Deep Learning - Go Project IASD 2022-23

Adrien Golebiewski - Lucas Pereira Fernandes

Contents

1 Introduction

Programming a go player is considered a much more difficult problem than for other games, such as chess, because of a much larger number of possible combinations. This makes it extremely complex to use traditional methods such as exhaustive search. To teach a machine to play this game it is therefore extremely expensive to calculate all possible moves and select the one that is judged to be the best. This is why this game lends itself very well to deep learning networks.

The goal of this project is to train a network for playing the game of Go. In order to be fair about training ressources the number of parameters for the networks you submit must be lower than 100 000. This greatly reduced maximum number of parameters makes the project more complex and forces us to exploit the potential of all Deep Learning architectures

The data used for training comes from the Katago Go program self played games. There are 1 000 000 different games in total in the training set. The input data is composed of 31 19x19 planes (color to play, ladders, current state on two planes, two previous states on four planes). The output targets are the policy (a vector of size 361 with 1.0 for the move played, 0.0 for the other moves), and the value (close to 1.0 if White wins, close to 0.0 if Black wins).

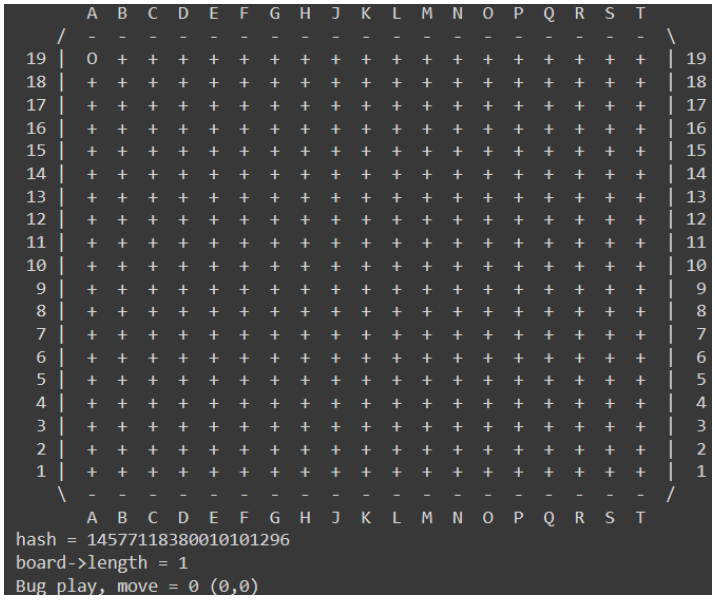


Figure 1: Schema of the GO game plane

In this report, we present our strategy to address the problem and the optimal neural architecture found in our research.

2 Overview of our strategy

Our goal was to build on the architectures seen in class and proposed through papers from the Deep Learning Course webpage. In a progressive order of complexity, several architectures, ranging from simple CNNs to MobileNetV2 using depthwise separable convolutions :

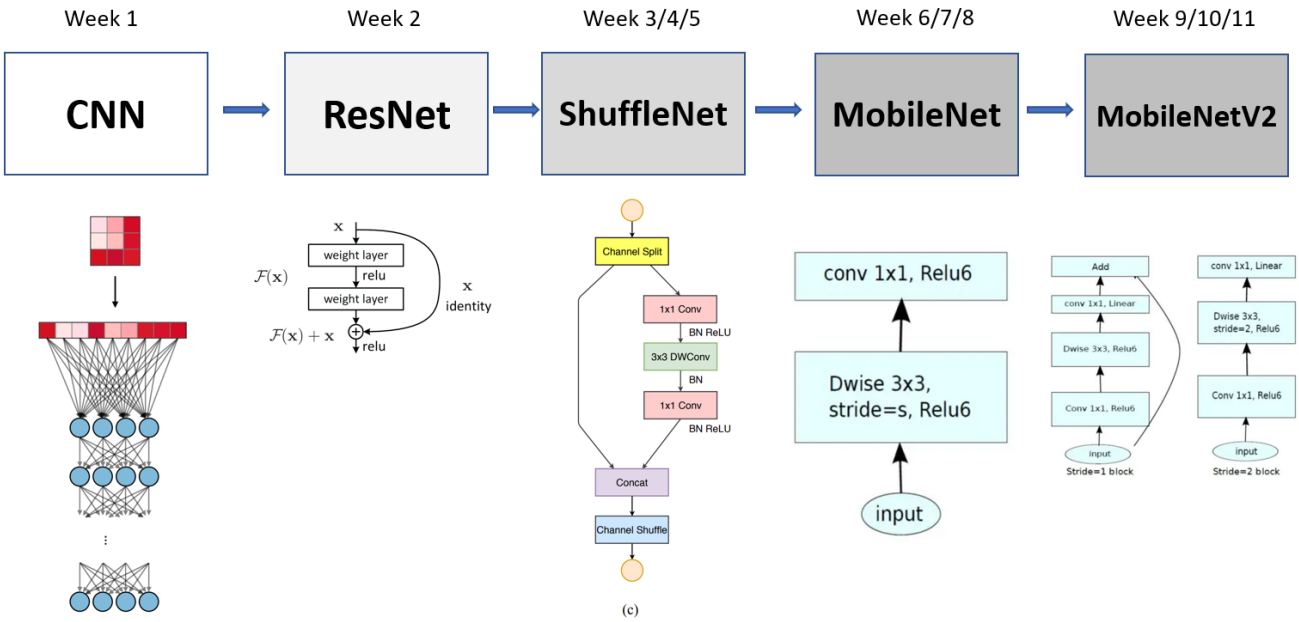


Figure 2: Planning project

Through this section of the report, the idea is not to present or describe all the tested architectures. Most of them have been taken from the course documents and therefore do not need to be described in detail.

The issue is more to present our strategy and the different intuitions that we had and that led us to our optimal architecture.

2.1 Convolutional Networks

In image analysis as well as in games, convolutional networks are widely used. They are particularly efficient for pattern recognition and have proven to be effective for learning the game of Go. We have therefore first implemented a simple network composed of convolution layers. We noticed after our first tests that this network was already quite efficient and learned the data well.

The first implemented network was very similar to the test.h5 network given as an example at the beginning of the tournament. We simply added layers and increased the number of parameters to improve the performance of the network. The results are interesting but the network seems to stabilize towards a validation loss around 15.

```
input = keras.Input(shape=(19, 19, planes), name='board')
x = layers.Conv2D(32, 3, activation='relu', padding='same')(input)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)
x = layers.Conv2D(32, 3, activation='relu', padding='same')(x)

policy_head = layers.Conv2D(1, 1, activation='swish', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
policy_head = layers.Flatten()(policy_head)
policy_head = layers.Activation('softmax', name='policy')(policy_head)
value_head = layers.GlobalAveragePooling2D()(x)
value_head = layers.Dense(50, activation='swish', kernel_regularizer=regularizers.l2(0.0001))(value_head)
value_head = layers.Dense(1, activation='sigmoid', name='value', kernel_regularizer=regularizers.l2(0.0001))(value_head)
```

Figure 3: Version 1 - CNN

In order **to make training of artificial neural networks faster and more stable**, we applied **Batch Normalization** with a normalization of our layers' inputs by re-centering and re-scaling.

```
input = keras.Input(shape=(19, 19, planes), name='board')
x = layers.Conv2D(32, 3, padding='same',
                  kernel_regularizer=regularizers.l2(0.01))(input)
x = layers.BatchNormalization(axis=-1)(x)
x = layers.Activation('relu')(x)

x = layers.Conv2D(128, 3, padding='same',
                  kernel_regularizer=regularizers.l2(0.01))(x)
x = layers.BatchNormalization(axis=-1)(x)
x = layers.Activation('relu')(x)
x = layers.Dropout(0.2)(x)
```

Figure 4: Version 2 - CNN with Batch Normalization

The results are more interesting but the loss was still too high to consider that the network could play autonomously. We have therefore chosen to keep this convolutional network architecture but **adding some modifications to improve the performances**.

2.2 Residual Networks

Residual networks are based on the principle of **reusing data from previous layers**. One of the reasons for skipping layers is to avoid the problem of gradient disappearance, by reusing the weights of a previous layer. So we used the same architecture as for the previous network but reintroducing at each step the weights of the previous layer.

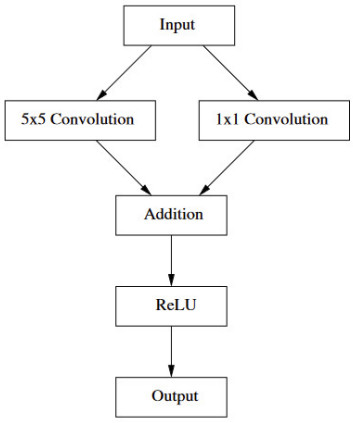


Fig. 3. The residual input layer for computer Go.

The schema above comes from the following article "Residual Networks for Computer Go - T.Cazenave" from which we were inspired to design the network We have observed better results than for the simple convolutional network **but the loss value remains high and the network can still be greatly improved**.

It was therefore necessary to innovate and test new architectures.

2.3 Mobile Net

It is the reading of Mobile Networks For Computer Go paper (T.Cazenave) and the implementation of the MobileNet network as described in the appendix of the article that allows to obtain first good results (under the constraint of 100 000 parameters) and to launch the exploration of the parameters.

This MobileNet architecture is based on the **Depthwise Separate Convolution** proposed by Google in April 2017. This new convolution strategy allowed us to optimize the convolution by making its **execution faster, less memory intensive and therefore less power consuming** than before. And that's while maintaining good accuracy.

The **depth-wise convolutions are used to apply a single filter into each input channel**. This is different from a standard convolution in which the filters are applied to all of the input chan-

```
def res_layer(layer_input):
  with tf.device('/device:GPU:0'):
    x = layers.Conv2D(filters, 5, padding='same')(layer_input)
    x1 = layers.Conv2D(filters, 1, padding='same')(layer_input)
    x = layers.add([x,x1])
    x = layers.Activation('relu')(x)
  return x

res_layers = 3

with tf.device('/device:GPU:0'):
  input = keras.Input(shape=(19, 19, planes), name='board')
  x = res_layer(input)
  for i in range(res_layers):
    x = res_layer(x)

  policy_head = layers.Conv2D(1, 1, activation='swish', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
  policy_head = layers.Flatten()(policy_head)
  policy_head = layers.Activation('softmax', name='policy')(policy_head)
  value_head = layers.GlobalAveragePooling2D()(x)
  value_head = layers.Dense(50, activation='swish', kernel_regularizer=regularizers.l2(0.0001))(value_head)
  value_head = layers.Dense(1, activation='sigmoid', name='value', kernel_regularizer=regularizers.l2(0.0001))(value_head)
```

Figure 5: Resnet architecture for Go

nels.This architecture is also characterized by a second layer is a 1×1 convolution, called a **pointwise convolution**, which is responsible for building new features through computing linear combinations of the input channels.

```
def standardconv(x):
  x = layers.Conv2D(filters= 32, kernel_size= (1,1), use_bias= False)(x)
  x = layers.BatchNormalization()(x)
  x = layers.Activation('swish')(x)
  return x

def residual_block(x, strides_depthwise: int, filter_pointwise: int, expansion: int):
  x1 = pointwiseconv(x, filters = filter_pointwise * expansion, linear = False)
  x1 = depthwiseconv(x1, strides = strides_depthwise)
  x1 = pointwiseconv(x1, filters = filter_pointwise, linear = True)
  if strides_depthwise == 1 and x.shape[-1] == filter_pointwise:
    x1 = SE_Block(x1, filter_pointwise)
    return layers.add([x1,x])
  else:
    return x1

def bottleneck_block(x, s: int, c: int, t: int, n: int):
  '''
    s : strides
    c : output channels
    t : expansion factor
    n : repeat
  '''
  x = inverted_residual_block(x, strides_depthwise= s, filter_pointwise= c, expansion= t)
  for i in range(n-1):
    x = inverted_residual_block(x, strides_depthwise= 1, filter_pointwise= c, expansion= t)
  return x
```

Figure 6: Mobilenet architecture

With this architecture and so the factorization of convolution , **the accuracy gain (more than 0.10 percent) and the decrease of our mse loss were real**. With 32 bottleneck blocks, an expansion to 200 filters, a batch size equal to 128 and a decrease of the learning rate, we obtained a very decent policy accuracy close to 40 percent.

2.4 Shuffle Net

Then, we have wanted to continue our research by testing other architectures based on depth wise convolutions.

We thus read and appropriated a first article **"ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices"**.

We learnt that the overall **ShuffleNet architecture is composed of a stack of ShuffleNet units grouped into three stages**. The first stage controls the connection sparsity of pointwise convolutions. At the same time, the second is assigned to be different numbers, so the output channels can be computed and evaluated to ensure the total computational costs are approximately the same

The important building block here in this case, is the channel shuffle layer which "shuffles" the order of the channels among groups in grouped convolution. Without channel shuffle, the outputs of grouped convolutions are never exploited among groups, resulting in the degradation of accuracy.

```python
def bottleneck_block(tensor, expand=96, squeeze=16):
  x = gconv(tensor, channels=expand, groups=4)
  x = BatchNormalization()(x)
  x = ReLU()(x)
  x = channel_shuffle(x, groups=4)
  x = DepthwiseConv2D(kernel_size=3, padding='same')(x)
  x = BatchNormalization()(x)
  x = gconv(x, channels=squeeze, groups=4)
  x = BatchNormalization()(x)
  x = Add()([tensor, x])
  output = ReLU()(x)
  return output
```

(a) Bottleneck function

```python
input = keras.Input(shape=(19, 19, planes), name='board')
x = Conv2D(trunk, 1, padding='same', kernel_regularizer=regularizers.l2(0.0001))(input)
x = BatchNormalization()(x)
x = ReLU()(x)

for i in range (blocks):
  x = bottleneck_block (x, filters, trunk)
policy_head = Conv2D(1, 1, activation='relu', padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
policy_head = Flatten()(policy_head)
policy_head = Activation('softmax', name='policy')(policy_head)
value_head = GlobalAveragePooling2D()(x)
value_head = Dense(50, activation='relu', kernel_regularizer=regularizers.l2(0.0001))(value_head)
value_head = Dense(1, activation='sigmoid', name='value', kernel_regularizer=regularizers.l2(0.0001))(value_head)
```

(b) Shuffle net arcitecture

In terms of performance, the results obtained with shuffle-net **were almost identical to those of MobileNet**.

2.5 MobileNet v2

The latest architecture tested in the tournament is the **MobileNetv2**. Based on the paper "MobileNetV2: Inverted Residuals and Linear Bottlenecks", we learnt that is a convolutional neural network architecture that seeks to perform well on mobile devices.

It is based on an inverted residual structure where the residual connections are between the bottleneck layers.

This is our optimal architecture with which we obtained the best metrics scores. The next section will describe its exact architecture in more detail and its metrics results.

2.6 Results Analysis

The metrics that we have for assessing our neural networks were the final loss of the policy and of the value, the final categorical accuracy of the policy and the final mean square error of the value.

We compared our models' performance based on **the policy categorical accuracy** and **MSE loss**. We obtained the results: obtained in the following table and in Figures 8 and 9.

Performances of ours models (X epochs)	CNN	ResNet	ShuffleNet	MobileNet	MobileNetv2
Accuracy	29.52%	35.71%	33.27%	38.47%	42.21%
MSE	0.1827	0.1244	0.1431	0.1129	0.081

After training for the first 300 epochs, all models performed at least 29.52% accuracy, but MobileNetV2 outperformed the others with 42.21% accuracy.

Thanks to these four metrics we assessed our different networks in order to choose the better for the competition. According to the results, MobileNetV2 is, according to us, the most adapted to our problem and especially the most powerful.

Performance of our best model MobileNetV2 :

This is the model we uploaded on the drive. And that we will detail further in the next section of the report :

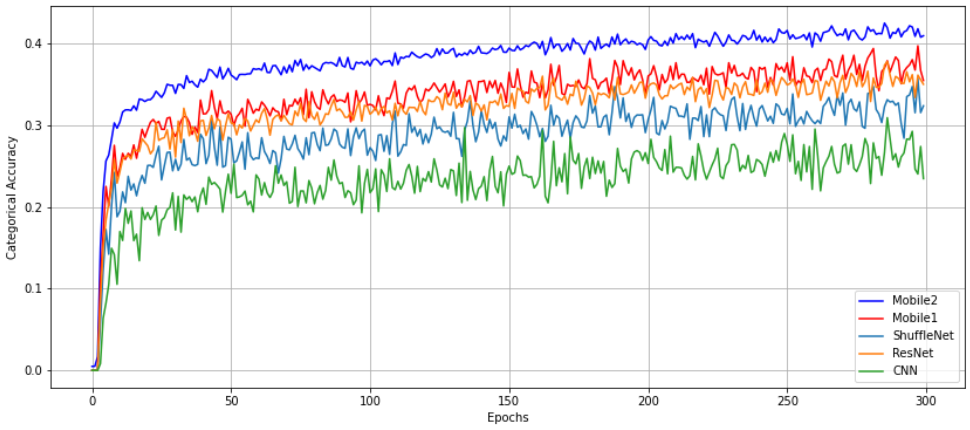


Figure 8: Model validation accuracy

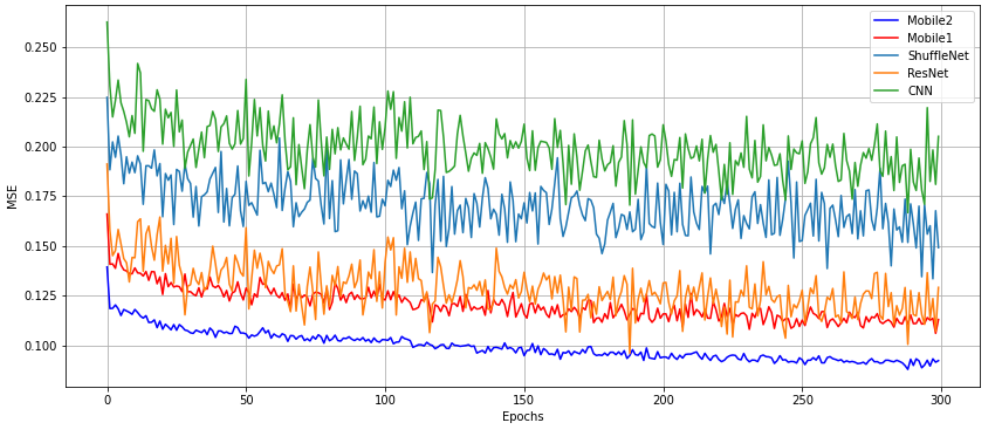


Figure 9: Models MSE

3 Focus on our optimal solution

3.1 Architecture and comments

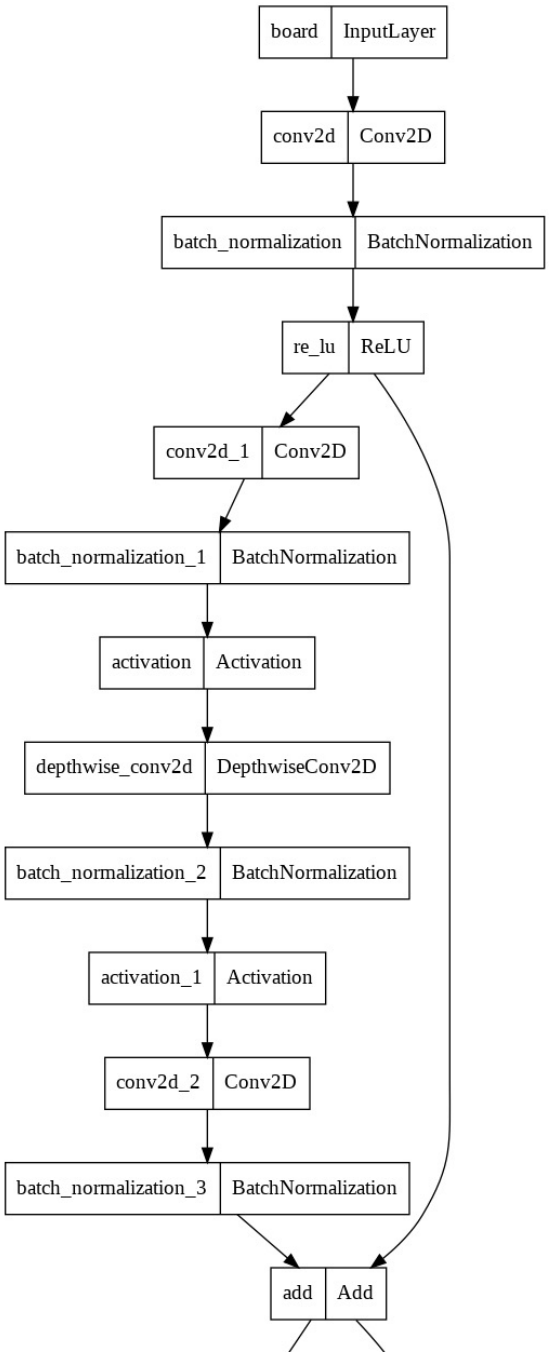
In Figure 10a, we can see the beginning of the architecture of our optimal solution (mobilenetV2). The left branch is a block that is repeated 33 times, as in Figure 10b.

The principle our MobileNetV2 architecture is to have blocks as in residual networks where the input of a block is added to its output. But instead of usual convolutional layers in the block they use depthwise convolutions

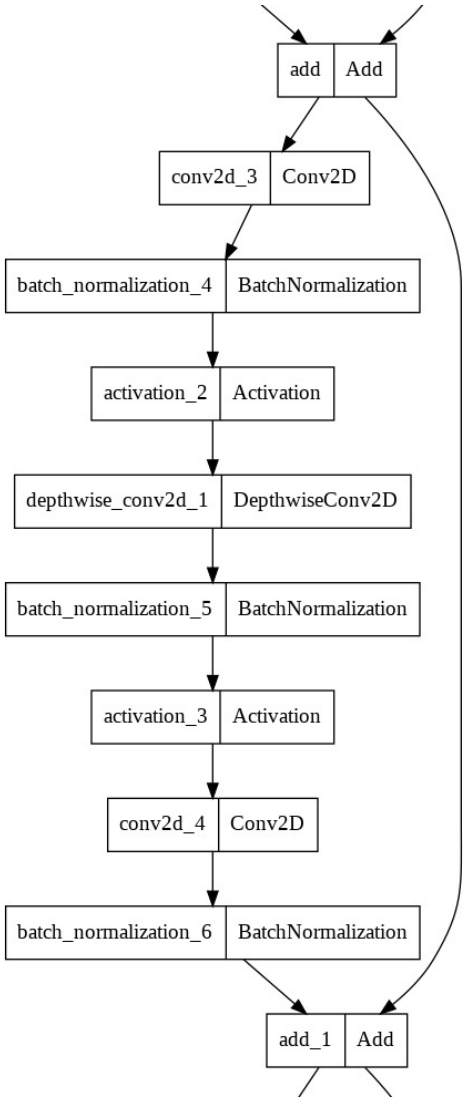
The intermediate expansion layer uses lightweight depthwise convolutions to filter features as a source of non-linearity. Our depthwise separable convolution is made of two operations: a depthwise convolution and a pointwise convolution. As a whole, the architecture of MobileNetV2 contains the initial fully convolution layer with 64 filters, followed by 19 residual bottleneck layers.

In the MobilenetV1 architecture, the pointwise convolution either kept the number of channels the same or doubled them. **In V2 it does the opposite: it makes the number of channels smaller**. This is why this layer is now known as the projection layer, it projects data with a high number of dimensions (channels) into a tensor with a much lower number of dimensions.

MobileNetV2	loss	Policy loss	Value loss	policy categorical accuracy	Value mse
Metrics	5.4711	2.2051	0.6216	42.21%	0.081



(a) MobileNetV2 beginning.



(b) MobileNetV2 fundamental block.

The second new thing in MobileNet V2's building block is **the residual connection**. This works just like in ResNet (presented in the previous section) and exists to help with the flow of gradients through the network.Notice that the residual connection is only used when the number of channels going into the block is the same as the number of channels coming out of it, which is not always the case as every few blocks the output channels are increased.

3.2 Model training and optimizers

Below are some of the contextual elements that allowed us to obtain a high-performance architecture :

- In order to fit well into memory, **we used a batch size of 64 and 64 filters firstly**. Then, given the different game configurations are extremely numerous, we needed to train the network on a lot of examples in order to learn the best way. We are going to use the **dynamicBatch function** which allows us to randomly select 100 000 examples out of the few millions available. So we kep the network but loop on the batches to train on as many examples as possible. In order not to overlearn we limited the epochs to 5 per batch.

- We decided to use **the relu and swish activation function f(x)** as replacement for the sigmoid activation function. Indeed, the relu and swish activation outperforms the others on our deep networks.

- Our default **learning rate** is 0.0005 for the first 50 epochs, then 0.0005 for 100 next epochs and finally 0.000005 for the last epochs.

- We observed on some of our tests, indications of a potential overfitting by comparing the performance with the training data and its performance with the test data. In order to make the learning of our Deep Learning model more robust and to counteract the overfitting, we applied a **l2 (Sum of the squared weights) penalty on the kernel of some layers**. The keras regularization prevents the over-fitting penalizing model from containing large weights.

- As underlined in these paper "Mobile networks for computer go" and "MobileNetV2: Inverted Residuals and Linear Bottlenecks", using **Global Average Pooling** in the value head leads to better results than using the AlphaZero value head. Each channel is averaged among its whole 19x19 plane leading to a vector of size equal to the number of channels. Moreover, in our case, it spares more than 10 000 parameters over the 100 000 allowed, which is why we adopted this method and transformed the value head as follows:

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

value_head = layers.GlobalAveragePooling2D()(x)
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)
```

Figure 11: Value Head with Global Average Pooling

- To get better results, we **switch from a SGD optimizer to Adam** (with starting learning rate tuned to 0.005), giving us 0.43 accuracy. Indeed, we observed that using Stochastic Gradient Descent (SGD), the loss value is not decreasing for a large number of epochs with a stabilizing effect at the end. Based on stochastic gradient descent to solve non-convex problems faster while using fewer resources, Adam optimizer is most effective in extremely large data sets by keeping the gradients "tighter" over many learning iterations. **We notice in our case, a much faster gradient decay of our MSE loss**.

4 Conclusion and perspectives

Throughout this project, we explored different neural network architectures (ranging from the simplest to the most complex architectures) applied to the learning of two-headed GO games (policy and value). We explored different parameters concerning both the architecture (e.g., number of blocks, filters, batch size) and the learning process (type of optimizer, weighting schemes for value and loss, learning rate, learning rate adapter). From a more advanced point of view, we have also implemented the batch normalization method as well as techniques for the complex MobileNet, ShuffleNet architectures and even a mixed of this two methods at the end of the tournament (but without success).

The main part of the work was to **document and test the different possible architectures, the associated hyper-parameters and the choice of the optimizer**. It is thus through this long phase of research, comparison and tests that we allow us to define the MobileNetv2 method as being optimal (although the network can still be improved). **We did observe improvements by using MobileNet blocks with batch normalization and globalAveragePooling instead of Residual/Shuffle blocks, or swish instead of ReLU for our activation functions**.

Given a fixed two-headed parameterization (policy and value) and with the constraint of 100'000 parameters, **we felt that we had reached saturation of the learning process through the different options evaluated**. Thus, most of the alternatives evaluated to optimize the model either significantly reduced the performance or increased it incrementally and negligibly. **A substantial breakthrough would likely require more creativity in the CNN architecture and this is likely an active topic for academic research**.

For example it would have been relevant to exploit more in depth our mobile net **by adding channel attention to the network** or to better optimize tried our step size by testing a Multi-step division and a Cosine decay scheduler, as explained in the paper "Cosine Annealing, Mixnet and Swish Activation for Computer Go " (T.Cazenave, J.Sentuc).

References

[1] T. Cazenave, "Mobile networks for computer go" IEEE Transactions on Games, vol. 14, pp. 76–84, 2020.

[2] Xiangyu Zhang,Xinyu Zhou,Mengxiao Lin , P. (2020). ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices, 33, 6840-6851.

[3] T. Cazenave, "Improving model and search for computer go," in 2021 IEEE Conference on Games (CoG), pp. 1–8, 2021.

[4] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov (2019, July). MobileNetV2: Inverted Residuals and Linear Bottleneck (pp. 8162-8171). PMLR.

[5] Tristan Cazenave, Julien Sentuc, and Mathurin Videau (2021). Cosine Annealing, Mixnet and Swish Activation for Computer Go.