Conception et entraînement de modèles de *deep learning* pour la prédiction de coups au jeu de Go

Aurélien Duvignac-Rosa, Edoardo Piciucchi

***Abstract*—Ce travail explore différentes architectures de réseaux de neurones appliquées à la prédiction de coups dans le jeu de Go. Cinq familles de modèles ont été retenues pour leur diversité structurelle : ResNet, MobileNet, ConvNeXt, ShuffleNet et une architecture personnalisée à base de convolutions en croix. L'évaluation comparative repose sur un entraînement limité à 50 époques, sous contrainte de 100 000 paramètres par modèle. L'architecture ShuffleNet composée de 4 blocs consécutifs s'est démarqué avec une précision finale de 27.85%, atteignant 30.74% après 100 époques. Ces résultats révèlent toutefois des performances globalement modestes, limitées par la contrainte du nombre de paramètres de 100 000 et le choix méthodologique centré uniquement sur la profondeur.**

Au-delà des aspects expérimentaux, l'ensemble du *pipeline* a été conçu de manière modulaire selon les principes de la programmation orientée objet. Les mécanismes d'abstraction, d'héritage et d'encapsulation assurent une extensibilité du code, facilitant l'intégration de nouvelles variantes architecturales. Ce cadre offre ainsi une base solide pour de futures améliorations, notamment via des approches multi-têtes, des mécanismes d'attention ou l'intégration du contexte stratégique du jeu.

***Keywords*—Réseaux de neurones de convolution ; ResNet ; MobileNet ; ConvNeXt ;ShuffleNet ; Convolutions en croix.**

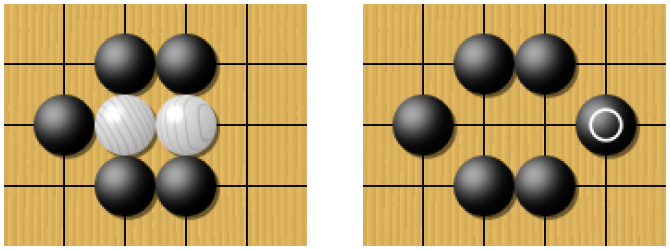
I. Introduction

Le jeu de Go est un jeu de stratégie millénaire originaire d'Asie de l'est, réputé pour la simplicité de ses règles mais la complexité extrême de ses dynamiques. Sa profondeur stratégique et son espace de recherche immensément vaste en font un terrain d'expérimentation privilégié pour l'intelligence artificielle. Si les premières avancées dans le domaine du *Computer Go* restaient limitées, La combinaison des réseaux neuronaux profonds et de la recherche arborescente de Monte Carlo a marqué une rupture technologique dans le domaine de l'intelligence artificielle appliquée aux jeux stratégiques, comme en témoignent des systèmes emblématiques tels qu'*AlphaGo* ou *KataGo*. Ces modèles dépassent désormais les meilleurs joueurs humains, en explorant un volume de positions considérablement supérieur à ce que peut envisager un joueur professionnel.

Dans tous les jeux à information parfaite et à espace discret, **il existe une politique qui, à partir de l'état actuel du jeu, fournit le coup optimal pour le joueur en cours, menant à l'utilité maximale sous hypothèse de jeu parfait**. Bien que les réseaux neuronaux soient théoriquement capables d'approximer toute fonction[1], y compris une telle politique, y parvenir en pratique demeure une tâche quasiment impossible.

Le jeu de plateau ancestral qu'est le Go constitue un exemple emblématique de jeu à information parfaite[2]. Il fait l'objet d'études en informatique depuis de nombreuses années et reste l'un des jeux à tours alternés les plus complexes en matière de performance algorithmique.

Une partie de Go oppose deux joueurs qui placent alternativement des pierres sur un plateau de 19 cases par 19, le joueur jouant les pierres noires commençant la partie. Toutes les pierres directement adjacentes les unes aux autres appartiennent au même groupe. Si un groupe est entièrement entouré par les pierres de l'adversaire, il est retiré du plateau, comme illustré à la Figure 1. L'objectif du jeu est de contrôler le plus de territoire possible en encerclant des zones du plateau.



(a) Les deux pierres blanches peuvent être capturées, car elles ne disposent que d'un seul point libre adjacent.

(b) Les deux pierres ont été capturées en jouant sur le point encerclé.

Figure 1. Exemple illustrant une séquence de capture. Les pierres blanches sont encerclées et retirées du plateau.

Dans ce contexte, le présent projet vise à concevoir, implémenter et évaluer plusieurs architectures de réseaux de neurones de convolution dans le cadre de la modélisation du jeu de Go. Chaque modèle étudié est contraint à un plafond strict de **100 000 paramètres**, assurant un cadre équitable et réaliste en termes de ressources computationnelles.

[1] cf. Théorème d'approximation universelle

[2] Un jeu est dit à information parfaite lorsque, à chaque étape, tous les joueurs connaissent l'ensemble des actions précédemment jouées et l'état complet du jeu.

L'entraînement repose sur un ensemble de **1 million de parties auto-jouées** par le moteur Katago. Pour chaque position, l'entrée du réseau est constituée de **31 plans** de taille 19×19, codant l'état du plateau et son historique immédiat. L'apprentissage supervisé est réalisé sur deux cibles distinctes :

- **La politique** : une distribution de probabilité sur les 361 coups possibles, représentant le coup joué dans la partie.
- **La valeur** : une estimation probabiliste de l'issue du match (victoire ou défaite de Blanc) issue de la recherche Monte-Carlo.

Plusieurs architectures seront étudiées, chacune présentant des caractéristiques spécifiques en matière de profondeur, de structure et de complexité :

- **ResNet** : réseaux résiduels profonds exploitant des connexions de saut (*skip connections*) pour faciliter l'apprentissage dans les architectures profondes ;
- **MobileNet** : modèles légers basés sur des convolutions séparables en profondeur, optimisés pour les environnements contraints ;
- **ConvNeXt** : architectures modernes combinant les avantages des réseaux de neurones de convolution classiques avec des éléments inspirés des *Transformers* ;
- **ShuffleNet** : modèles à faible coût computationnel, s'appuyant sur les convolutions groupées et la réorganisation des canaux (*channel shuffle*) ;
- **Convolutions en croix** : modèles basés sur des couches convolutives spécialisées, construites à partir de filtres en croix.

Ce cadre expérimental permettra d'évaluer l'efficacité de différentes architectures compactes pour un jeu à forte complexité stratégique. L'étude met l'accent sur l'impact de la structure et de la profondeur des réseaux dans un contexte contraint en paramètres.

II. Justification du protocole expérimental

Dans le cadre de cette étude, **le choix a été fait de faire varier exclusivement la profondeur des architectures (nombre de blocs) tout en maintenant constantes les autres hyperparamètres, notamment le nombre de filtres.**

Cette stratégie s'inscrit dans une logique rigoureuse, à la croisée de contraintes pratiques et d'objectifs analytiques :

- **Objectif ciblé :** l'étude vise à évaluer l'impact direct de la profondeur sur la capacité d'un modèle à apprendre les régularités d'un jeu de stratégie complexe comme le Go, dans un cadre contraint en ressources ;
- **Contrainte stricte de complexité :** tous les modèles sont limités à un maximum de 100 000 paramètres, ce qui exclut toute montée en capacité par simple élargissement du réseau ;
- **Contrôle expérimental :** le maintien constant du nombre de filtres permet d'isoler l'effet de la profondeur sur la performance, autorisant ainsi une analyse plus fine de son influence ;
- **Approche progressive :** en explorant un grand nombre de profondeurs différentes (jusqu'à 28 blocs pour MobileNet), le travail permet de cartographier la relation entre profondeur et performance dans un budget de complexité fixe ;
- **Justification temporelle :** compte tenu des contraintes de temps inhérentes au projet, un balayage systématique de la profondeur a été privilégié à une exploration conjointe profondeur-largeur, qui aurait exigé davantage de ressources.

Le choix de ne faire évoluer que la profondeur des réseaux dans une enveloppe de paramètres limitée à 100 000 permet d'évaluer précisément la complexité verticale des architectures tout en respectant un budget computationnel réaliste. Ce cadre expérimental assure une analyse structurée, reproductible et pertinente dans le contexte du projet.

III. Détails d'implémentation

La première étape a consisté à implémenter une classe abstraite GONet[3], conçue comme un modèle générique pour la construction de réseaux de neurones capables de prédire les coups joués (politique) ainsi que la probabilité de victoire (valeur). Cette classe fournit un cadre modulaire, réutilisable et traçable, facilitant l'expérimentation de différentes architectures convolutives (ResNet, MobileNet, ConvNeXt, ShuffleNet et *Cross Convolution*). Elle garantit que toutes les variantes partagent une base commune, tant sur le plan technique (entrée, sortie, entraînement) que méthodologique (nombre de paramètres, métriques, etc.).

La classe regroupe l'ensemble des méthodes communes aux architectures étudiées : génération des données d'entrée et des cibles simulées, définition des couches d'entrée du modèle, création des têtes de sortie (politique et valeur), compilation, entraînement, évaluation, traçage des métriques et sauvegarde. L'enchaînement complet de ces étapes est centralisé dans une méthode unique : `workflow()`.

Pour garantir une conception claire et modulaire, l'approche repose sur les principes fondamentaux de la programmation orientée objet :

- **Abstraction**: via la méthode abstraite `set_backbone()`, qui permet à chaque classe dérivée de définir sa propre architecture du réseau ;
- **Héritage**: les architectures spécifiques (ResNet, MobileNet, ConvNeXt, ShuffleNet et *Cross Convolution*) sont définies dans des classes filles héritant de GONet, qui tirent parti des fonctionnalités communes tout en personnalisant la structure inhérente à chaque architecture ;

[3]Le notebook d'entraînement est disponible à l'adresse suivante : `train_go_ai.ipynb` sur le dépôt GitHub.

- **Encapsulation** : les différentes étapes de préparation, de modélisation et d'apprentissage sont regroupées dans des méthodes internes cohérentes, activées via un point d'entrée unique : `workflow()`.

L'objectif principal de cette classe, décrite dans le code source 1, est de standardiser l'entraînement, l'évaluation et la comparaison des modèles, dans le respect strict des contraintes du projet :

- Un plafond de 100 000 paramètres par architecture ;
- Deux sorties fixes : politique (*softmax* sur 361 coups) et valeur (probabilité sigmoïde) ;
- Un format d'entrée unique : 31 plans de taille 19×19 représentant l'état du plateau ;
- Données d'entraînement générées automatiquement à partir de parties simulées.

```python
class GONet(ABC):
    # Methodes utilitaires
    # (initialisation et preparation des donnees)
    def __init__(...)
    def set_end(...)
    def set_groups(...)
    def set_input_data(...)
    def set_input_layer(...)
    def set_policy(...)
    def set_value(...)

    # Methodes liees au modele
    def create_policy_value_heads(...)
    def plot_model(...)
    def set_backbone(...) # methode abstraite
    def set_model(...)

    # Methodes : entrainement et evaluation
    def log_accuracy(...)
    def plot_training_history(...)
    def train_model(...)
    def workflow(...)

    # Methode de sauvegarde
    def save_model(...)
```

Code source 1. Initialisation de la classe GONet.

IV. Éléments de précision sur l'architecture commune des réseaux

Dans le cadre de ce projet, l'ensemble des modèles explorés partagent une structure logicielle et fonctionnelle commune. Chaque architecture repose sur un tronc convolutif, désigné sous le nom de *backbone*, qui est la seule partie variable d'un réseau à l'autre. Les têtes de sortie de prédiction, responsables de la politique et de la valeur, sont, quant à elles, fixes pour garantir une comparaison cohérente et équitable.

A. Structure des têtes de sortie

Comme illustré sur la figure 2, le réseau se ramifie en deux branches principales à partir de la sortie du *backbone*, chacune correspondant à une tâche d'apprentissage supervisé : la politique (*policy*) et la valeur (*value*)[4].

- **La valeur** : estime la probabilité de victoire du joueur courant à partir de la position considérée ;
- **La politique** : prédit la probabilité de jouer chacun des 361 coups possibles sur le plateau.

Ces deux branches utilisent des mécanismes architecturaux distincts mais compacts, compatibles avec la contrainte stricte de **100 000 paramètres** au total.

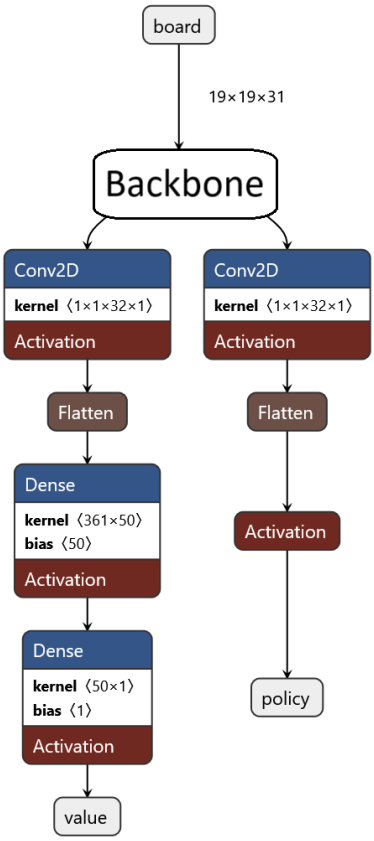


Figure 2. Architecture des têtes de sortie partagées par tous les modèles. Le *backbone* est la seule partie personnalisable.

1) Tête de valeur: La tête de valeur (cf. code source 2) permet d'extraire une estimation probabiliste plus informative. Elle utilise :

- Une couche *Conv2D* 1×1 (sans biais) avec activation ReLU ;
- Un *Flatten* pour vectoriser la sortie spatiale ;
- Une couche *Dense(50)* avec ReLU, jouant le rôle de couche intermédiaire ;
- Une couche *Dense(1)* avec une activation *sigmoïde* pour produire une probabilité entre 0 et 1.

Ce chemin est bien adapté à une tâche de type binaire (victoire/défaite), tout en restant compatible avec les contraintes paramétriques du projet.

[4]les définitions de politique et de valeurs sont rappelées dans la section I

```
def create\_policy\_value\_heads(...):
    ...
    # En-tete de valeur
    value\_head = layers.Conv2D(
        1,
        1,
        activation="relu",
        padding="same",
        use\_bias=False,
        kernel\_regularizer=regularizers.l2(0.0001),
    )(self.x)
    value\_head = layers.Flatten()(value\_head)
    value\_head = layers.Dense(
        50,
        activation="relu",
        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=self.input_layer,
        outputs=[policy\_head, value\_head],
        name=self.name,
    )
    ...
```

Code source 2. Implémentation de la tête de valeur dans la classe GONet.

2) Tête de politique: La tête de politique (cf. code source 3) suit un schéma similaire, mais adopte un chemin moins profond. Elle commence par une couche *Conv2D* de noyau 1×1 (sans biais), qui projette les canaux de sortie du *backbone* vers un unique canal. Cela revient à produire une seule carte de caractéristiques, représentant une distribution spatiale des coups jouables sur le plateau. Cette opération est suivie d'un *Flatten* qui transforme la sortie 19×19 en un vecteur de 361 dimensions. Une activation *softmax* est ensuite appliquée pour générer une distribution de probabilité sur les coups possibles. Ce bloc réalise une transformation minimale, tout en capturant les signaux spatiaux finaux issus du *backbone*.

```
def create\_policy\_value\_heads(...):
    ...
    # En-tete de politique
    policy\_head = layers.Conv2D(
        1,
        1,
        activation="relu",
        padding="same",
        use\_bias=False,
        kernel\_regularizer=regularizers.l2(0.0001),
    )(self.x)
    policy\_head = layers.Flatten()(policy\_head)
    policy\_head = layers.Activation("softmax", name="
        policy")(
        policy\_head
    )
    ...
```

Code source 3. Implémentation de la tête de politique dans la classe GONet.

B. Compilation et vérification du modèle

Le modèle final est construit à partir de la couche d'entrée *Input(shape=(19, 19, 31))* et des deux sorties *policy* et *value*. Il est compilé avec l'optimiseur *SGD* (taux d'apprentissage 0.0005, *momentum* 0.9). Deux fonctions de perte sont utilisées :

- *categorical_crossentropy* pour la politique ;
- *binary_crossentropy* pour la valeur.

Chaque sortie est également associée à une métrique :

- *categorical_accuracy* pour la politique ;
- *mean squared error (MSE)* pour la valeur.

Cette configuration est visible dans le bloc de compilation du modèle, où la perte de politique est spécifiée comme suit :

```
model.compile(
    ...
    loss={
        "policy": "categorical\_crossentropy",
        "value": "binary\_crossentropy",
    },
    ...
)
```

Code source 4. Définition de perte de politique et de valeur dans la méthode `create_policy_value_heads` de la classe mère GoNet.

(cf. fonction `create_policy_value_heads`, ligne de compilation du modèle).

Une vérification automatique du nombre de paramètres garantit que toutes les architectures respectent la limite de 100 000, indépendamment du *backbone* utilisé. En fixant une structure de sortie identique, ce mécanisme assure une comparaison équitable des capacités des différentes implémentation du *backbone*.

V. Réseau de référence fourni dans l'énoncé

Avant de nous pencher sur l'étude comparative des architectures telles que ResNet, MobileNet, ConvNeXt, ShuffleNet et *Cross Convolution*, il est essentiel d'analyser le réseau minimal fourni dans l'énoncé, implémenté dans la classe GONetDemo. Ce modèle de base constitue un point d'ancrage indispensable pour la suite de notre travail puisqu'il permet de valider l'ensemble du *pipeline* d'apprentissage supervisé, de tester la structure logicielle commune à tous les modèles, et d'obtenir une première mesure de performance servant de référence.

A. Description de l'architecture

Le réseau *GONetDemo*, dont l'implémentation est décrite dans le code source 5, repose sur une pile de couches de convolution simples, conçues pour être faciles à entraîner tout en respectant les contraintes du projet. Ici, l'objectif n'est pas d'optimiser la performance, mais de proposer une structure rapide à entraîner pour valider l'implémentation de la classe mère *GONet* et des méthodes associées.

Ce modèle remplit plusieurs objectifs pédagogiques et techniques :

- Valider le *pipeline* d'entraînement et d'évaluation dans un contexte simple ;
- Offrir un point de comparaison neutre pour mesurer les gains apportés par les architectures avancées ;
- Fixer une première borne inférieure de performance accessible sans complexité excessive.

```python
class GONetDemo(GONet):

    def __init__(self):
        ...

    def set_backbone(self):
        filters = 32
        x = layers.Conv2D(filters, 1, activation="relu",
             padding="same")(
            self.input_layer
        )
        for _ in range(5):
            x = layers.Conv2D(filters, 3, activation="relu"
                 , padding="same")(x)
        self.x = x
```

Code source 5. Définition du backbone dans GONetDemo

La figure 3 présente les composantes de cette architecture[5]

Il s'agit d'un réseau entièrement convolutionnel, **sans opération explicite de *downsampling***. Aucun *MaxPooling* ou *stride* > 1 n'est utilisé, ce qui permet de conserver une résolution spatiale maximale, un aspect potentiellement avantageux pour un jeu comme le Go, où la localisation exacte des pierres est cruciale.

Pour rappel, en aval des convolutions, le réseau applique une couche *Flatten*, puis deux têtes de sortie séparées : une couche *Dense(361)* avec activation *softmax* pour la politique, et une *Dense(1)* avec activation *sigmoïde* pour la valeur. Ce schéma est identique pour toutes les variantes testées dans le projet, assurant une base commune pour la comparaison.

[5]L'architecture des têtes de sortie partagées par tous les modèles est décrite dans IV.

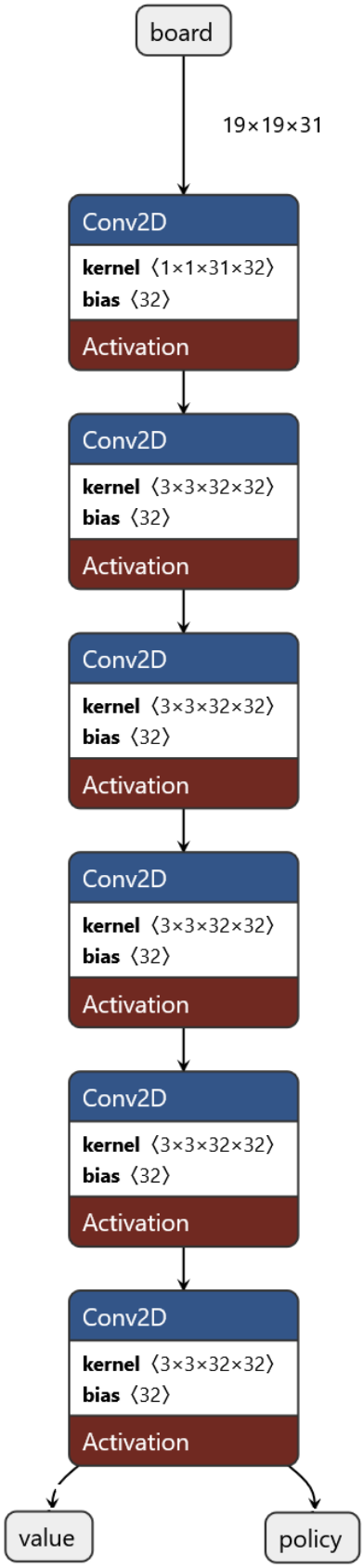
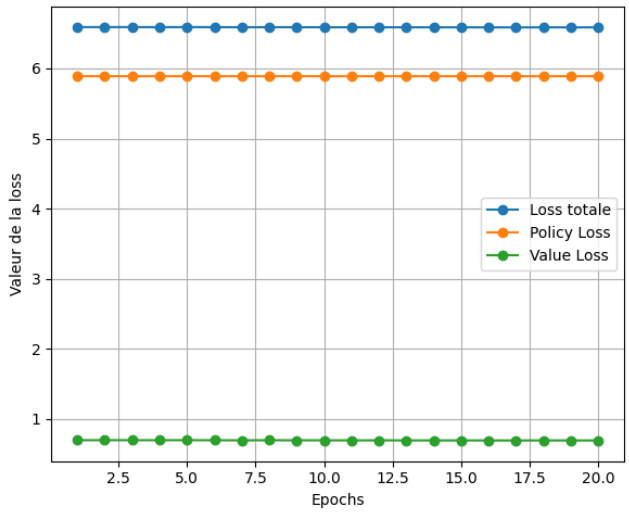


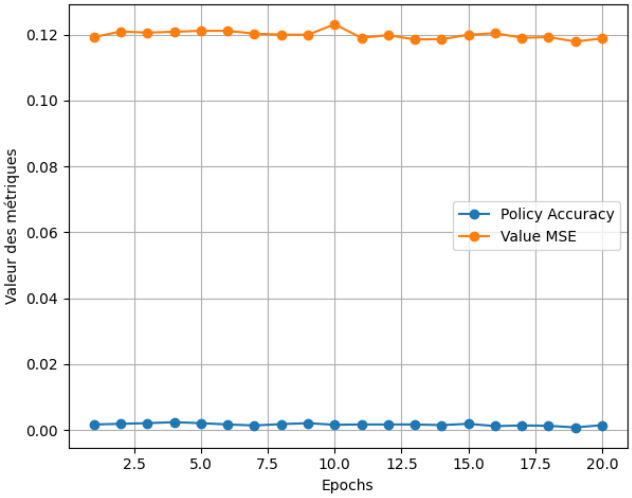
Figure 3. *backbone* du réseau de référence : GONetDemo, fourni dans l'énoncé.

B. Analyse des résultats obtenus

L'analyse des courbes d'apprentissage du modèle GONetDemo, présentées sur la figure 4 et synthétisées dans le tableau 1, permet de tirer plusieurs conclusions importantes sur ses capacités d'apprentissage.



(a) Évolution des pertes d'entraînement et de validation pour GONetDemo.



(b) Évolution des métriques (précision catégorielle et MSE) pour GONetDemo.

Figure 4. Courbes d'apprentissage du modèle GONetDemo.

La figure 4a rend compte de l'évolution globale de la perte au cours de l'entraînement. Cette courbe combine les contributions des deux têtes du réseau : la tête de politique et la tête de valeur. L'absence de variation des valeurs de perte indique que le modèle ne parvient pas à optimiser ses paramètres au cours de l'entraînement.

Du côté de la tête de valeur, la perte correspondante est mesurée à l'aide de l'erreur quadratique moyenne (*Mean Squared Error* abrégée en *MSE*). Comme le montre également la figure 4a, la valeur de la perte reste constante autour de 0.07 tout au long de l'entraînement. Ce comportement suggère que la capacité du modèle à estimer correctement la probabilité de victoire n'a pas progressé.

La tête de politique, quant à elle, produit une distribution de probabilité sur l'ensemble des coups possibles, via une activation *softmax*. Pour rappel, la perte associée est définie dans le code comme une entropie croisée catégorielle (*categorical crossentropy*), qui mesure l'écart entre cette distribution prédite et la cible issue des données d'entraînement (cf. code source 4).

Comme l'illustre la figure 4a, cette *loss* ne diminue pas durant l'entraînement, ce qui reflète un apprentissage nul de la composante afférente à la politique.

Cette stagnation est également visible dans les métriques associées (cf. figure 4b). La *MSE* de la tête de valeur reste stable autour de 0.12, confirmant l'absence d'amélioration notable. Quant à la précision catégorielle de la politique, elle reste extrêmement faible : 0.007, tout au long de l'entraînement. Ce niveau très bas indique que le réseau peine à prédire correctement les coups joués dans les données d'entraînement, et souligne une difficulté majeure à apprendre une politique pertinente dans cette configuration.

La tableau ci-dessous résume les performance du modèle GONetDemo.

Tableau 1
RÉSUMÉ DES PERFORMANCES DU MODÈLE GONETDEMO.

Élément analysé	Observation principale
Loss totale	Constante, absence d'optimisation
Policy loss	Stable, aucun apprentissage apparent
Value loss	Inchangée, ∼0.7
Policy accuracy	Très faible, aucune amélioration
Value MSE	Stable (∼0.12), peu d'évolution

Ces résultats indiquent clairement que l'architecture GONetDemo, bien qu'utile comme point de départ technique, est insuffisante pour prédire les coups au jeu de Go de manière satisfaisante. Justifiant ainsi l'exploration d'architectures plus profondes et spécialisées dans les sections suivantes.

VI. RÉSEAUX DE NEURONES RÉSIDUELS (RESNET)

Introduite pour remédier au problème d'évanescence du gradient dans les réseaux profonds, l'architecture des réseaux de neurones résiduels repose sur l'ajout de connexions résiduelles, qui facilitent considérablement l'apprentissage de modèles plus profonds et expressifs : une propriété particulièrement précieuse dans le contexte stratégique du jeu de Go.

A. Principes généraux

Les réseaux résiduels (*Residual Networks*, ou ResNet) tirent leur nom du modèle ayant remporté la compétition ILSVRC en 2015. Leur spécificité tient à l'introduction de connexions saute-couches (*skip connections*), permettant

au flux d'information (et au gradient) de contourner certaines couches du réseau. Ces connexions ajoutent directement l'entrée d'un bloc à sa sortie, facilitant ainsi l'optimisation de réseaux très profonds en atténuant la dégradation des gradients.

Comme le souligne la littérature [1], l'utilisation de ResNet pour le Go permet :

- un apprentissage plus rapide grâce à une rétropropagation efficace ;
- une meilleure stabilité d'entraînement, même pour des réseaux profonds ;
- une expressivité accrue dans la modélisation des stratégies complexes.

Le choix d'une architecture résiduelle s'inscrit dans la volonté d'explorer des modèles profonds tout en maintenant la stabilité de l'apprentissage. Dans le contexte du jeu de Go, où l'identification de motifs multi-échelles est cruciale, ces connexions saute-couches permettent de préserver les représentations de bas niveau tout en construisant des abstractions plus complexes à travers la profondeur.

L'implémentation retenue dans ce projet est directement inspirée du travail de [1], avec des adaptations portant à la fois sur la structure modulaire imposée par la classe GONet et sur les dimensions des noyaux de convolution. Alors que l'article original repose sur une combinaison de convolutions 5×5 et 1×1, les blocs résiduels ont ici été conçus avec deux convolutions 3×3, dans le but de réduire le nombre de paramètres tout en préservant une capacité d'extraction de motifs locaux.

B. Description de l'architecture

Le modèle implémenté, intitulé : GONetResNet, s'appuie sur un empilement de blocs résiduels, dont l'implémentation est décrite dans le code source 6, chacun intégrant une transformation non linéaire combinée à une connexion directe entre l'entrée et la sortie.

```
def residual_block(self, x):
    shortcut = x
    x = layers.Conv2D(self.filters, 3, padding="same")(x)
    x = layers.ReLU()(x)
    x = layers.Conv2D(self.filters, 3, padding="same")(x)

    # Projection du shortcut si necessaire
    if shortcut.shape[-1] != x.shape[-1]:
        shortcut = layers.Conv2D(self.filters, 1, padding="
            same")(shortcut)

    x = layers.Add()([x, shortcut])
    return layers.ReLU()(x)
```

Code source 6. Bloc résiduel de l'architecture GONetResNet.

Chaque bloc résiduel applique deux convolutions 3×3 séquentielles, séparées par une activation ReLU, sur le flux principal. Ce résultat est ensuite additionné au *shortcut*, c'est-à-dire l'entrée originale, projetée via une convolution 1×1 si nécessaire (en cas de changement de profondeur). L'addition est suivie d'une activation ReLU.

La méthode `set_backbone()`, présentée sur la figure 5 et dans le code source 7, contrôle le nombre de blocs résiduels empilés. Par défaut, un seul bloc est utilisé, mais ce nombre peut être augmenté pour accroître la profondeur du réseau.

```
def set_backbone(self):
    x = self.input_layer
    for _ in range(self.num_blocks):
        x = self.residual_block(x)
    self.x = x
```

Code source 7. Définition du *backbone* dans GONetResNet.

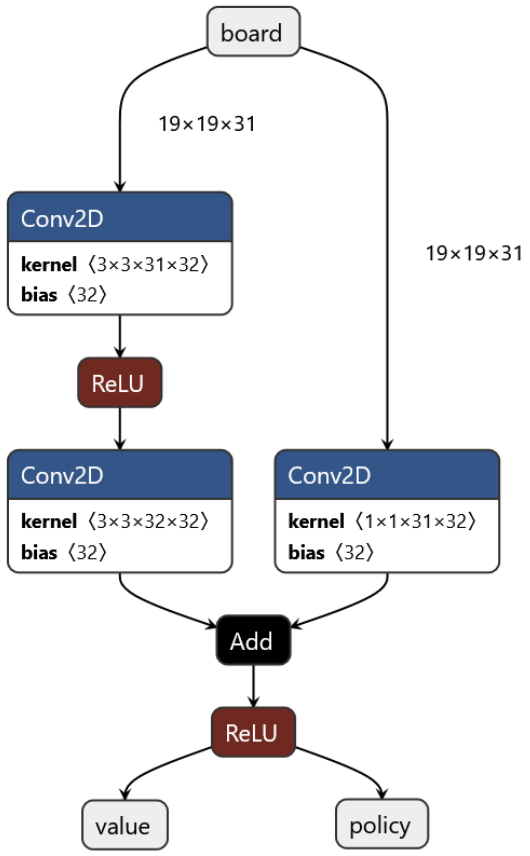


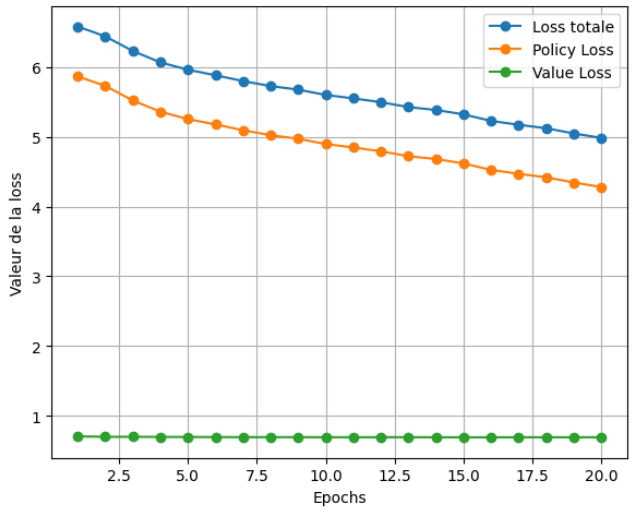
Figure 5. *backbone* du réseau : GONetResNet (avec un seul bloc).

Remarquons que cette structure capture l'esprit des architectures ResNet, tout en adoptant une forme simplifiée par rapport aux implémentations standards. Elle ne comporte ni normalisation de lots (*BatchNormalization*), ni structure de bloc *bottleneck*, mais introduit une convolution 1×1 spécifique lorsque c'est nécessaire afin d'assurer la compatibilité des dimensions. En particulier, l'entrée initiale du réseau possède 31 canaux, tandis que la première couche convolutive produit une sortie à 32 canaux : une opération de projection via une convolution $1 \times 1 \times 31 \times 32$ est nécessaire pour permettre l'addition résiduelle entre les deux tenseurs de dimensions respectives $19 \times 19 \times 32$ et $19 \times 19 \times 31$.

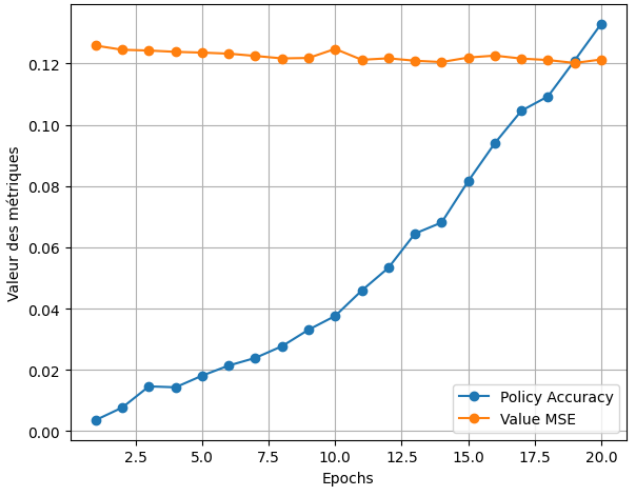
C. Analyse des résultats obtenus

Les performances du modèle GONetResNet, présentées dans les figures 6a et 6b, se révèlent significativement supérieures à celles observées avec l'architecture de référence GONetDemo.

Après 20 époques d'entraînement, la précision atteint près de 12,5%, marquant une amélioration notable par rapport au modèle de référence (cf. figure 4). La courbe de *loss* totale montre une diminution progressive, principalement portée par la baisse de la *loss* associée à la politique. Toutefois, ce résultat demeure contrasté : la *loss* de valeur et la métrique *MSE* restent quasi constantes, prenant respectivement pour valeur 0,5 et 0,12 respectivement, indiquant une stagnation de la tête de valeur.



(a) Évolution des pertes d'entraînement et de validation pour GONetResNet (1 bloc).



(b) Évolution des métriques (précision catégorielle et *MSE*) pour GONetResNet (1 bloc).

Figure 6. Courbes d'apprentissage du modèle `GONetResNet` avec un seul bloc résiduel.

Ces résultats restent cependant prometteurs, motivant ainsi l'analyse de l'évolution de l'*accuracy* en fonction de l'augmentation du nombre de blocs.

D. Analyse quantitative de l'augmentation du nombre de blocs résiduels sur l'efficacité du réseau

Les résultats prometteurs obtenus avec l'architecture GONetResNet ont motivé un approfondissement de l'étude en augmentant le nombre de blocs résiduels dans le réseau. Toutefois, afin de respecter la contrainte stricte de 100 000 paramètres, le nombre total de blocs ne peut excéder trois. Une analyse similaire à celle menée pour la version à un seul bloc a donc été menée pour les variantes comportant respectivement deux et trois blocs.

Pour mieux comprendre l'évolution de l'apprentissage, les figures 6 à 9 présentent les courbes détaillées de pertes et de métriques associées à ces trois variantes. Pour chaque profondeur (de 1 à 3 blocs), sont représentées la dynamique de la perte globale, celles des sous-objectifs (*policy* et *value*), ainsi que l'évolution des indicateurs de performance. La figure 7 illustre les valeurs d'*accuracy*) obtenues pour les modèles GONetResNet selon le nombre de blocs résiduels utilisés.

L'architecture à trois blocs atteint la meilleure précision avec une valeur de 21,06 %, suivie de très près par celle à deux blocs avec une précision de 20,60%. En comparaison, le modèle à un seul bloc reste nettement en retrait, plafonnant à 13,30%.

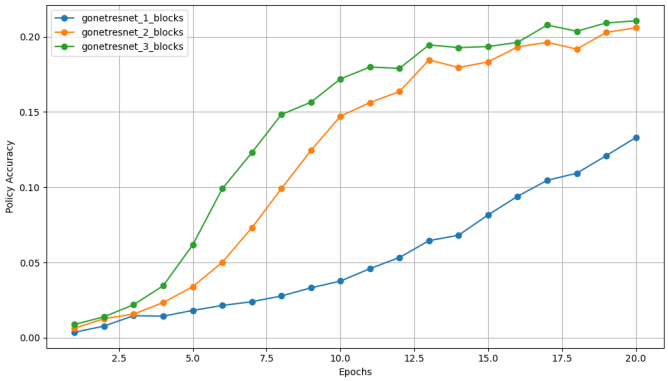
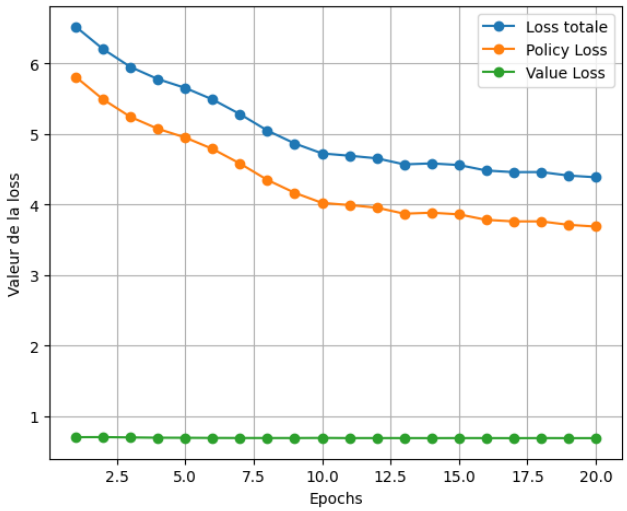
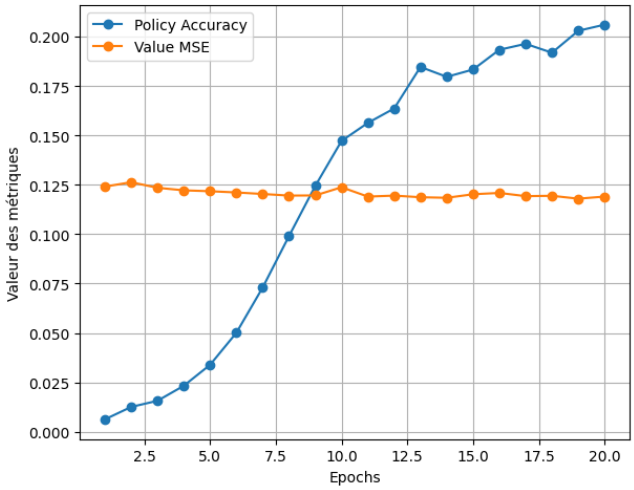


Figure 7. Comparaison des *accuracies* selon le nombre de bloc dans les réseaux `GONetResNet`.

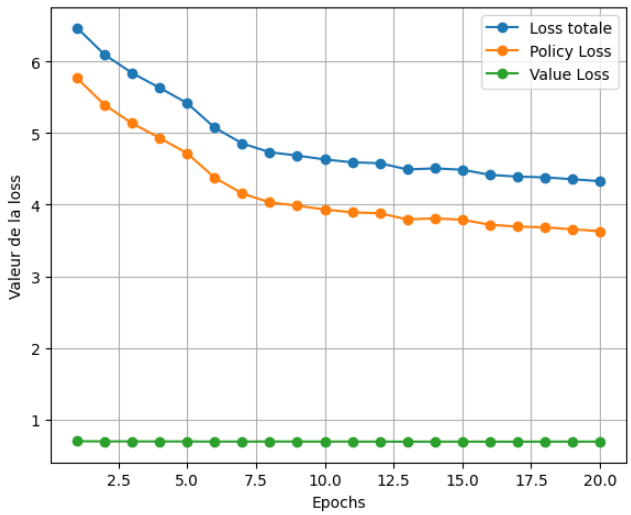


(a) Évolution des pertes d'entraînement et de valeur pour l'architecture GONetResNet (2 blocs).

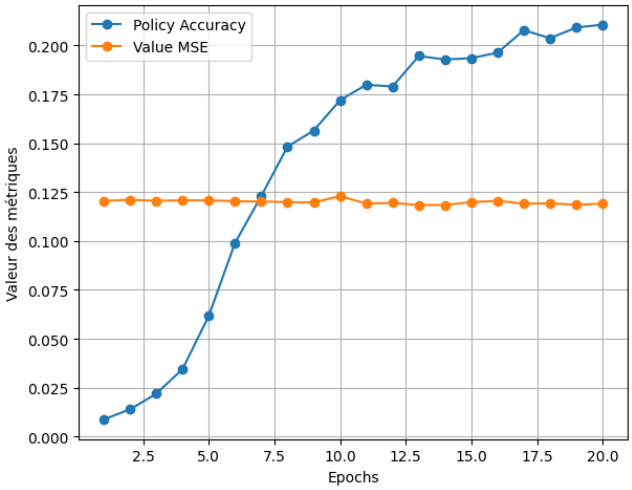


(b) Évolution des métriques (précision et *MSE*) pour l'architecture GONetResNet (2 blocs).

Figure 8. Courbes d'apprentissage pour le modèle GONetResNet avec 2 blocs résiduels.



(a) Évolution des pertes d'entraînement et de valeur pour l'architecture GONetResNet (3 blocs).



(b) Évolution des métriques (précision et *MSE*) pour l'architecture GONetResNet (3 blocs).

Figure 9. Courbes d'apprentissage pour la modèle GONetResNet avec 3 blocs résiduels.

Cette étude a mis en exergue que l'ajout de blocs résiduels améliore significativement la performance des réseaux de neurones convolutifs, jusqu'à un certain point. Toutefois, cette amélioration s'accompagne d'une augmentation du nombre de paramètres, du temps d'entraînement et de la consommation mémoire.

Dans une optique d'optimisation du rapport performance et d'efficacité, il devient pertinent de se tourner vers des architectures plus légères et efficaces.

C'est dans ce contexte que s'inscrit MobileNet, une architecture conçue pour offrir une bonne précision tout en réduisant fortement la complexité computationnelle, grâce à l'utilisation de convolutions séparables en profondeur.

La prochaine étape consiste à étudier cette architecture et la comparer avec les architectures étudiées jusqu'à présent.

VII. Réseaux de neurones type MobileNet

A. Principes généraux

Les réseaux MobileNet sont couramment utilisés en vision par ordinateur pour classer des images. Ils atteignent une haute précision sur les jeux de données standards tout en conservant un nombre de paramètres inférieur à celui d'autres architectures de réseaux neuronaux.

L'implémentation s'appuie directement sur les éléments proposés dans [2].

B. Description de l'architecture

L'architecture GONetMobileNet s'inspire du principe des réseaux MobileNetV1, conçus pour être légers, efficaces, et particulièrement adaptés aux environnements contraints en ressources. Elle repose sur l'utilisation de **convolutions séparables en profondeur**, une technique qui décompose la convolution standard en deux opérations distinctes :

- une convolution *depthwise*, appliquée indépendamment à chaque canal d'entrée ;
- suivie d'une convolution *pointwise*, qui agrège l'information entre canaux.

Cette stratégie permet de réduire significativement le nombre de paramètres et le coût de calcul tout en conservant une capacité d'extraction de caractéristiques satisfaisante. Dans l'implémentation proposée, chaque bloc (*bottleneck*) suit une structure typique :

1) une première convolution 1×1 (appelée *expansion*) pour augmenter la dimension des canaux (*filters*) ;
2) une convolution séparée en profondeur 3×3 (*DepthwiseConv2D*) pour extraire les motifs spatiaux ;
3) une seconde convolution 1×1 (*projection*) ramenant les canaux à leur nombre initial (*trunk*) ;
4) une *BatchNormalization* suivie d'une activation ReLU après chaque opération convolutive ;
5) un ajout résiduel avec l'entrée du bloc (*Add*), à condition que les dimensions correspondent.

Le *backbone*, présenté dans le code source 9 et la figure 10, débute par une couche de convolution 1×1 appliquée à l'entrée, suivie d'une normalisation et d'une activation, puis empile *num_blocks bottlenecks* (cf. code source 8).

```
def bottleneck_block(self, x):
    m = layers.Conv2D(
        self.filters,
        (1, 1),
        kernel_regularizer=regularizers.l2(0.0001),
        use_bias=False,
    )(x)
    m = layers.BatchNormalization()(m)
    m = layers.Activation("relu")(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("relu")(m)

    m = layers.Conv2D(
        self.trunk,
        (1, 1),
        kernel_regularizer=regularizers.l2(0.0001),
        use_bias=False,
    )(m)
    m = layers.BatchNormalization()(m)

    return layers.Add()([m, x])
```

Code source 8. Méthode `bottleneck_block()` du modèle GONetMobileNet.

```
def set_backbone(self):
    x = self.input_layer

    x = layers.Conv2D(
        self.trunk,
        1,
        padding="same",
        kernel_regularizer=regularizers.l2(0.0001),
    )(x)
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)

    for _ in range(self.num_blocks):
        x = self.bottleneck_block(x)

    self.x = x
```

Code source 9. Méthode `set_backbone()` du modèle `GONetMobileNet`.

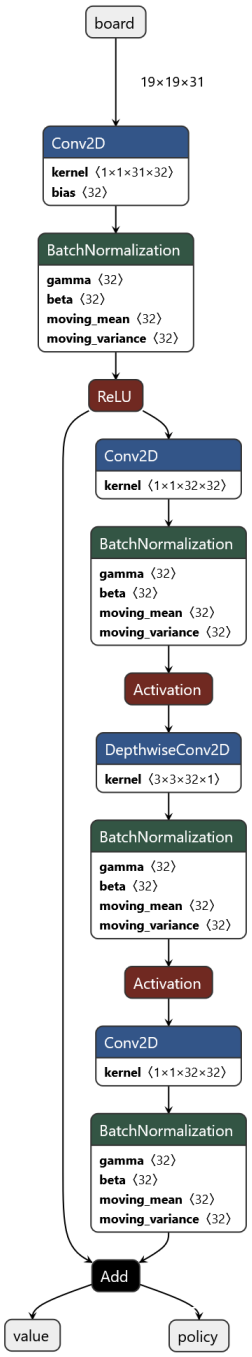
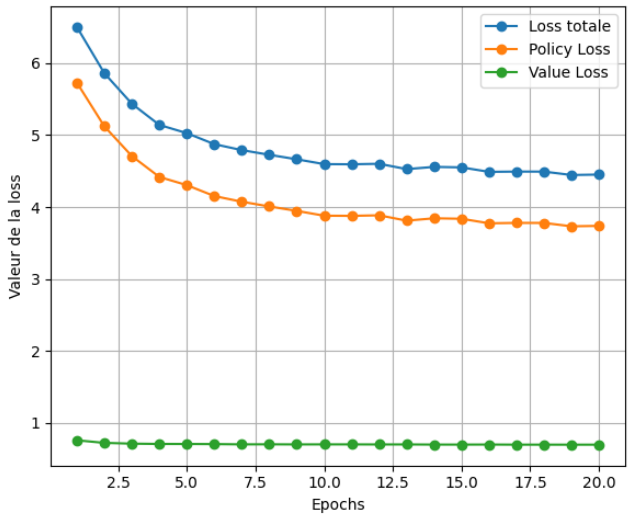


Figure 10. backbone du réseau `GONetMobileNet` (avec un seul bloc).

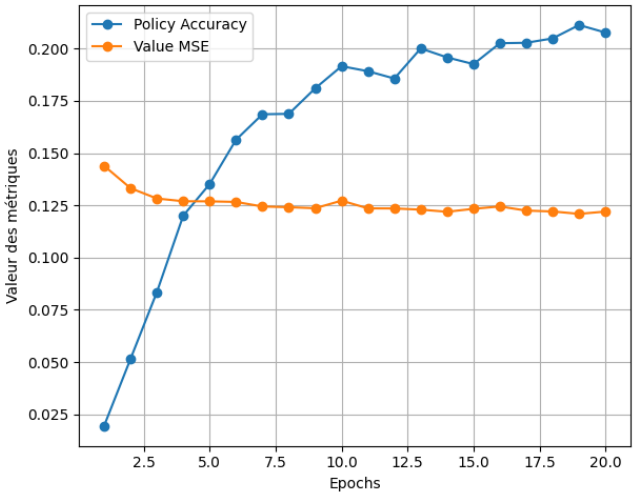
C. Analyse des résultats obtenus

Dans une démarche analogue à celle réalisée pour les structures de réseaux précédents, la première étape a consisté à étudier les performances de cette architecture avec une seule couche.

La figure 11 présente l'évolution des pertes, de la *MSE* ainsi que de l'*accuracy* pour l'architecture décrite sur la figure 10.



(a) Évolution des pertes d'entraînement et de valeur pour GONetMobileNet (1 bloc).



(b) Évolution des métriques (précision catégorielle et MSE) pour GONetMobileNet (1 bloc).

Figure 11. Courbes d'apprentissage pour le modèle `GONetMobileNet` avec 1 bloc convolutif.

Les résultats obtenus mettent en évidence une dynamique d'entraînement contrastée selon les têtes de sortie. En effet, la courbe de *loss* totale diminue significativement au cours des premières époques, passant d'environ 6,5 à 4,5, avant de se stabiliser. Cette tendance est essentiellement due à la baisse de la *policy loss*, qui chute de 5,8 à environ 3,7, traduisant une amélioration progressive

de la prédiction des coups. En revanche, la *value loss* reste quasi constante autour de 0,65, suggérant une difficulté du modèle à apprendre efficacement à évaluer les positions.

Quant à la *policy accuracy*, elle progresse régulièrement de 2% à plus de 21% en 20 époques, ce qui constitue un résultat encourageant pour une architecture aussi légère. À l'inverse, la métrique de régression associée à la tête de valeur (*Value MSE*) stagne autour de 0,125, indiquant une amélioration marginale dans la prédiction de l'issue des parties.

Ces résultats montrent que l'architecture MobileNet, bien que compacte, parvient à extraire des représentations pertinentes pour la tête de politique. En revanche, sa capacité à estimer la valeur d'une position reste limitée, probablement en raison de sa faible profondeur. Cette observation a motivé une analyse approfondie de l'influence de la profondeur du réseau. Il a d'abord été établi que, pour respecter la contrainte imposée de 100 000 paramètres, la profondeur maximale atteignable pour cette architecture était de 28 couches.

D. Analyse quantitative de l'augmentation du nombre de blocs résiduels sur l'efficacité du réseau

Les figures 12 à 16 synthétisent l'évolution des pertes totale, politique et de valeur, ainsi que des métriques *MSE* et précision catégorielle (*accuracy*) pour l'ensemble des 28 variantes d'architecture GONetMobileNet évaluées.

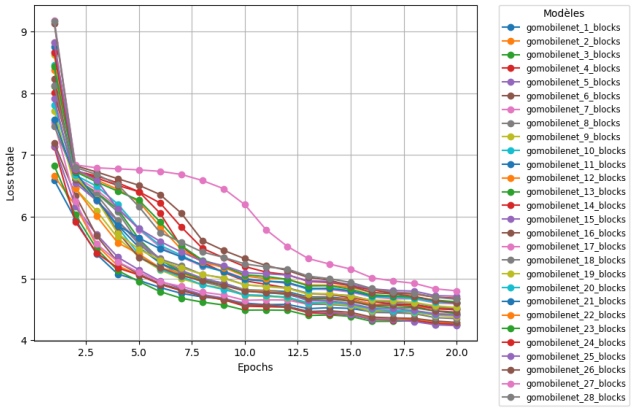


Figure 12. Comparaison de la perte totale (*total loss*) au cours de l'apprentissage pour les différentes profondeurs du modèle GONetMobileNet.

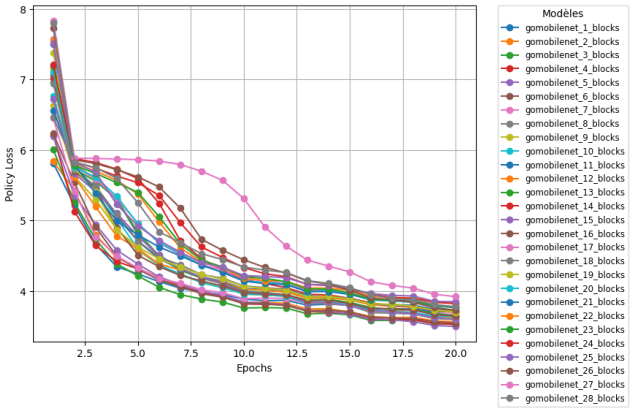


Figure 13. Comparaison de la perte associée à la tête de politique (*policy loss*) pour les différentes variantes de GONetMobileNet.

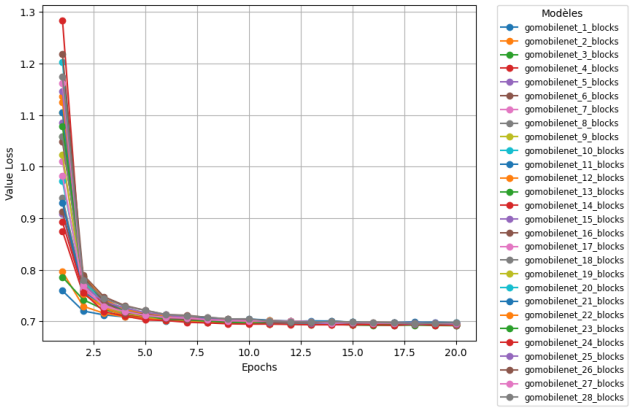


Figure 14. Comparaison de la perte associée à la tête de valeur (*value loss*) en fonction de la profondeur du modèle GONetMobileNet.

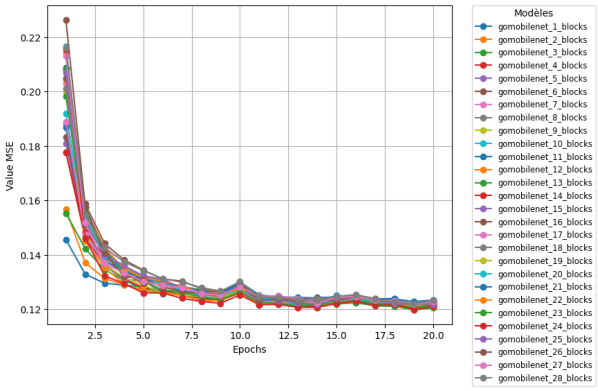


Figure 15. Comparaison de l'erreur quadratique moyenne (MSE) pour les différentes profondeurs de GONetMobileNet.

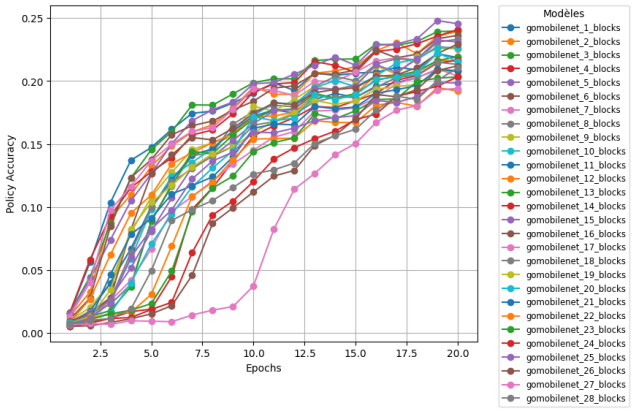


Figure 16. Évolution de la précision catégorielle (*accuracy*) selon le nombre de blocs utilisés dans l'architecture GONetMobileNet.

L'architecture contenant 5 blocs s'illustre comme la plus performante parmi les variantes testées, atteignant une précision catégorielle légèrement inférieure à 25%. Il s'agit là de l'un des meilleurs résultats obtenus jusqu'à présent dans le cadre de cette étude.

Par ailleurs, les valeurs des pertes, tant pour la tête de politique que pour celle de valeur, montrent une diminution significative au fil des époques. Cela indique que le modèle ne se contente pas d'extraire efficacement des motifs de jeu, mais qu'il est également capable d'estimer de manière plus fiable l'issue des parties. Cette double compétence, classification des coups et évaluation de position, marque une nette amélioration par rapport à une architecture résiduelle, qui peine à progresser sur la tête de valeur.

En synthèse, l'analyse des performances des différentes variantes de GONetMobileNet conduit aux observations suivantes :

- Les meilleures performances globales sont obtenues avec des architectures comprenant **entre 4 et 6 blocs**. En se basant uniquement sur l'*accuracy*, l'architecture la plus efficace est celle comportant **5 blocs** ;
- Une augmentation du nombre de blocs au-delà de ce seuil n'apporte aucun gain notable, ni sur les pertes ni sur les métriques associées. Elle tend même à dégrader les performances globales, suggérant un effet de surcapacité ou de saturation.

Après cette exploration approfondie de l'architecture MobileNet à différentes profondeurs, il est désormais pertinent d'élargir la comparaison à des modèles plus récents et expressifs.

Parmi ceux-ci, ConvNeXt s'impose comme une évolution notable des réseaux convolutifs classiques. S'inspirant des meilleures pratiques issues des *Transformers*, cette architecture revisite en profondeur la conception des réseaux de neurones de convolution pour les rendre plus compétitifs sur des tâches de vision à grande échelle.

Étudions à présent cette architecture dans le cadre de notre projet, en l'adaptant à la structure commune `GONet`, et en analysant ses performances comparativement aux réseaux précédemment évalués.

VIII. Réseaux de neurones type ConvNeXt

A. Principes généraux

Le terme ConvNeXt désigne une famille de modèles d'apprentissage profond spécifiquement conçus pour l'extraction de caractéristiques, en particulier sur des ensembles de données visuelles. Ces modèles s'inscrivent dans la lignée des réseaux de neurones de convolution, tout en intégrant des principes inspirés des *Transformers* afin de moderniser la conception des architectures convolutives.

Les modèles ConvNeXt reposent sur une succession de couches de convolution, suivies de couches entièrement connectées. Leur structure permet de capturer efficacement les motifs visuels présents dans les données, ce qui les rend particulièrement performants pour les tâches de classification ou de régression dans le domaine de la vision par ordinateur.

Comme les autres architectures profondes, ConvNeXt est entraîné sur de grands volumes de données à l'aide de la rétropropagation du gradient (*backpropagation*). Une fois l'apprentissage réalisé, le modèle peut être utilisé pour prédire de nouvelles données, extraire des représentations ou résoudre diverses tâches supervisées.

Enfin, bien que performants, les modèles ConvNeXt présentent également l'avantage d'être déclinables en versions compactes, adaptées aux dispositifs à faibles ressources, tels que les terminaux mobiles. Cela en fait une solution robuste et polyvalente pour l'apprentissage profond dans un cadre contraint.

B. Description de l'architecture

L'architecture GONetConvNeXtV1 repose sur une adaptation simplifiée des principes introduits par le modèle ConvNeXt [3], combinant profondeur, normalisation et convolutions séparables. L'implémentation proposée dans ce projet s'inspire directement des blocs utilisés dans le cadre du cours, avec quelques ajustements pour répondre à la contrainte de complexité maximale imposée (100 000 paramètres).

Le *backbone* du réseau, dont l'implémentation est décrite dans le code source 10 et schématisé sur la figure 17, est structuré comme suit :

- Une projection initiale par convolution 1×1 permet de passer des 31 canaux d'entrée à un nombre configurable de canaux internes (`filters`);
- Une séquence de blocs de type ConvNeXt est ensuite répété `num_blocks` fois. Chaque bloc applique une convolution en profondeur (`DepthwiseConv2D`) avec un noyau 7×7, suivie d'une normalisation de couche `LayerNormalization` et de deux convolutions 1×1 avec activation GELU entre les deux. Une addition résiduelle permet de stabiliser l'apprentissage, suivie d'une normalisation de lot (`BatchNormalization`) ;
- Enfin, un réduction de canaux ramène la sortie du tronc à 32 filtres, assurant une compatibilité avec les têtes de prédiction et de valeur.

Le code source 10 et la figre 17 présentent respectivement l'implémentation et l'architecture associée.

```python
def set_backbone(self):
    x = self.input_layer

    # Projection initiale
    x = layers.Conv2D(self.filters, kernel_size=1, padding=
        "same")(x)

    for _ in range(self.num_blocks):
        residual = x
        x1 = layers.DepthwiseConv2D(
            kernel_size=7, padding="same", use_bias=False
        )(x)
        x1 = layers.LayerNormalization(epsilon=1e-6)(x1)
        x1 = layers.Conv2D(
            4 * self.filters,
            kernel_size=1,
            activation="gelu",
            padding="same",
        )(x1)
        x1 = layers.Conv2D(self.filters, kernel_size=1,
            padding="same")(x1)
        x = layers.Add()([x, x1])
        x = layers.BatchNormalization()(x)

    # Reduction finale en 32 canaux pour
        create_policy_value_heads
    x = layers.Conv2D(32, kernel_size=1, padding="same",
        use_bias=False)(x)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    self.x = x
```

Code source 10. *Backbone* GONetConvNeXtLike implémenté dans GONetConvNeXtV1.

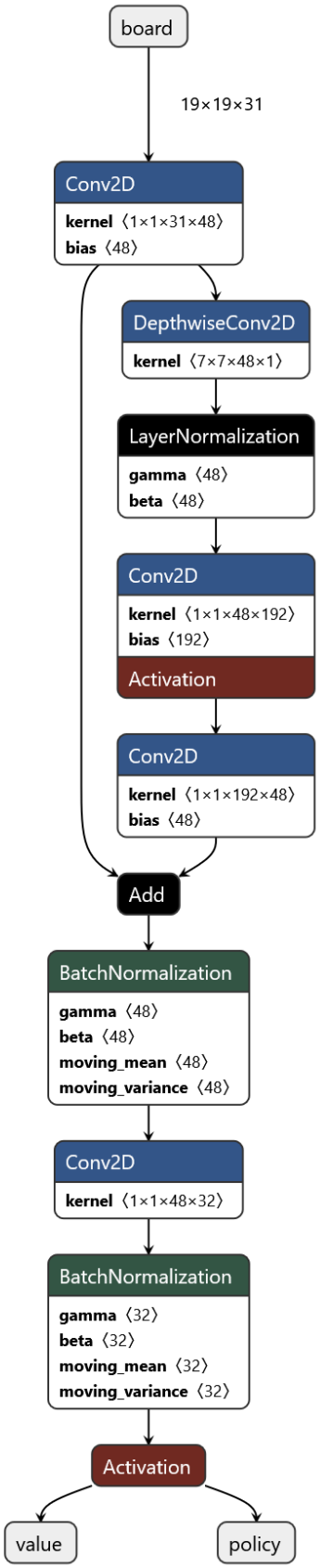
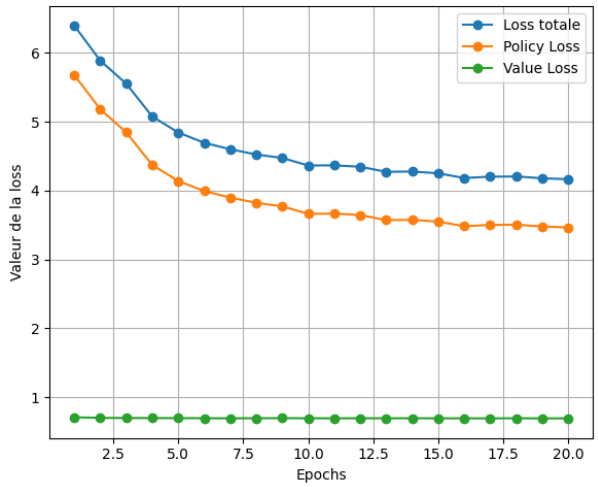


Figure 17. *backbone* du réseau GONetConvNeXtLike (1 bloc).

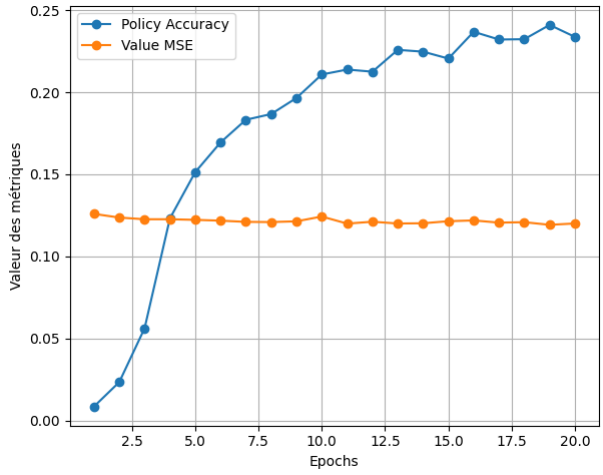
Cette implémentation met en œuvre les éléments essentiels de ConvNeXt tout en les adaptant au format des données du jeu de Go (plans 19×19), assurant ainsi une bonne capacité d'apprentissage dans un cadre à faible budget de paramètres.

C. Analyse des résultats obtenus

La figure 18 présente l'évolution des pertes d'apprentissage (cf. 18a) ainsi que des métriques associées (cf. 18b) pour le modèle GONetConvNeXtLike entraîné sur 20 époques.



(a) Évolution des pertes d'entraînement et de valeur pour GONetConvNeXtLike (1 bloc).



(b) Évolution des métriques (précision catégorielle et *MSE*) pour GONetConvNeXtLike (1 bloc).

Figure 18. Courbes d'apprentissage pour le modèle GONetConvNeXtLike (1 bloc).

La figure 18a illustre la manière dont la perte totale diminue résultant exclusivement de la baisse de la perte de politique.

Sur le plan des métriques, la précision catégorielle de la tête *policy* atteint un niveau encourageant, avoisinant les 25% en fin d'apprentissage, ce qui en fait l'un des meilleurs résultats obtenus par rapport aux autres architectures. En parallèle, l'erreur quadratique moyenne (*MSE*) de la tête *value* se stabilise autour de 0.12.

Ces résultats démontrent que l'architecture ConvNeXt-like permet une extraction de caractéristiques suffisamment expressive pour améliorer l'apprentissage de la politique tout en assurant une estimation plus stable de la valeur, surpassant notamment les résultats obtenus avec les variantes plus légères comme MobileNet.

À la lumière des performances prometteuses observées pour l'architecture GONetConvNeXtLike, il est apparu pertinent de poursuivre l'analyse en explorant plus en détail l'effet de la profondeur du réseau. En particulier, une attention a été portée sur l'augmentation progressive du nombre de blocs ConvNeXt afin d'évaluer son impact sur les capacités d'apprentissage.

Après une étude préalable des contraintes de complexité, il a été déterminé que le nombre maximal de blocs résiduels autorisé sans dépasser la limite fixée de 100 000 paramètres est de trois. Cela a permis de définir un protocole d'expérimentation systématique sur des architectures contenant 1, 2 et 3 blocs, dans le but de caractériser leur influence sur les performances globales du modèle.

D. Analyse quantitative de l'augmentation du nombre de blocs résiduels sur l'efficacité du réseau

L'analyse des performances des variantes de l'architecture GoNetConvNeXtLike selon le nombre de blocs résiduels (1 à 3) révèle que l'augmentation de profondeur n'apporte pas d'amélioration significative dans le cadre de la contrainte imposée sur le nombre de paramètres.

Comme l'indique la figure 23, les trois configurations atteignent une précision comparable, légèrement inférieure à 25%, avec un léger avantage pour l'architecture à 2 blocs. Cela suggère qu'un réseau peu profond est déjà capable d'extraire les *patterns* pertinents pour la politique.

Les figures 19 et 20 confirment cette observation : les courbes de perte (totale et de politique) diminuent de manière similaire pour les trois architectures. L'ajout d'un second ou d'un troisième bloc n'améliore pas significativement le taux de convergence ni les valeurs finales.

Concernant la tête de valeur, les figures 21 et 22 montrent que ni la *value loss* ni le *MSE* ne bénéficient d'une augmentation du nombre de blocs. Les performances stagnent autour de 0,69 pour la perte de valeur et de 0,12 pour le *MSE*, confirmant les limites de prédiction de la tête de valeur sur ce type de données simulées.

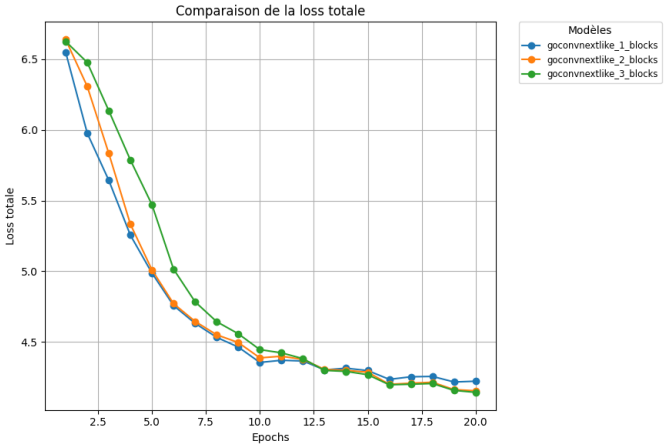


Figure 19. Comparaison de la *loss* totale pour les architectures `GONetConvNeXtLike` avec 1 à 3 blocs.

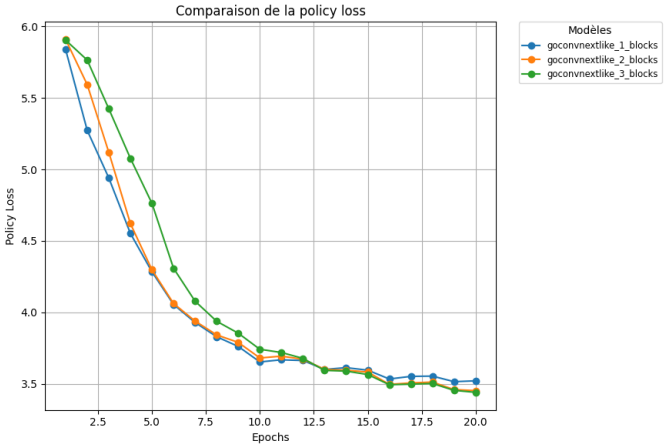


Figure 20. Évolution de la *policy loss* selon le nombre de blocs pour les modèles `GONetConvNeXtLike`.

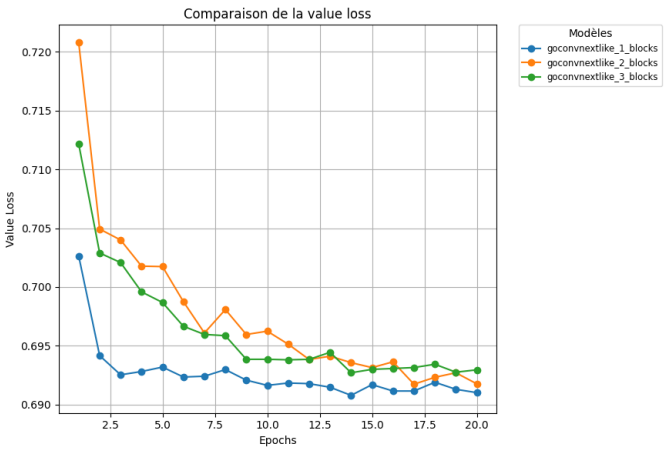


Figure 21. Évolution de la *value loss* selon la profondeur de l'architecture `ConvNeXtLike`.

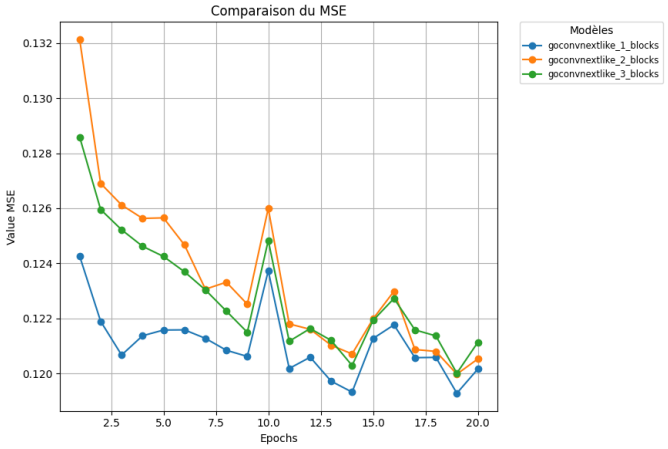


Figure 22. Comparaison des valeurs de MSE pour les têtes `value` sur les différentes variantes `GONetConvNeXtLike`.

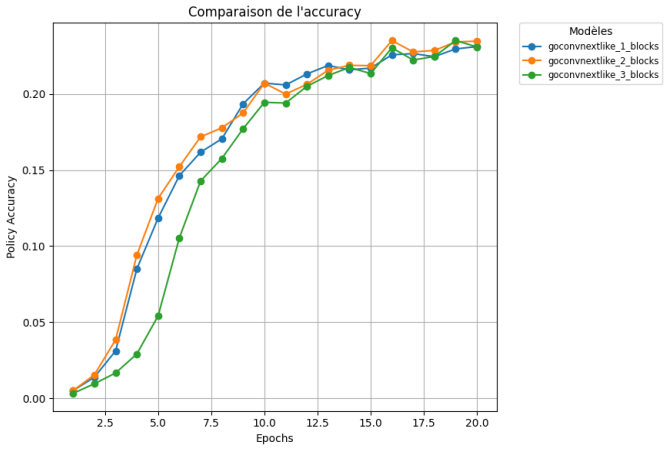


Figure 23. Comparaison de l'*accuracy* des têtes *policy* selon la profondeur de l'architecture `ConvNeXtLike`.

En conclusion, ces résultats suggèrent que pour ce type d'architecture ConvNeXt-like contraint par une capacité réduite, augmenter la profondeur au-delà d'un certain point n'apporte pas de bénéfice notable, et peut même induire une complexité inutile. **L'architecture à 2 blocs présente les meilleures performances parmi celles évaluées.**

IX. Réseaux de neurones type ShuffleNet

A. Principes généraux

ShuffleNet est une architecture de réseau de neurones convolutifs conçue pour les dispositifs à ressources limitées, notamment les mobiles et objets connectés. Elle a été introduite par Zhang et al. dans leur article "ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices" [4], dans le but de réduire drastiquement le coût computationnel tout en maintenant une précision compétitive sur des tâches de vision.

L'innovation principale repose sur deux mécanismes combinés :

1) *Group convolutions* : pour diviser les canaux d'entrée en sous-groupes traités séparément, réduisant ainsi le nombre d'opérations ;
2) *Channel Shuffle* (Permutation des canaux) : pour restaurer la communication entre les groupes en permutant les canaux après chaque bloc, ce qui évite l'isolation des groupes introduite par les convolutions groupées.

L'architecture se compose donc typiquement de blocs comportant :

- Une convolution 1×1 groupée, suivie d'une activation ;
- Une convolution 3×3 en profondeur (*depthwise separable convolution*), pour le traitement spatial ;
- Une opération de *channel shuffle*, qui assure un mélange efficace de l'information à travers les groupes.

Cette approche permet à ShuffleNet d'obtenir des performances proches de MobileNet, voire supérieures, tout en utilisant encore moins de paramètres. En ce sens, il peut être vu comme une alternative plus légère à MobileNet, avec une expressivité maintenue grâce au mécanisme de permutation.

ShuffleNet s'inscrit donc dans la famille des réseaux efficaces, en misant sur l'optimisation de l'architecture plutôt que sur une augmentation brute des capacités, ce qui en fait un choix pertinent dans notre contexte contraint à 100 000 paramètres.

B. Description de l'architecture

Chaque bloc commence par une convolution groupée (*gconv*) élargissant le nombre de canaux, suivie d'une normalisation puis d'un *channel shuffle* (cf. code source 13). Une convolution *depthwise* applique ensuite des filtres spatiaux indépendants par canal. Enfin, une deuxième convolution groupée projette à nouveau les canaux vers une dimension réduite. Le tenseur d'entrée est alors additionné (résidu) pour former la sortie du bloc (cf. code source 12).

Le *backbone*, dont l'implémentation est présentée dans le code source 11 et la figure 24 commence par une projection initiale suivie d'une répétition de blocs *ShuffleNet*. Ce tronc reste compatible avec les têtes de politique et de valeur communes aux autres modèles `GONet`.

En adoptant ce design, le réseau bénéficie d'une réduction significative de la complexité (via la séparation de canaux et la réduction du nombre d'opérations), tout en maintenant une expressivité suffisante pour extraire des motifs pertinents dans des jeux de données visuels.

```
def set_backbone(self):
    x = Conv2D(self.trunk, 1, padding='same',
         kernel_regularizer=regularizers.l2(0.0001))(self.
         input_layer)
    x = BatchNormalization()(x)
    x = ReLU()(x)

    for _ in range(self.num_blocks):
        x = self.bottleneck_block(x, expand=self.filters *
             3, squeeze=self.trunk)

    self.x = x
```

Code source 11. Méthode `set_backbone()` du modèle `GONetShuffleNet`.

```
def bottleneck_block(self, x, expand, squeeze):
    residual = x
    x = self.gconv(x, channels=expand, shuffle_groups=self.
         shuffle_groups)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = self.channel_shuffle(x, self.shuffle_groups)
    x = DepthwiseConv2D(kernel_size=3, padding='same',
         use_bias=False)(x)
    x = BatchNormalization()(x)
    x = self.gconv(x, channels=squeeze, shuffle_groups=self
         .shuffle_groups)
    x = BatchNormalization()(x)
    x = Add()([residual, x])
    return ReLU()(x)
```

Code source 12. Bloc `bottleneck_block()` avec *grouped convolutions* et `channel shuffle`.

```
def gconv(self, x, channels, shuffle_groups):
    input_ch = x.shape[-1]
    group_ch = input_ch // shuffle_groups
    out_ch = channels // shuffle_groups
    group_outputs = []
    for i in range(shuffle_groups):
        slice_x = x[:, :, :, i * group_ch:(i + 1) *
             group_ch]
        conv = Conv2D(out_ch, kernel_size=1, use_bias=False
             )(slice_x)
        group_outputs.append(conv)
    return Concatenate(axis=-1)(group_outputs)

def channel_shuffle(self, x, shuffle_groups):
    batch_size = tf.shape(x)[0]
    height = tf.shape(x)[1]
    width = tf.shape(x)[2]
    channels = tf.shape(x)[3]

    if x.shape[-1] is not None and x.shape[-1] %
         shuffle_groups != 0:
        raise ValueError("Le nombre de canaux doit etre
             divisible par le nombre de groupes.")

    group_channels = channels // shuffle_groups
    x = tf.reshape(x, [batch_size, height, width,
         shuffle_groups, group_channels])
    x = tf.transpose(x, [0, 1, 2, 4, 3])
    x = tf.reshape(x, [batch_size, height, width, channels
         ])
    return x
```

Code source 13. Fonctions auxiliaires `gconv()` et `channel_shuffle()` du modèle GONetShuffleNet.

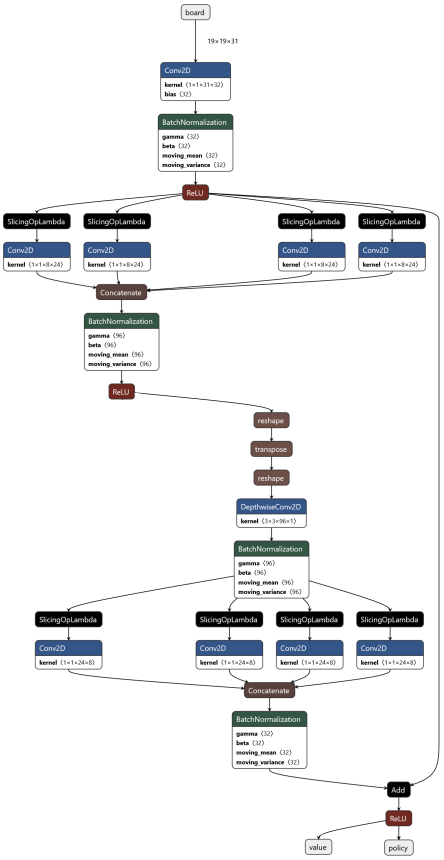


Figure 24. *backbone* du réseau GONetShuffleNet (1 bloc). Certains éléments ont été retirés pour permettre une meilleure aisance de lecture.

C. Analyse des résultats obtenus

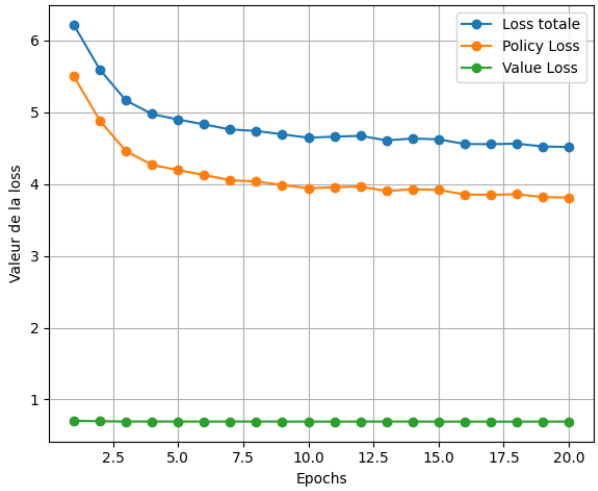
Les figures 25a et 25b présentent les performances du modèle GONetShuffleNet avec un seul bloc. L'observation des pertes montre une décroissance la perte de la politique, ce qui témoigne d'un apprentissage de la stratégie au fur et à mesure des époques.

Sur le plan des métriques (cf. figure 25b), la précision de la politique atteint environ 22%, ce qui constitue un bon résultat pour une architecture aussi légère, comparable à ceux obtenus par les meilleures configurations MobileNet et ConvNeXt-like. En revanche, la *MSE* reste relativement constante autour de 0,125, signe d'une difficulté persistante à estimer correctement la valeur d'une position, un phénomène déjà observé dans les modèles précédents.

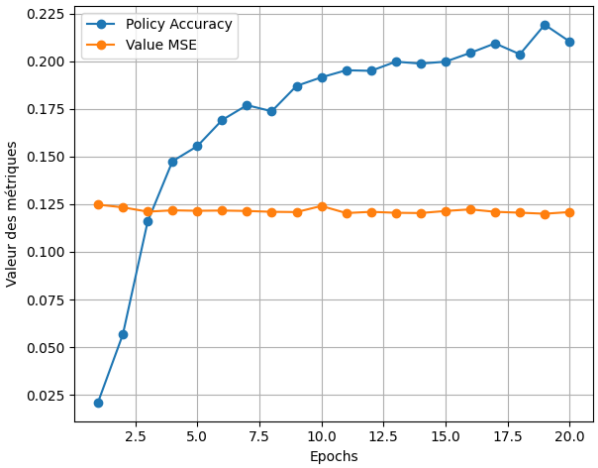
Dans une démarche analogue à celle menée pour les architectures ResNet, MobileNet et ConvNeXt, examinons l'influence de l'augmentation du nombre de blocs sur les performances du modèle.

D. Analyse quantitative de l'augmentation du nombre de blocs résiduels sur l'efficacité du réseau

Les figures 26 à 30 présentent une évaluation comparative du modèle GoNetShuffleNet en fonction du nombre de blocs utilisés dans le *backbone*.



(a) Évolution des pertes d'entraînement pour le modèle GONetShuffleNet avec 1 bloc.



(b) Évolution des métriques (*policy accuracy* et *value MSE*) pour le modèle GONetShuffleNet avec 1 bloc.

Figure 25. Courbes d'apprentissage du modèle GONetShuffleNet avec une profondeur d'un bloc résiduel.

La figure 26 montre une réduction progressive de la loss totale pour la majorité des architectures, avec une convergence autour de la vingtième époque. Toutefois, on remarque que l'ajout massif de blocs (notamment à partir de 13 blocs) n'apporte plus d'amélioration significative. L'architecture à 24 blocs affiche même une stagnation marquée, voire une instabilité. Cette tendance se retrouve également dans la perte liée à la prédiction de la valeur, illustrée en figure 27 : celle-ci reste très similaire entre les différentes variantes, montrant une convergence rapide dès les premières époques. Cela suggère que l'augmentation de la profondeur n'améliore pas la capacité du modèle à estimer la valeur d'une position.

En parallèle, la figure 28 met en évidence une diminution régulière de la *policy loss* jusqu'à un certain seuil (environ 6 à 8 blocs), au-delà duquel les courbes ont tendance à se chevaucher. Cela indique une saturation de l'efficacité

d'apprentissage de la politique. Cette observation est renforcée par la figure 29, qui montre que la *MSE* atteint un plateau avec des écarts faibles entre les différentes configurations. Le gain de précision sur la tête de valeur devient donc marginal après un certain nombre de blocs.

Enfin, la figure 30 met en lumière que les meilleures performances en termes d'*accuracy* (près de 24%) sont atteintes par des modèles comportant entre 4 et 8 blocs. En revanche, l'architecture à 24 blocs progresse lentement, voire échoue à converger correctement. En l'occurrence, la meilleure performance observée est celle associée à l'architecture disposant de **4 blocs**, avec 23,67% d'*accuracy*.

Ces résultats démontrent que l'augmentation du nombre de blocs dans GONetShuffleNet n'apporte un gain que jusqu'à une certaine limite. Une complexité trop élevée devient pénalisante, notamment en raison des contraintes sur le nombre de paramètres. Une profondeur modérée (entre 4 et 8 blocs) semble offrir un bon compromis entre performance et efficacité dans le contexte du modèle restreint à 100 000 paramètres.

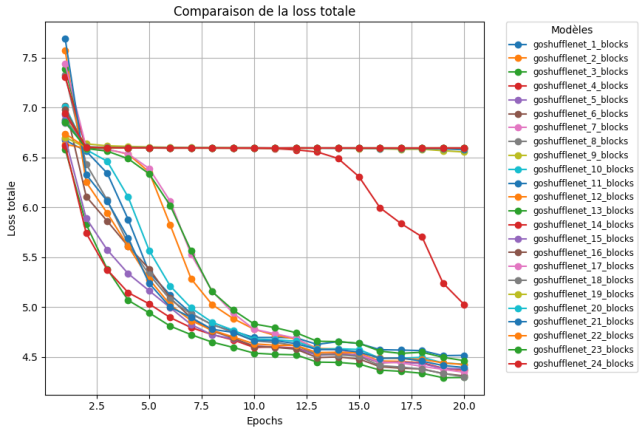


Figure 26. Comparaison de la **loss totale** pour différentes profondeurs du modèle GoNetShuffleNet.

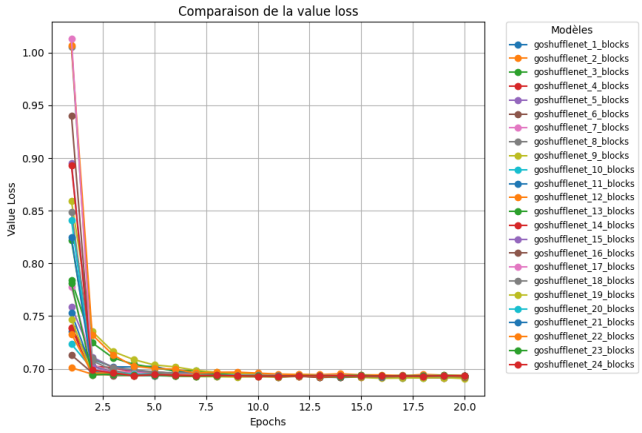


Figure 27. Comparaison de la **value loss** selon la profondeur du modèle GoNetShuffleNet.

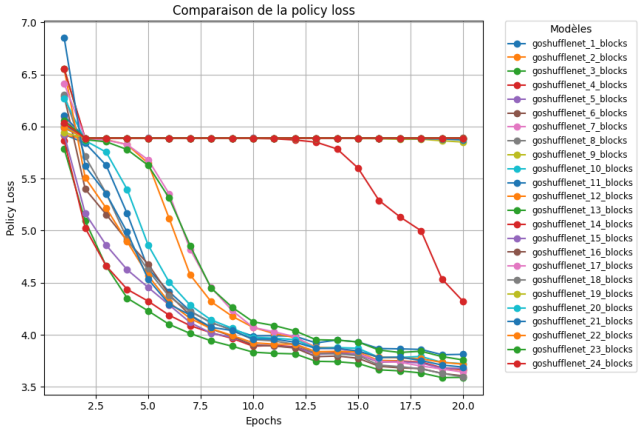


Figure 28. Comparaison de la **policy loss** pour différents nombres de blocs dans le modèle GoNetShuffleNet.

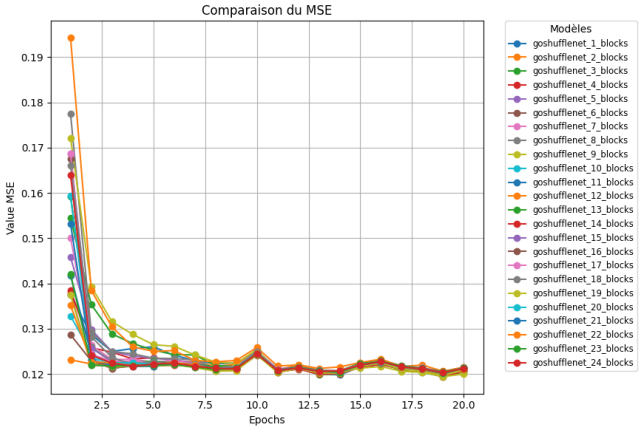


Figure 29. Comparaison du **MSE de la tête value** pour les variantes de GoNetShuffleNet.

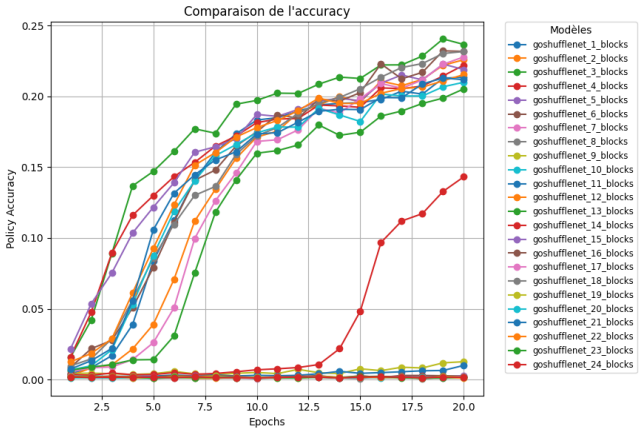


Figure 30. Comparaison de l'**accuracy de la tête policy** pour les différentes variantes de GoNetShuffleNet.

Malgré l'efficacité des blocs de type ShuffleNet en matière de réduction de paramètres et de calculs, ces structures reposent sur des schémas classiques de convolutions standards et groupées. Ces approches restent limitées dans leur capacité à capturer les dépendances spatiales croisées complexes, essentielles dans des tâches stratégiques comme le jeu de Go, où les interactions entre lignes et colonnes jouent un rôle fondamental.

C'est dans cette optique que nous nous tournons désormais vers les réseaux à convolution en croix (*cross convolution networks*), une alternative architecturale conçue pour mieux modéliser les interactions orthogonales.

X. Réseaux de neurones type convolution en croix

Nous introduisons à présent une nouvelle famille d'architectures intégrant des convolutions en croix, qui visent à dépasser les limites observées dans les approches précédentes. L'objectif est d'évaluer si cette stratégie permet de mieux capter les motifs stratégiques complexes du Go tout en conservant une empreinte paramétrique maîtrisée.

Dans le cadre de l'étude bibliographique, l'article [5], consacré aux réseaux de neurones à convolutions en croix, a été identifié comme une approche prometteuse. Cette architecture a été adaptée au projet, sous la contrainte de ne pas dépasser 100 000 paramètres.

A. Principes généraux

Les convolutions en croix cherchent à améliorer la capacité d'un réseau à :

- Capturer les dépendances spatiales globales, en particulier entre les lignes et les colonnes ;
- Réduire les coûts computationnels, en factorisant les opérations convolutives (souvent via des convolutions séparables ou orthogonales) ;
- Maintenir une structure légère mais plus expressive que les convolutions classiques.

B. Description de l'architecture de référence et réduction du nombre de paramètres associés

Le *backbone* du modèle GONetCrossLayer repose sur deux branches symétriques intégrant des convolutions classiques et des couches *cross-layer* (combinant convolutions verticales et horizontales).

1) Branche gauche:

- **Conv2D(128, 7×7)** : $(7 \times 7 \times 31 + 1) \times 128 = 195328$ paramètres ;
- **Conv2D(16, 1×1)** : $(1 \times 1 \times 128 + 1) \times 16 = 2064$ paramètres ;
- **Cross-layer(1×1, 16 filtres)** : $2 \times (1 \times 1 \times 16 + 1) \times 16 = 544$ paramètres ;
- **Conv2D(64, 1×1)** : $(1 \times 1 \times 16 + 1) \times 64 = 1088$ paramètres.

Total branche gauche : 199024 paramètres

2) Branche droite:

- **Conv2D(16, 1×1)** : $(1 \times 1 \times 31 + 1) \times 16 = 512$ paramètres ;
- **Cross-layer(5×1 et 1×5, 16 filtres)** : $2 \times (5 \times 1 \times 16 + 1) \times 16 = 2592$ paramètres ;
- **Conv2D(64, 1×1)** : $(1 \times 1 \times 16 + 1) \times 64 = 1088$ paramètres

Total branche droite : 4192 paramètres

En addition la branche de gauche et de droite, le nombre de paramètres total dans le bloc de référence est donc 199024 + 4192 = 203216

Ce total dépasse la contrainte de **100 000 paramètres** fixée. Une modification de l'architecture est donc nécessaire.

Plusieurs ajustements permettent de réduire significativement la complexité sans compromettre la structure du réseau :

- **Réduire les filtres de la première couche** : passer de 128 à 32

$$(7 \times 7 \times 31 + 1) \times 32 = 48832$$

- **Réduire les filtres des `cross-layer`** : passer de 16 à 8 filtres
- **Réduire les filtres finaux** : passer de `Conv2D(64)` à `Conv2D(32)`

Le tableau 2 rassemble les modifications apportées au réseau de référence permettant ainsi de respecter la contrainte des **100 000 paramètres** tout en conservant l'idée originale de symétrie et de convolutions en croix.

Tableau 2
DIMINUTION DU NOMBRE DE PARAMÈTRES DANS L'ARCHITECTURE DE CONVOLUTION EN CROIX.

Composant	Nombre de paramètres estimé
`Conv2D(32, 7×7)`	Environ 48 800 paramètres
`Cross-layer + 1×1 convs`	Environ 15 000 paramètres
`Conv2D(32)` (2 couches)	Environ 2 000 paramètres
Total	**Environ 65 000 paramètres**

C. Description de l'architecture

Les codes sources 14 et 15 illustrent respectivement l'implémentation du *backbone* de l'architecture GONetCrossLayer et la définition d'une couche en croix. L'architecture complète du modèle est quant à elle représentée dans la figure 31.

```
def set\_backbone(self):
    # Branche gauche
    left = layers.Conv2D(
        32, 7, padding="same", kernel\_regularizer=
            regularizers.l2(0.0001)
    )(self.input\_layer)
    left = layers.BatchNormalization()(left)
    left = layers.ReLU()(left)

    left = layers.Conv2D(
        8, 1, padding="same", kernel\_regularizer=
            regularizers.l2(0.0001)
    )(left)
    left = layers.BatchNormalization()(left)
    left = layers.ReLU()(left)

    left = self.cross\_layer(left, width=1, filters=8)

    left = layers.Conv2D(
        32, 1, padding="same", kernel\_regularizer=
            regularizers.l2(0.0001)
    )(left)

    # Branche droite
    right = layers.Conv2D(
        8, 1, padding="same", kernel\_regularizer=
            regularizers.l2(0.0001)
    )(self.input_layer)
    right = layers.BatchNormalization()(right)
    right = layers.ReLU()(right)

    right = self.cross\_layer(right, width=5, filters=8)

    right = layers.Conv2D(
        32, 1, padding="same", kernel\_regularizer=
            regularizers.l2(0.0001)
    )(right)

    # Fusion
    x = layers.Add()([left, right])
    x = layers.ReLU()(x)

    self.x = x
```

Code source 14. *Backbone* du modèle GONetCrossLayer.

```
def cross\_layer(self, x, width=1, filters=16):
    """
    Implemente une couche convolutive "en croix" via deux
        convolutions separables :
    - verticale : (k, 1)
    - horizontale : (1, k)
    """
    vertical = layers.Conv2D(
        filters, (width, 1), padding="same", activation="
            relu"
    )(x)
    horizontal = layers.Conv2D(
        filters, (1, width), padding="same", activation="
            relu"
    )(x)
    return layers.Add()([vertical, horizontal])
```

Code source 15. Implémentation d'une couche en croix du modèle GONetCrossLayer.

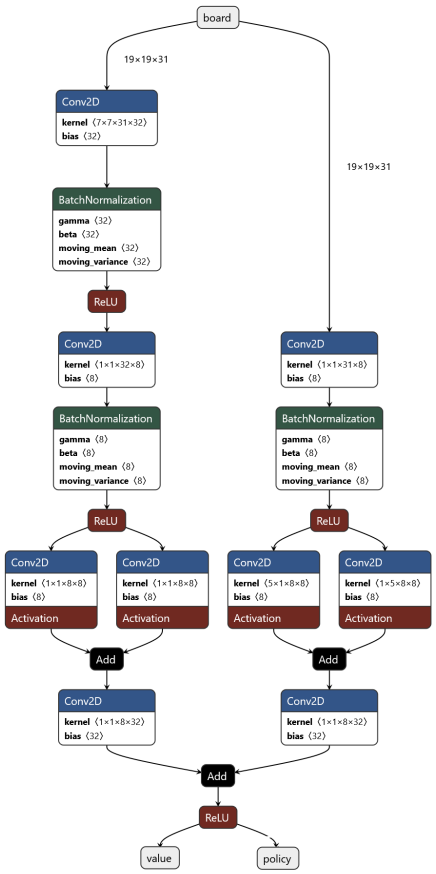


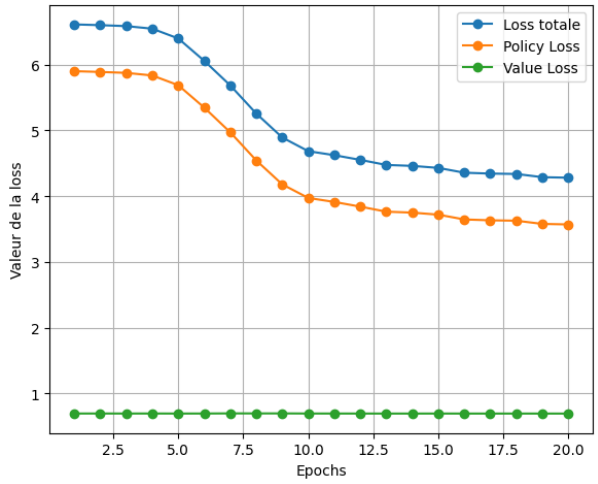
Figure 31. *backbone* du réseau GONetCrossLayer.

D. Analyse des résultats obtenus

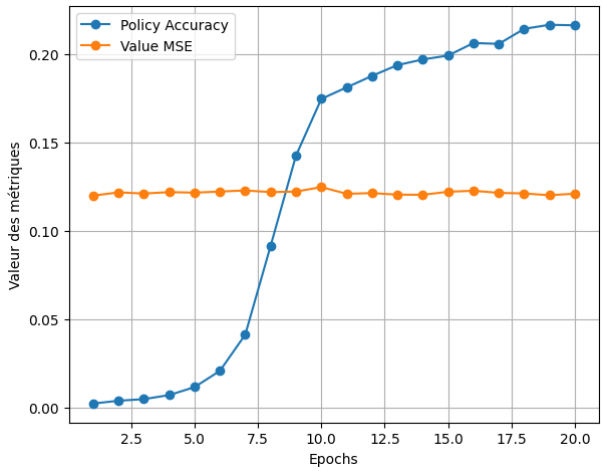
Au cours des 20 époques d'entraînement, la *loss* totale du modèle GONetCrossLayer a diminué de manière continue, passant d'environ 6.5 à une valeur finale de 4.2796. Cette tendance indique une amélioration générale de l'optimisation du modèle. En parallèle, la *loss value* est restée stable autour de 0.69, avec une valeur finale de 0.6927, traduisant une difficulté persistante à améliorer cette composante durant l'apprentissage.

La *policy loss*, en revanche, a nettement baissé, passant de près de 5.9 à 3.5690, signalant une progression efficace sur cet aspect du modèle. Cette dynamique est cohérente avec l'évolution de la *MSE*, qui reste relativement constant et atteint 0.1210 en fin d'entraînement, confirmant l'absence d'amélioration notable sur la tête de valeur.

Enfin, la précision catégorielle de la politique a montré une nette progression, atteignant 0.2162 après un départ très faible. Cela reflète une capacité croissante du modèle à reproduire fidèlement les coups issus des données d'entraînement, et met en évidence la solidité de l'architecture GONetCrossLayer pour l'apprentissage de la politique.



(a) Évolution des pertes d'entraînement et de validation pour GONetCrossLayer.



(b) Évolution des métriques (*accuracy* et *MSE*) pour GONetCrossLayer.

Figure 32. Courbes d'apprentissage pour le modèle GONetCrossLayer.

XI. Choix du modèle final

A. Accuracy après 50 époques

Afin d'identifier l'architecture présentant les meilleures performances, un entraînement de 50 époques a été effectué pour chaque modèle (cf. figure 33).

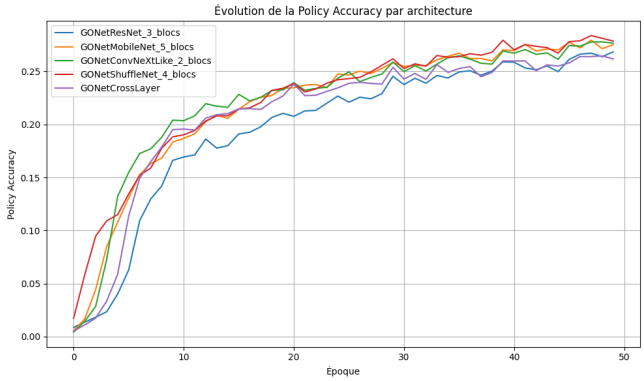


Figure 33. Comparaison de l'**accuracy** pour les structures retenues entraînées sur 50 époques.

Dans la continuité, le tableau 3 présente les *accuracies* finales des architectures évaluées, classées par ordre décroissant.

Tableau 3
Précision finale obtenue pour chaque architecture après 50 époques d'entraînement.

Architecture	Accuracy finale (%)
GONetShuffleNet_4_blocs	27.85
GONetConvNeXtLike_2_blocs	27.65
GONetMobileNet_5_blocs	27.51
GONetResNet_3_blocs	26.83
GONetCrossLayer	26.17

Malgré une amélioration globale des performances au fil des époques, les résultats obtenus restent modestes, avec des *accuracies* finales comprises entre **26,17%** et **27,85%**. Aucun des modèles testés avec 50 époques ne dépasse significativement le seuil des 30%. Cela montre les limites des architectures évaluées dans leur capacité à prédire précisément le coup joué par un expert à partir d'une position donnée.

Parmi les cinq architectures retenues, **GONetShuffleNet_4_blocs** se démarque comme la plus performante à l'issue des 50 époques, suivie de près par ConvNeXtLike et MobileNet.

Ainsi, le modèle final fourni est un **modèle basé sur l'architecture ShuffleNet avec 4 blocs** entraîné sur 100 époques (cf. 34) et aboutit à une *accuracy* de **30,74%**.

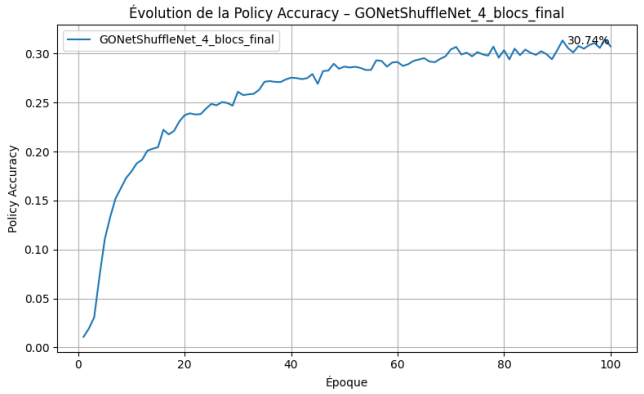


Figure 34. Augmentation de l'*accuracy* sur l'architecture GONetShuffleNet avec 4 blocs sur 100 époques.

XII. Conclusion

Ce travail a permis d'explorer et de comparer plusieurs architectures de réseaux de neurones appliquées à la prédiction de coups au jeu de Go. Cinq modèles aux structures variées ont été retenus, reposant sur des paradigmes architecturaux distincts tels que les connexions résiduelles (ResNet), les convolutions séparables (MobileNet), les inspirations transformer (ConvNeXt), les schémas de permutation de canaux (ShuffleNet) ou encore les convolutions en croix.

Parmi ces architectures, le modèle GONetShuffleNet_4_blocs a obtenu les meilleures performances sur un entraînement de 50 époques, avec une précision finale de 27.85%. Poussé à 100 époques, ce même modèle atteint une *accuracy* de 30.74%, témoignant d'un potentiel d'apprentissage plus important que ses concurrents. Toutefois, les résultats globaux restent modestes, et aucun modèle "a dépassé le seuil de 31% de précision.

Les performances limitées observées s'expliquent en partie par la contrainte stricte imposée sur le nombre de paramètres (100 000) et par un protocole centré exclusivement sur la variation de profondeur des architectures. Faute de temps, d'autres dimensions structurantes, comme le choix des noyaux de convolution, n'ont pas été explorées. Par ailleurs, ces résultats confirment que l'augmentation de la profondeur ne garantit pas une amélioration des performances. L'optimisation architecturale doit ainsi s'inscrire dans une approche plus globale, intégrant la nature du problème, la richesse des données et le contexte stratégique du jeu.

En somme, bien que les architectures testées aient montré des capacités d'apprentissage différenciées, les résultats suggèrent la nécessité d'explorer des approches plus robustes, éventuellement inspirées de modèles multi-têtes, de politiques probabilistes ou encore de mécanismes d'attention, pour progresser vers un apprentissage plus compétitif dans le cadre du jeu de Go.

Glossaire

backbone
: Partie centrale du réseau dédiée à l'extraction hiérarchique de caractéristiques. 3

convolution *depthwise*
: La convolution depthwise consiste à appliquer un filtre sur chaque canal, contrairement à la convolution classique qui applique un filtre sur l'ensemble des canaux. 10

convolution *pointwise*
: Type de convolution utilisant un noyau 1×1, appliqué point par point à travers l'image. Ce noyau couvre toute la profondeur des canaux d'entrée. Employée avec les convolutions *depthwise*, elle permet de former des convolutions séparables en profondeur, plus efficaces en calcul. 10

downsampling
: Dans les réseaux de convolution, le *downsampling* est une technique essentielle pour réduire la dimension spatiale des données tout en augmentant le champ réceptif des neurones. Il peut être réalisé via du *pooling* ou des convolutions à *stride* > 1. Toutefois, dans le cas de GONetDemo, ce mécanisme est volontairement absent afin de préserver la structure locale du plateau. 5

hyperparamètre
: Élément indépendant de l'apprentissage, tels que le nombre de nœuds et la taille des couches cachées du réseau de neurones, l'initialisation des poids, le coefficient d'apprentissage, la fonction d'activation. 2

MaxPooling
: Opération de sous-échantillonnage qui conserve la valeur maximale dans une région locale. 5

pooling
: Le Pooling est utilisé pour réduire la taille des données. En effet, elle compresse le tenseur d'entrée pour n'en garder que l'information pertinente. Comme la couche de Convolution, elle fonction par morceaux. 23

stride
: Pas de déplacement du noyau de convolution ; un stride de 2 réduit la résolution spatiale par deux. 5, 23

References

[1] T. Cazenave, "Residual networks for computer go," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, 2018.

[2] ——, *Mobile networks for computer go*, 2020. arXiv: 2008.10080 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2008.10080.

[3] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, *A convnet for the 2020s*, 2022. arXiv: 2201.03545 [cs.CV]. [Online]. Available: https://arxiv.org/abs/2201.03545.

[4] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *arXiv preprint arXiv:1707.01083*, 2017.

[5] J. Barratt and C. Pan, *Playing go without game tree search using convolutional neural networks*, 2019. arXiv: 1907.04658 [cs.AI]. [Online]. Available: https://arxiv.org/abs/1907.04658.