10x10 planes. When applying Spatial Average Pooling again to the 10x10 planes with a padding of one we obtain 6x6 planes. The last convolutional layer of the value network is a single 6x6 plane. It is flattened to give a vector of 36 neurons. It is then followed by a 50 neurons layer and the final 9 neurons output followed by a Sigmoid (the value network outputs the probability of winning between 0 and 1).

Modèles	« Loss » du modèle	« Policy accuracy »	« Value_mse »
4	8.400	0.3770	0.1005
3	8.5288	0.3598	0.0995
Shuffle Net + AvgGP	8.1329	0.3717	0.0897
Mobile Net	3,4934	0.3406	0,1183

4) Performances de la variante du Mobile Net avec des modifications de l'article [3] (modèle 5)

Le modèle est constitué de 98,3K paramètres avec 6 répétitions du bloc principal adapté du 'Mobile Net' et les couches cachées dédiées aux têtes initiales.

La modification du bloc Mobile Net a consisté à ajouter avant l'addition, le résultat de la multiplication de deux couches denses précédées par un 'Global Average Pooling' (le détail est présenté ci-dessous).

D. Squeeze and Excitation

We add Squeeze and Excitation [16] to the MobileNets so as to improve their performance in computer Go. The way we do it is by adding a Squeeze and Excitation block at the end of the MobileNet block before the addition.

The squeeze and excitation block starts with Global Average Pooling followed by two dense layers and a multiplication of the input tensor by the output of the dense layers.

We give here the Keras code we used for this block:

```
def SE_Block(t, filters, ratio=16):
    se_shape = (1, 1, filters)
    se = GlobalAveragePooling2D()(t)
    se = Reshape(se_shape)(se)
    se = Dense(filters // ratio,
               activation='relu',
               use_bias=False)(se)
    se = Dense(filters,
               activation='sigmoid',
               use_bias=False)(se)
    x = multiply([t,se])
    return x
```

```
def bottleneck_block_bis(x, expand=6*filters, squeeze=filters):
  m = layers.Conv2D(expand, (1,1), use_bias = False)(x)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.DepthwiseConv2D((3,3), padding='same', use_bias = False)(m)
  m = layers.BatchNormalization()(m)
  m = layers.Activation('relu')(m)
  m = layers.Conv2D(squeeze, (1,1), use_bias = False)(m)
  m = layers.BatchNormalization()(m)

  m=SE_block(m,filters)
  return layers.Add()([m, x]) #ici on rajoute l'entrée du bloc à l'entré
```

Ce modèle a constitué un progrès très important par rapport aux précédents dès les 100 premières époques.

Modèles	« Loss » du modèle	« Policy accuracy »	« Value_mse »
5	8.0368	0.3839	0.0883
4	8.400	0.3770	0.1005
3	8.5288	0.3598	0.0995
Shuffle Net + AvgGP	8.1329	0.3717	0.0897
Mobile Net	3,4934	0.3406	0,1183

Références des articules utilisés

[1] Residual Networks for Computer Go – march 2017 -Tristan Casenave – paragraphe II
[2] Spatial Average Pooling for Computer Go – June 2019 -Tristant Casenave – paragraphe 4
[3] Improving Model And Search For Computer Go – fev 2021 – Tristan Casenave – chapitre D Squeeze and Excitation
[4] Deep Learning python- Chollet François

Focus sur la « loss weight » et sur la « batch normalisation »

Fonctionnement des « loss weights »

"Optional list or dictionary specifying scalar coefficients (Python floats) to weight the loss contributions of different model outputs. The loss value that will be minimized by the model will then be the weighted sum of all individual losses, weighted by the loss_weights coefficients. "

Utilité de la batch "Normalisation"

'Batch normalization is a technique in deep learning used to normalize the activations of a layer within a batch of data. This helps to prevent the vanishing or exploding gradient problem and also speeds up the training process. By normalizing the activations, batch normalization helps to stabilise the distribution of the inputs to each layer, reducing the covariate shift and allowing the network to learn more effectively'.