Deep Learning for Computer Go Project

Author: Roc de Larouzière
Supervisor: Prof. Tristan Cazenave

Master Intelligence Artificielle, Systèmes, Données
Université Paris Dauphine
March 2025

Contents

1 Introduction

1.1 History of Computer Go

The game of Go was invented more than 4 000 years ago in China. It is one of the oldest and most complex board game. Its apparent simplicity, with basic rules that are easy to learn, contrasts with the intellectual depth it offers, which thus makes each game unique.
Despite rapid advances in Artificial Intelligence in games such as chess, Go has for a long time remained an important challenge for programmers. The dimension of the board, known as the Goban, and the combinatorial explosion of possible configurations rendered traditional AI approaches ineffective. At the beginning of the 21st century, significant progresses were made, thanks to the integration of techniques such as neural networks and deep learning. In 2016, a major milestone was reached when DeepMind's AlphaGo program beat world champion Lee Sedol, making a historic advance in AI [9].
Since then, the development of sophisticated classification models has made it possible to accurately analyze and predict moves in Go. These technological advances have first enriched our understanding of the game but have also opened new perspectives in the application of AI to complex domains that require strategic decision-making.

1.2 The project

The goal of the project was to build and train neural networks for playing the game of Go. These networks were then plugged into a Monte Carlo Tree Search (MCTS) model, made by our professor, to compete against other models proposed by students, on a weekly basis contest.

Originally, the model of AlphaGo [9] used two distinct neural networks: the policy network and the value network, made respectively to predict the optimal policy given a particular board position and estimate the probability of winning given the same board position. These networks were trained separately and their evaluations combined in the MCTS.
Subsequently, AlphaGo Zero [10] introduced a unified architecture with a single network that has two heads, a policy head and a value head. This configuration simplified the global architecture and enhanced the program performances.
For the project, our models had to be built on this global architecture: a single network with a policy and a value head, like in the figure underneath.

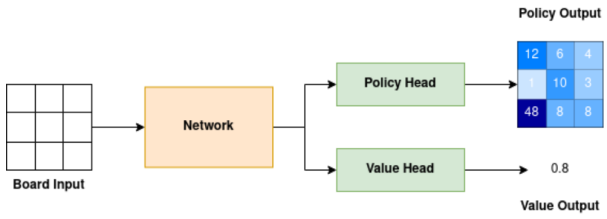

Figure 1: AlphaGo Zero network architecture [8]

In order to be fair about training resources, the number of parameters for the networks had to be lower that 100 000.

The data used for training comes from the Katago Go program self-played games. Katago is one of the most powerful Go programs, far surpassing professional human level through self-play training and it also has an architecture inspired by AlphaGo Zero. There are 1 000 000 different games in total in the training set.
The input data is composed of 31 19×19 planes that represent the ladders, the current state on two planes, the two previous states on four planes and the color to play. Given our model architecture, the output targets of our models are, the policy to follow, represented by a vector of size 361 with a 1 for the move to play and 0 otherwise, and the value, close to 1 if the player in White wins and close to 0 if the player in Black wins. The dataset management with the collection of the Katago plays, the transformation into the 31 19×19 planes and the creation of a training set in batches and validation sets were made thanks to the Golois library, implemented by our professor.

2 Computation

For this project, I ran my programs on a MacBook Air M1 using Python and Tensorflow with the tensorflow-metal extension. The M1 chip is, thanks to the Metal backend, well recognized as a GPU with 8 GB of system memory and around 2.67 GB of available cache. The following table shows the execution times of my programs:

Network	Training compute
ResNetSanSe	10 epochs : 10×15 = 150 s (2min30) 300 epochs : 150×30 = 1h15
ResNet	10 epochs : 14×10 = 140 s (2min20) 300 epochs : 140×30 = 1h10
EfficientFormerRough	10 epochs : 775 s (13min) 1000 epochs : 21h30
EfficientFormerWay	10 epochs : 728 s (12min) 1000 epochs : 20h
GOat	10 epochs : 1110 s (18min30) 900 epochs : 27h
MobileNet3_1	10 epochs : 367 s (6min) 800 epochs : 8h
MobileNet3_2	10 epochs : 469 s (7min45) 800 epochs : 10h30

Table 1: Training compute of the models

3 Residual Networks

3.1 The structure

Residual Networks have been introduced in 2015. They have had a major impact in the world of AI, specifically for computer vision tasks (image classification, detection, segmentation etc.) and have also been used in other AI domains such as Natural Language Processing for example.

The main idea behind residual network architecture is the use of “skip” connections that skip one or more layers, forming a residual block.

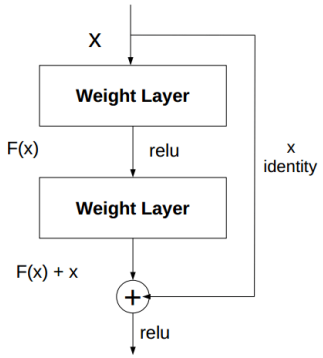

Figure 2: The residual block

In concrete terms, we keep the input x and we add it to a residual calculated by the block. The output of the residual block becomes $y = x + F(x)$ as shown in the figure above.
If the bloc doesn't need to transform the input, then $F(x)$ will learn to become null, and the output will become $y \approx x$. Hence, it allows the neural network to learn the identity function. Also, in residual networks, gradients backpropagate more smoothly because they do not need to pass entirely through complex operations like convolution. This tackles the vanishing gradient problem.

3.2 Implementation

To implement some residual networks for our project, I first took the residual block structures in [2]. This network is named ResNetSanSe. The input layer and the residual blocks are described by the following figures.

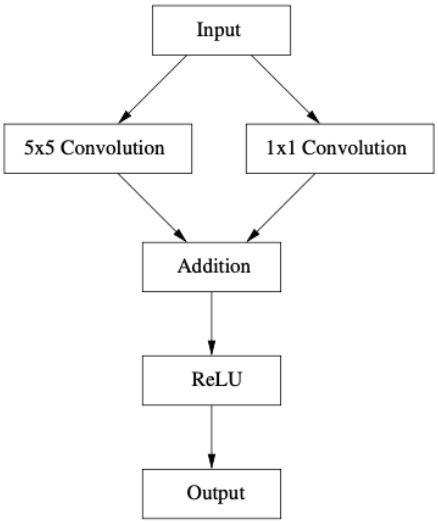

Figure 3: Input Layer used [2]

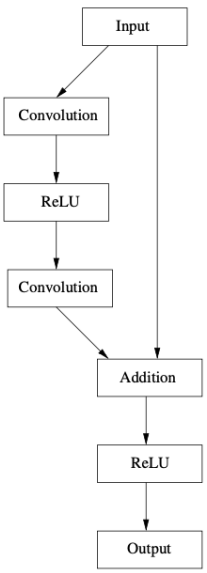

Figure 4: Residual Block used [2]

I created and tested different policy and value heads for the last layers of my network and decides to keep the following structures. The policy head applies a first 2D

convolution (32 filters, 3×3 kernels) followed by batch normalization and ReLU, then a second 1×1 convolution, again followed by batch normalization and ReLU. The value head applies a 2D convolution (3 filters, 3×3 kernel), followed by batch normalization and ReLU, then a flatten layer and a dense layer with sigmoid activation to output the scalar value.

The second residual block network model, named ResNet2, we chose and adapted came from a tutorial of the implementation of AlphaZero from scratch, which code can be found in [5]. For the choice of the policy and the value head, I decided to use the same policy and value as proposed in [3]. However, the policy and value were not converging properly so I chose to add a convolutional layer at the beginning of each head to improve the expressiveness of the heads and allow better feature extraction before the final layers, which helped stabilize training and improved the results.

Talking about the model details, all the layers are regularized using L2 regularization with a coefficient of 1e-4. I also managed to play on the hyperparameters of the model such as the number of residual blocks, the batch size, the learning rate and the loss weights.

The next plots give the evolution of the accuracy and loss during training and validation. The two models were trained on 300 epochs, with the same learning rate, batch size and loss weights (see notebook), in order to compare them.

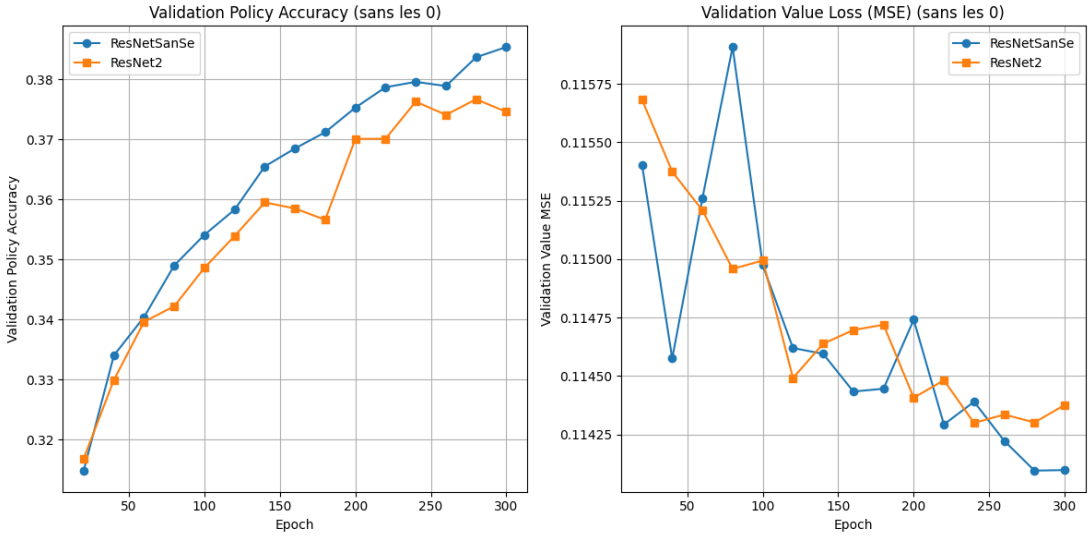

Figure 5: The 2 residual networks comparison on validation set

We can observe that **ResNetSanSe** manages to get a policy accuracy of **38.5%** on the validation set but both models to improve much the validation value mean square error during training, reaching **0.1140** for ResNetSanSe.

After studying these residual networks for the project, I decided to shift my focus to Vision Transformers, as they are now considered one of the most powerful and promis-

ing approaches in computer vision, particularly due to their ability to model long-range dependencies and outperform traditional convolutional architectures in many benchmarks.

4 Vision Transformers

Transformers were originally developed for Natural Language Processing tasks and have been successfully adapted to the field of computer vision. It is a class of neural networks that rely on self-attention mechanisms to effectively process and model complex data sequences. This self-attention allows the model to assess the relevance of each element within a sequence by evaluating its relationships with others. By employing multiple attention heads in parallel, Transformers can capture rich patterns, dependencies, and contextual information across the entire input. Vision Transformers extend this principle by analyzing interactions between different segments of an image, providing a novel framework for image understanding based on contextual relationships rather than local features alone.

4.1 The Efficient Former model

My experiments on the use of Vision Transformers for the project will be conducted using the EfficientFormer model, which was first introduced in [7]. It is a vision transformer architecture specifically tailored for low-latency inference on edge devices. It adopts a dimension-consistent design, where the network is divided into two main processing stages. The early stages operate on 4D feature tensors using convolutional MetaBlocks (MB4D) with local pooling as the token mixer. These blocks enable efficient extraction of low-level spatial features and maintain compatibility with hardware accelerators. The later stages transition to 3D representations and the use of Multi-Head Self-Attention (MHSA) within MB3D blocks to capture the global dependencies.

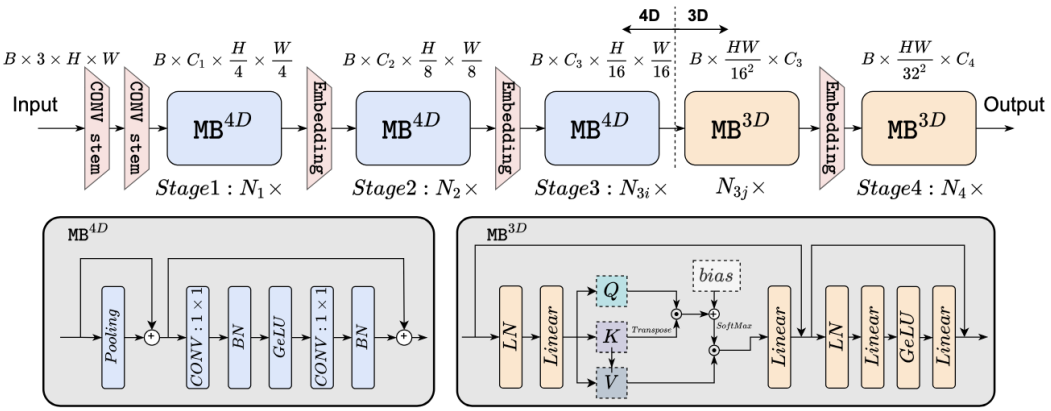

Figure 6: Overview of EfficientFormer architecture [7]

The network starts with a convolutional stem that replaces conventional large-kernel patch embeddings of traditional Vision transformers, which reduces computational overhead and preserves essential visual information. Between each stage, embedding layers perform spatial downsampling and channel projection, preparing the features for the next level of abstraction. By structuring the architecture in this hybrid 4D–3D manner, EfficientFormer avoids frequent reshaping operations.

To give an idea of the performances of this model structure, EfficientFormer-L1, presented in [7] as the smallest model of the EfficientFormer model collection, achieves over 79% top-1 accuracy on ImageNet-1K with only 1.6 ms inference latency on an iPhone 12, which outperforms lightweight CNNs like MobileNetV2 both in speed and accuracy.

The purpose of this study will thus be to analyze the performances of a simplified EfficientFormer model in our project and see what it can achieve with a 100 000 parameters constraint.

4.2 Adaptation for our project

4.2.1 Difference with the standard EfficientFormer

First of all, EfficientFormer recommends a relatively well-developed conv stem, mentioning a large repeated stride 2 convolution, to replace the traditional 16×16 patch embedding of ViTs. In my code, I opted for a single Conv2D for the stem, followed by BN+ReLU, which is simpler and lighter.
In EfficientFormer, the 4D blocks apply a "pool mixer" to mix spatial information. However, we use in my version a MB4D block that looks like an inverted residual block: it has a pointwise Conv (1×1), then a Depthwise Conv (3×3), then a pointwise Conv (1×1) again, with BatchNorm and ReLU, as well as a skip connection. In other words, we replace the token mixer with a simple convolutional mini block inspired by MobileNet.
For the 3D blocks of my model, we do not have the second normalization or the second MLP layer, which makes the block much lighter.
Finally, the policy and value head are also added, contrary to the classic EfficientFormer model which has a single head output.
I named this first model type: EfficientFormerRough.

4.2.2 Difference between EfficientFormerRough and EfficientFormerWay

I implemented a second type of simpler EfficientFormer model, named EfficientFormerWay. The new model adopt a clear two-stage design. It begins with a local feature extraction phase (using MB4D blocks) to process spatial patterns through convolutional operations, then transitions to a global reasoning phase (using MB3D blocks) with transformer-based attention. The revised MB4D block better emulates EfficientFormer's token mixer concept. Instead of relying on depthwise convolutions

(as in the old MB4D), the new version uses additive average pooling for local token mixing which preserves spatial relationships. In the MB3D blocks, we now apply layer normalization before the MHSA and MLP layers. The MLP now includes an expansion layer with GeLU activation, closely resembling the feed-forward networks in vision transformers. These changes make the new MB3D a more authentic transformer block, whereas the old version used a single dense layer without proper normalization or non-linearities. Finally, positional relationships on the Goban are spatially precise and would degrade with a reduction of the height and the width of the board. To address this, all layers maintain height and width dimensions through strict "same" padding. Hence, we avoid strided operations or pooling that shrink the grid. This idea was suggested in [8] and was applied in our model.

4.2.3 GOat model

This last model uses two ameliorations for MobileNets of the paper written by my professor [4].
First, I replaced the ReLU activation function by the Swish activation function in the MB4D blocks, the policy and the value head.

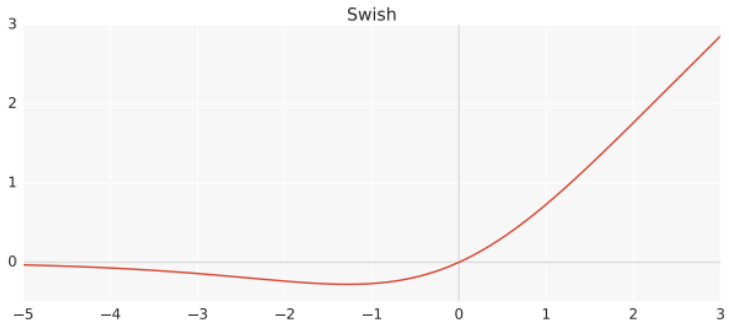

Figure 7: Swish activation function

Also, in the EfficientFormerWay model, I used an average pooling with pool_size equals to 3 for the local token mixer. I replaced this in GOat by using a new mixconv2D function which combines several depthwise convolutions of different kernel sizes (3x3 and 5x5) and concatenates their results. This enhances the network's ability to capture spatial information at different scales.

4.3 Results of the trainings

I made many trainings for the different vision transformers models. Those trainings took a lot of time, as it can be seen in 2.
Especially, I struggled to understand at the beginning that I put the value mean square error as the loss for the value learning whereas the binary cross entropy appeared to be way more efficient.
In this section we will present some results that we found interesting to show.

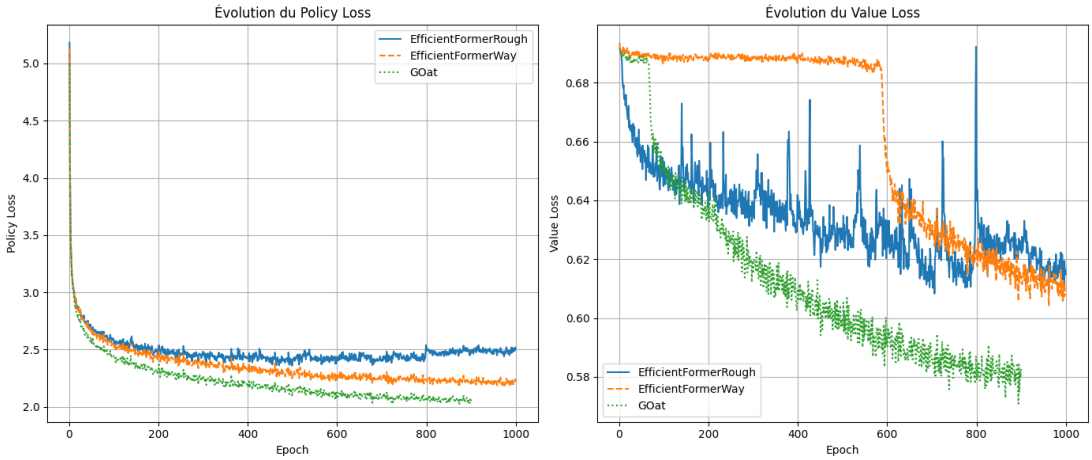

Figure 8: Training of the three models over the 1000 epochs

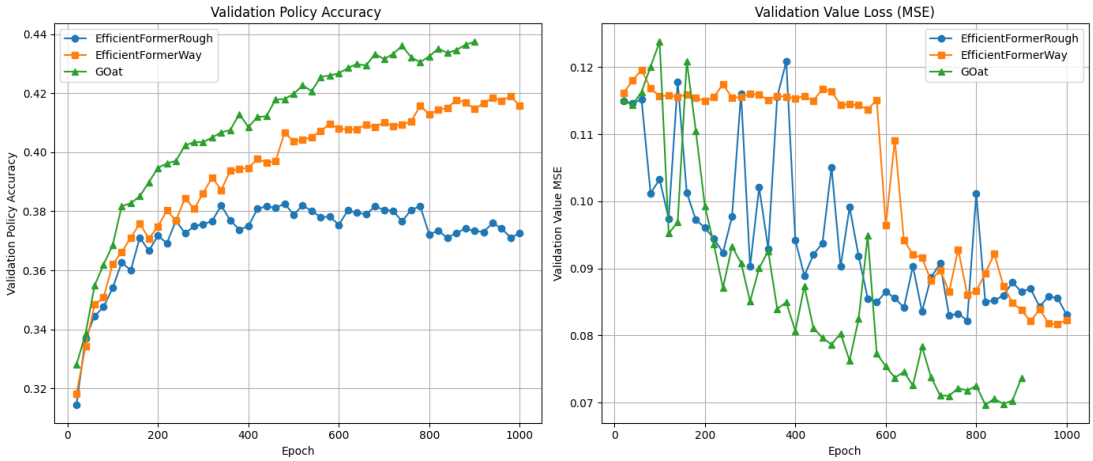

Figure 9: Validation of the three models over the 1000 epochs

The hyper-parameters for three models are the ones presented in the colab notebook.

The results that those plots give are very interesting. First of all, we can see that for the EfficientFormerRough model, the validation value loss is highly volatile. This could be due to the missing regularization in the value head convolutional layer, that can be seen in the notebook.

Also, we can see that the value loss dropped significantly sooner for GOat (at about 100 epochs) than EfficientFormerWay (at about 700 epochs), which could mean that the mixconv2D help the model to predict the value target quickly and efficiently.

EfficientFormerWay reached at the end of the training about **41.9%** accuracy on the validation tests policy and **0.082** for the value MSE.

GOat reached at the end of the training about **43.8%** policy accuracy on the validation sets and a value MSE of **0.070**.

However, even if the EfficientFormerWay model, with his high accuracy and low value MSE, looked to perform well with the Katago data, it had bad results on the class tournament. Indeed, it participated twice in the competition and was not better than the

ResNet model of my teacher. This could be linked to a bad robustness of the model or more technically maybe to a bad serialization of the model in a .h5 file.

5 Mobile Networks

5.1 The MobileNetv3 model

MobileNetV3 is the third evolution of the MobileNet architectures, originally designed by Google to meet the need for image classification models that are both high-performing and lightweight. After the first version, named MobileNetV1, which introduced depth wise-separable convolutions, a technique used to significantly reduce the number of parameters, and MobileNetV2, which added the concept of the "inverted residual" to boost network efficiency, MobileNetV3 goes even further by combining new building blocks with Neural Architecture Search (NAS).

MobileNetv3 uses a new architecture block named "Squeeze-and-Excitation" (SE) blocks. These blocks assign different "weights" to each image channel which helps the network to better capture essential features with good computation and memory budget. MobileNetV3 also introduces the h-swish (hard-swish) activation function, a streamlined version of swish that computes more quickly by replacing the sigmoid function with a ReLU6-based activation.

Finally, it is also important to mention the CBAM blocks, which are used in my last model MobileNet3_B. CBAM (Convolutional Block Attention Module) is an attention mechanism that enhances feature representations in convolutional neural networks. It focuses on both what and where to attend; it operates in two stages: first, channel attention highlights the most relevant feature channels, then spatial attention refines the important regions within those channels. By sequentially applying these two attention steps, CBAM boosts the network's ability to capture meaningful details. More details of the CBAM can be found in [11].

5.2 Implementation and results

The code we have used have been adapted from [1] for the MobileNetv3 architecture and from [6] for the implementation of the CBAM block.
We trained two models : MobileNet3_A that implements MobileNetv3 minimized architecture but with mixconv2D operations (the same than for GOat) instead of the depthwise convolution of the bottleneck block. MobileNet3_B is equivalent to MobileNet3_A but with a CBAM block replacing the squeeze and excitation block. The results of the training are given above :

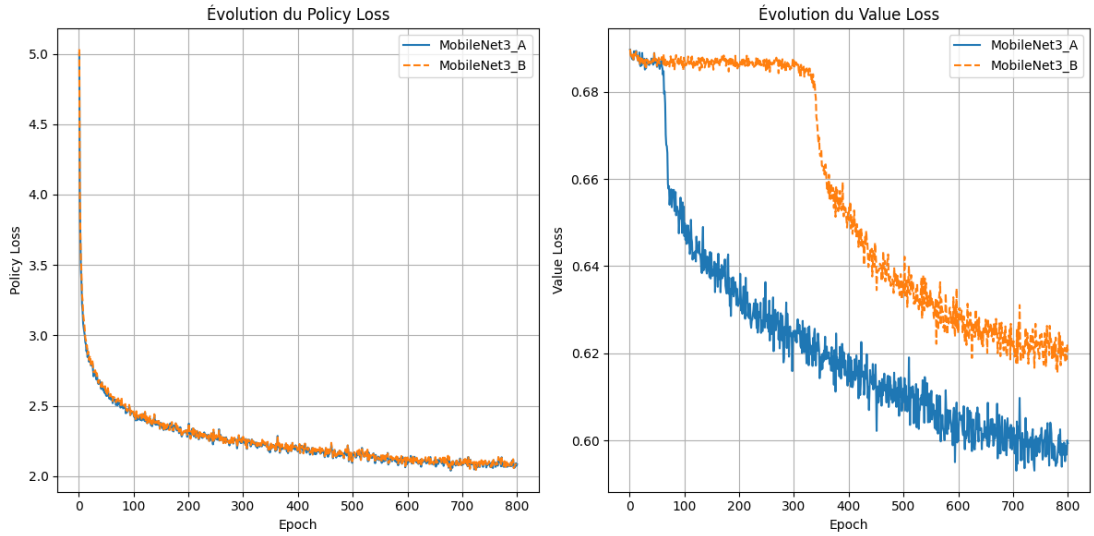

Figure 10: Training of the two models over 800 epochs

It seems that the CBAM block does not have a significant impact for a 100 000 parameters model. It even has worse results for the value predictions than MobileNet3_A.
This last model, **MobileNet3_A**, trained on 800 epochs, shows an accuracy of **0.429** for the validation policy and a value MSE of **0.075** on the validation set.
MobileNet3_A participates to the final tournament under the name of **de_Larouziere_3_A** and **MobileNet3_B** under the name of **de_Larouziere_3_B**.

6 Conclusion

In conclusion, this project enabled me to apply the principles and architectures we had studied in class. For weeks, I juggled between testing new architectures and optimizing the hyperparameters of our models.
I also chose to focus globally on Vision Transformers and more specifically Efficient Former. The result during the tournament was not necessarily what I expected for EfficientFormerWay, even if the training and validation were very good. With a week to spare, I would have loved to present my GOat model for the tournament, but it had serialization problems when loading its h5 file, so I wasn't able to test it in time.
More generally, having a tournament between the students was great fun, and helped motivate us during the months of the project.

References

[1] arawxx. Mobilenetv3. https://github.com/arawxx/MobileNetV3. URL https://github.com/arawxx/MobileNetV3. GitHub repository.

[2] Tristan Cazenave. Residual networks for computer go. *Computers and Games (CG 2018)*, pages 25–34, 2018. doi: 10.1007/978-3-030-04891-4_3. URL https://www.lamsade.dauphine.fr/~cazenave/papers/resnet.pdf.

[3] Tristan Cazenave. Mobile networks for computer go. *arXiv preprint arXiv:2008.10080*, 2020. URL https://arxiv.org/abs/2008.10080.

[4] Tristan Cazenave, Julien Sentuc, and Mathurin Videau. Cosine annealing, mixnet and swish activation for computer go. In *Advances in Computer Games: 17th International Conference, ACG 2021, Virtual Event, November 23–25, 2021, Revised Selected Papers*, volume 13262 of *Lecture Notes in Computer Science*, pages 53–60. Springer, 2022. doi: 10.1007/978-3-031-11488-5_5. URL https://www.lamsade.dauphine.fr/~cazenave/papers/CosineAnnealingMixnetAndSwishActivationForComputerGo.pdf.

[5] Robert Foerster. Alphazerofromscratch. https://github.com/foersterrobert/AlphaZeroFromScratch, 2021. URL https://github.com/foersterrobert/AlphaZeroFromScratch. GitHub repository.

[6] Byung Soo Ko. Cbam-tensorflow. https://github.com/kobiso/CBAM-tensorflow. URL https://github.com/kobiso/CBAM-tensorflow. GitHub repository.

[7] Yanyu Li, Geng Yuan, Yang Wen, Eric Hu, Georgios Evangelidis, Sergey Tulyakov, Yanzhi Wang, and Jian Ren. Efficientformer: Vision transformers at mobilenet speed. *arXiv preprint arXiv:2206.01191*, 2022. URL https://arxiv.org/abs/2206.01191.

[8] Amani Sagri, Tristan Cazenave, Jérôme Arjonilla, and Abdallah Saffidine. Vision transformers for computer go. *arXiv preprint arXiv:2309.12675*, 2023. URL https://arxiv.org/abs/2309.12675.

[9] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game

of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016. doi: 10.1038/nature16961. URL https://www.nature.com/articles/nature16961.

[10] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017. doi: 10.1038/nature24270. URL https://www.nature.com/articles/nature24270.

[11] Sanghyun Woo, Jongchan Park, Joon-Young Lee, and In So Kweon. Cbam: Convolutional block attention module. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 3–19, September 2018. doi: 10.1007/978-3-030-01234-2_1. URL https://arxiv.org/abs/1807.06521.