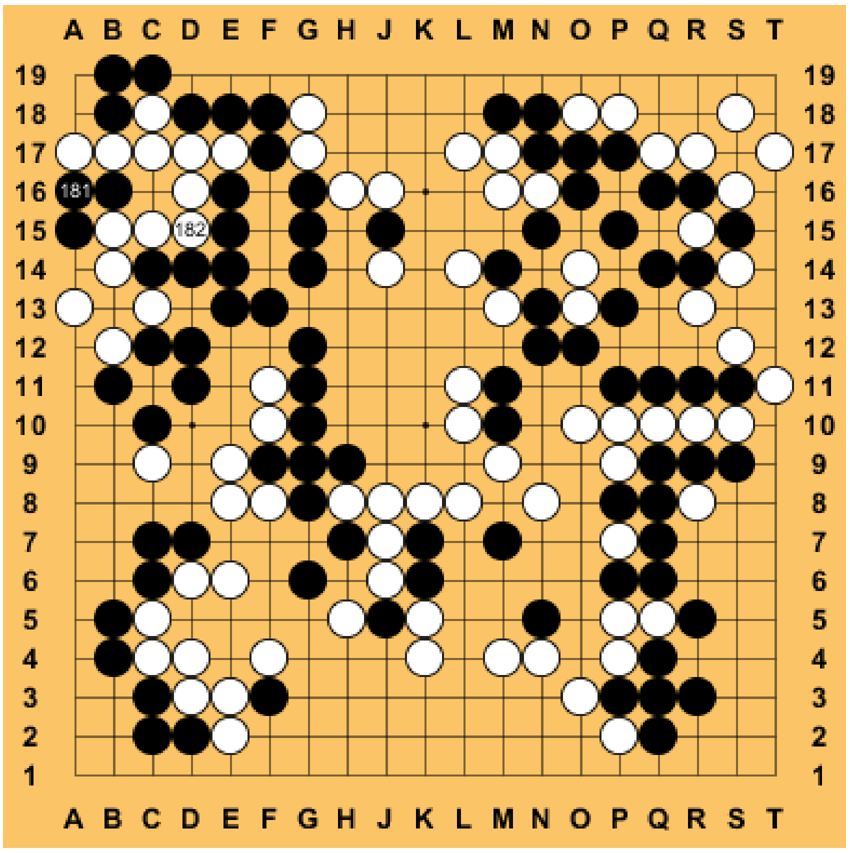
Deep Learning
Implementing state-of-the-art networks for the game of Go

Pierre Neveu

January 2025



Contents

1 Introduction **2**

1.1 Context 2

1.2 Global process of my implementations 2

2 ResNet **3**

3 MobileNet **5**

3.1 Naive implementation 5

3.2 Upgrades from the state of the art 5

3.3 Personnal experiments 7

3.3.1 Intuition and other 7

3.3.2 Architecture width and depth 7

3.4 Final model 8

A Appendix **9**

A.1 Code 9

A.1.1 Models 9

A.1.2 Schedulers 11

A.2 Tournaments scores 12

A.3 More Graphics 12

1 Introduction

1.1 Context

The goal of this project is to train a network for playing the game of Go. Each week, **a round robin tournament** will be organized with students networks. Each network will be used by a **PUCT engine** that takes 2 seconds of CPU time at each move to play in the tournament. In order to be fair with the training, the networks submitted must be around **100,000 parameters**.

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).

Games and data implementations are handled by a C++ code (/Game folder) provided by Tristan Cazenave. This projects focuses on building a network in a supervised-learning manner.

1.2 Global process of my implementations

In this report, I describe the steps I went through before to build my final model (Part 3.4).

I mostly used Tristan Cazenave's papers. I first implemented the Resnet network from [1] but then realized that the paper doesn't implement any specific value head. I moved on and train a first mobilenet v1 [9] which uses Global Max Pooling for the value head and no optimization of the architecture and training parameters. In the first tournament, the teacher's resnet model had a much better winning rate than my naive mobilenet. I concluded that the value is not to be neglected.

Therefore, I first focused on building a Resnet with a good value head [4], [3], in order to understand what makes a value head efficient.

Then, I implemented mobilenets v2 / v3 [7], [8] and their upgrades for computer go [12].

Because the final purpose is to compare each student network in a round robin tournament, and because we might all have a mobilenet at the end or something close to it, I based my final optimizations on my winrates in tournaments. Finally, Monte Carlo algorithms are used for the tournaments so winrates are not determinists. Therefore I compared my models winrates on more than one tournament and only between them as I didn't know what other students architectures were.

2 ResNet

Starting from [1] architecture and with a naive architecture for the value head, which basically consisted of flattenning the last residual block into a mlp with an output of size 1. I first started to search for the best width, depth and size of batch and used SGD for the optimizer with a default and fixed learning rate value. Then I compared with the dividing scheduler [1] (code here: A.1.2).

As explained in the introduction, I then focused on building a good value head as it plays an important role in the Monte Carlo search. Therefore, I compared 3 implementations (see **Appendix A.1.1** for the code) that consist in the resnet architecture with the value head chosen between **AlphaGo** [2], **Spatial Average Pooling** [3], or the KataGo Program using Global Average Pooling (dicussed in [10]). I used the following setup:

- **batch size = 32** because it appeared to be the most efficient.
- **40 filters, 3 blocks**
- **epoch = 25**. Regarding my first experiment, i considered 25 epochs was enough to evaluate and compare the 3 heads.
- **Optimizer = Adam** and **lr = 0.01** with no scheduler seemed to give better results.
- **BatchNorm layer** after each convolution layer (which obviously significantly improved the convergence speed).
- policy loss weight = 0.5 and value loss weight = 1.5

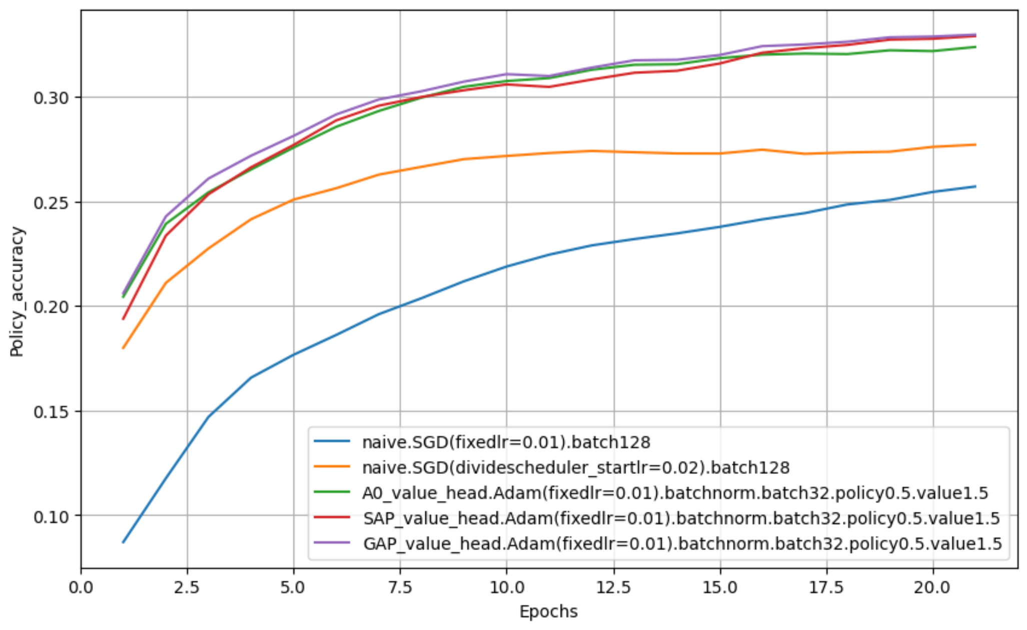


Figure 1: Resnets: **Policy accuracy** moving average

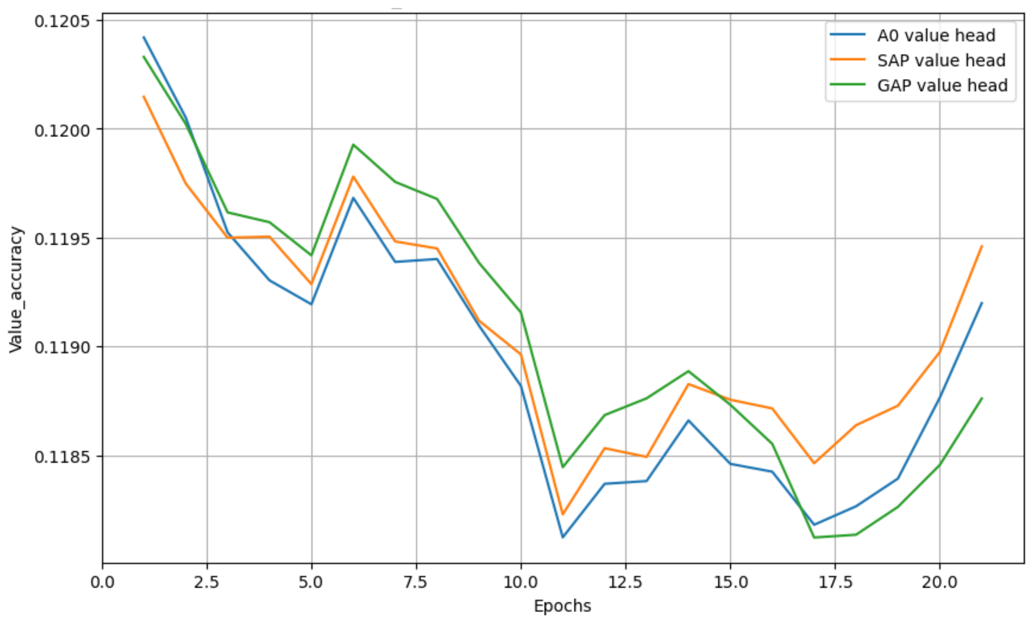


Figure 2: Resnets: **Value MSE** moving average

Regarding the results and considering discussions of [10], I understood that in order to **improve** the way our network learns the **value**, we must **upgrade the core of its architecture** (and not only the architecture of the value head) so that it can learn the global patterns from the input. This intuition is supported in figure 8 (from [10]) where we can see models like mobilenet with squeeze excitation blocks and Global Average Pooling in the head having better performance in the value thanks to their core architecture.

3 MobileNet

3.1 Naive implementation

Thanks to dephtwise convolution, mobilenet drastically reduces the number of parameters allowing us to increase the number of block. [9] gives us the baseline code for the mobilenet network. I first implemented it and just adapted the code to match the parameter limit (19 blocks of filters = 64, trunk = 32) and compared it with the resnet model.

3.2 Upgrades from the state of the art

Based on this model, [10] and [12] give us many improvements to implement:

1. In [10], model are trained on **mini batch size of 32**, which confirms our first choice.
2. **200 epochs** based on [9].
3. **19 blocks** of **filters = 64**, **trunk = 32**
4. [9] highlights the benefits of setting the **value loss weight to 4**. (We will not focus on this point, but I found that value weight = 4 does not only improve value mse, it doesn't significantly decrease policy accuracy. Tournaments results (A.2) highlight a better winrate with this setting for a given model
5. **Squeez-Excitation (SE) blocks** [10], [11]. They are inserted at the end of the mobilenet blocks and allow to build weights that give each feature map a different importance.
6. **Swish activation** is better than ReLU [12].
7. **Mix sized convolutions** (MixConv [12],[13]) help the model to learn different patterns. Basically, we change each depthwise convolution by two parallel (3, 3) and (5, 5) depthwise convolutions that we concatenate after.
8. **Cosine annealing scheduler** [12], [14]. I did not use it at first because it did not give me better results on 200 epochs. We will discuss it later.

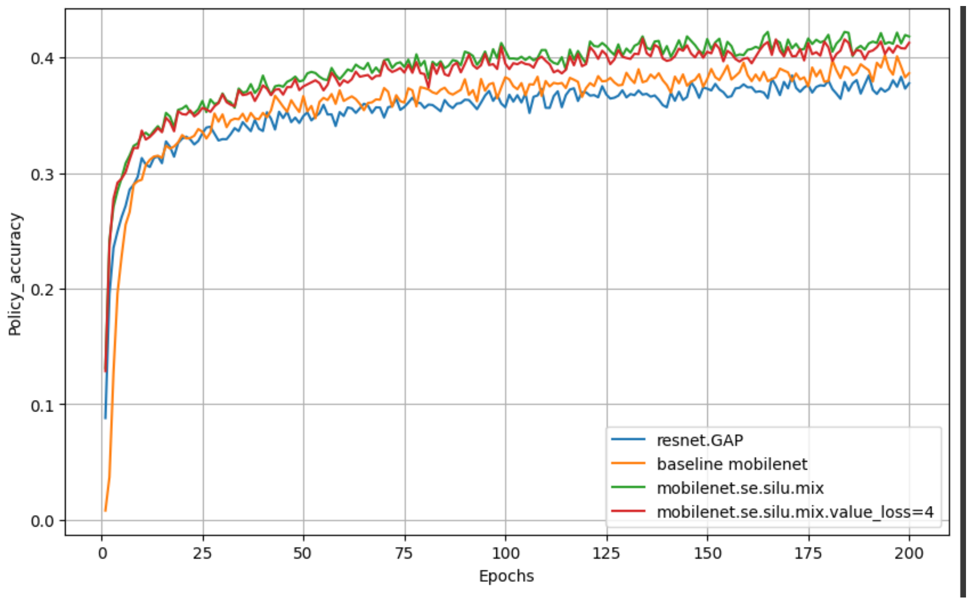


Figure 3: Resnet and Mobilenets: **Policy accuracy**

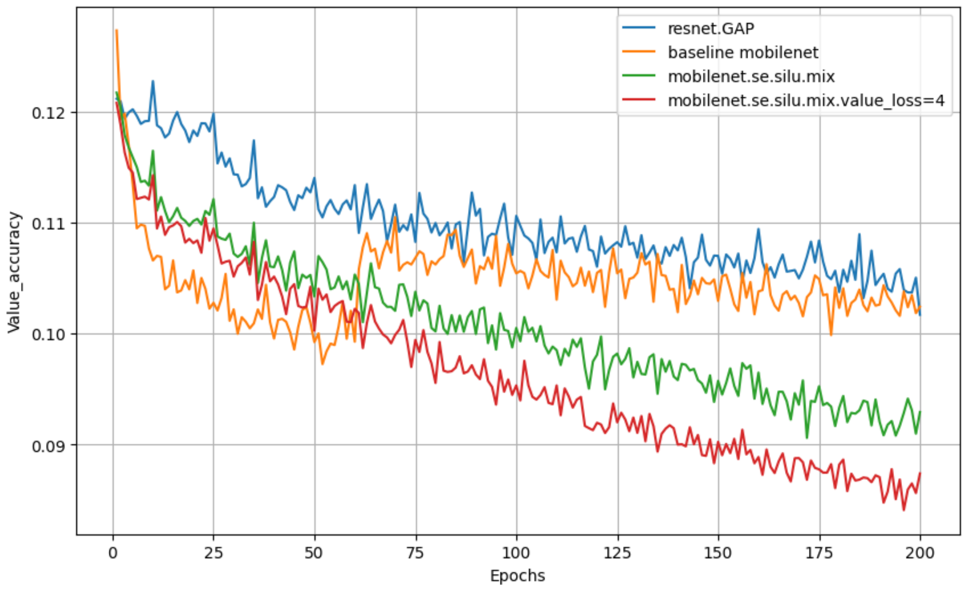


Figure 4: Resnet and Mobilenets**Value MSE**

These results highlight the capacity of mobilenet (and especially the one with upgrades) to better learn the value and to improve the policy by a few significant percentages. Moreover, it shows that it is not much of a tradeoff to raise the value loss weight to 4 as it does not really deteriorate the policy accuracy (see red and green curves). Tournaments (see 7) confirmed that a network with vw = 4 wins over the same network with vw = 2.

3.3 Personnal experiments

3.3.1 Intuition and other

At this point, I had implemented most of the state of the art but some variations are still to be explored. Here are some of my side experiments:

1. In Mobilenet V3 [8], one of the proposed improvement is to **move the SE block** just before the projection layer (= the mobilenet block's end convolution). However, because it is before the convolution which reduces the number of feature map, **it leads to increase the number of parameters** and we have to **decrease the number of block** by one to match the parameter limit. This **negatively affects the model performances**, see 9 and 10.
2. In validation, the value mse was very unstable. I thought it might be because the value is just hard to learn or the model was overfitting. In order to ensure that it was not overfitting, I decided to **add Dropout layers in the value head**. Based on the intuitions from [15] where authors study the impact of layers close to the output on the prediction, I also added dropout in the last layer. This didn't help to stabilize the value mse and **policy was deteriorated** as 11 and 12 show.
3. When applying a moving average with a window size of 50 epochs on the training curves of 200-300 epochs, I realized that the trend of the curves was not exactly converging and still could still improve. Therefore I **increased the number of epoch** until I found that the metrics converge around **700 epochs**. This allowed me to **improve the policy accuracy by 2% and the value mse by 0.05 units**.

3.3.2 Architecture width and depth

In [10], the author states that it is better to increase both width and depth when making the networks grow (on networks with a minimum of 1 million parameters). Therefore, I thought that when making the network shrink, it would also be better to balance width and depth. However I realized that **in our context, depth wins over width**. In this section I will give the performances of 3 different models I have trained. They go from deep and thin model to shallow and large models. **The values of the next table are to be read as follows: (Number of parameters, validation policy accuracy, validation value mse, winrate in the same tournament)**. **b** stands for number of blocks, **f** stands for number of filters in block input, **t** stands for number of filters in block output.

Table 1: (N params, pol. acc., val. mse, winrate) of different models dimensions

$b \setminus f, t$	32,16	64,16	64,48
31	**(98k, 44%, 0,075, 80%)**	x	x
16	x	(99k, 43,7%, 0.08, 78%)	x
8	x	x	(105k, 43,7%,0.077, 73%)

3.4 Final model

Finally, I managed to implement **cosine annealing scheduler** [12], [14]. It turned out that it was benefic on 800 epochs and **improve policy accuracy** by 0,5%-1% up **to 44,65%** and **value mse to 0.065**. Here are the cosine scheaduler parameters I used:

1. optimizer = Adam
2. epochs = 800
3. Starting lr = 0,002, ending lr = 0
4. Tmul = 1
5. Ti = epochs

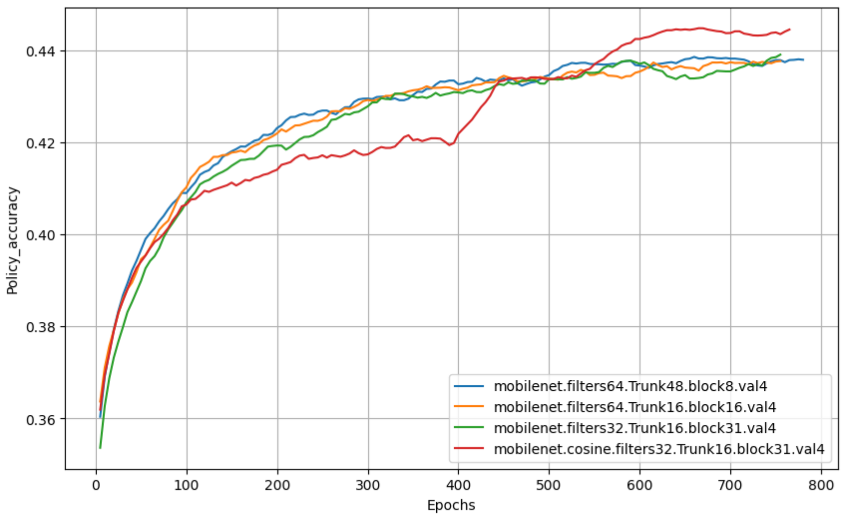


Figure 5: **Validation Policy accuracy**: final mobilenets - 50 epochs moving average

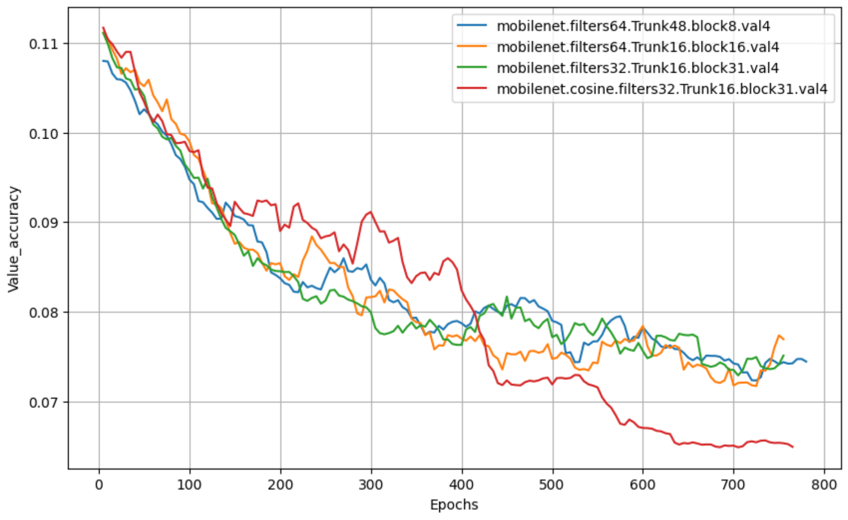


Figure 6: **Validation Value MSE**: final mobilenets - 50 epochs moving average

A Appendix

A.1 Code

A.1.1 Models

```python
def resnet(filters=filters, nb_layers=3):
    input = keras.Input(shape=(19, 19, 31))

    # Input residual layer
    small_kernel = layers.Conv2D(filters, (1, 1), padding='same')(input)
    large_kernel = layers.Conv2D(filters, (5, 5), padding='same')(input)
    x = layers.add([small_kernel, large_kernel])
    x = layers.ReLU()(x)

    # Residual block
    def residual_block(x, filters):
        residual = x

        x = layers.Conv2D(filters, (3, 3), use_bias = False, padding='same')(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)

        x = layers.Conv2D(filters, (3, 3), use_bias = False, padding='same')(x)
        x = layers.add([x, residual])
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
        return x

    # Add several residual blocks
    x = residual_block(x, filters)
    for i in range(nb_layers - 1):
      x = residual_block(x, filters)

    # Policy head: 1 x 19*19 grid for move probabilities
    policy_head = layers.Conv2D(1, 1, padding='same', use_bias = False, kernel_regularizer=regularizers.l2(0.0001))(x)
    policy_head = layers.BatchNormalization()(policy_head)
    policy_head = layers.Activation('relu')(policy_head)
    policy_head = layers.Flatten()(policy_head)
    policy_head = layers.Activation('softmax', name='policy')(policy_head)

        # AlphaGo Value Head
    # value_head = layers.Conv2D(1, (1, 1), padding="same", strides=1, use_bias=False)(x)
    # value_head = layers.BatchNormalization()(value_head)
    # value_head = layers.ReLU()(value_head)
    # value_head = layers.Flatten()(value_head)
    # value_head = layers.Dense(64, activation="relu")(value_head)
    # value_head = layers.Dense(1, activation="sigmoid", name="value", kernel_regularizer=regularizers.l2(0.0001))(value_head)

        # Value Head using Spatial Average Pooling
    # Use average pooling twice
    # value_head = layers.AveragePooling2D(pool_size=(2, 2), strides=2, padding='same')(x)
    # value_head = layers.ZeroPadding2D(padding=1)(value_head)  # Expand from 10x10 → 12x12
    # value_head = layers.AveragePooling2D(pool_size=(2, 2), strides=2, padding='same')(value_head)
    # value_head = layers.Conv2D(1, kernel_size=(1, 1), padding="valid", use_bias=False)(value_head)  # Final 6x6 filter
    # value_head = layers.Flatten()(value_head)
    # value_head = layers.Dense(50, use_bias=False, kernel_regularizer=regularizers.l2(0.0001))(value_head)
    # value_head = layers.BatchNormalization()(value_head)
    # value_head = layers.ReLU()(value_head)
    # value_head = layers.Dense(1, activation="sigmoid", name="value", kernel_regularizer=regularizers.l2(0.0001))(value_head)

        # Value Head using Global Average Pooling
    value_head = layers.GlobalAveragePooling2D()(x)
    value_head = layers.Dense(50, use_bias=False, kernel_regularizer=regularizers.l2(0.0001))(value_head)
    value_head = layers.BatchNormalization()(value_head)
    value_head = layers.ReLU()(value_head)
    value_head = layers.Dense(1, activation="sigmoid", name="value",
    ↪  kernel_regularizer=regularizers.l2(0.0001))(value_head)

    # Define the model with two outputs: policy and value
    model = keras.Model(inputs=input, outputs=[policy_head, value_head])

    # Model summary
    model.summary()

    # Compile model
    model.compile(
        # optimizer=tf.keras.optimizers.Adam(learning_rate = 0.2),
        optimizer="Adam",
        loss={'policy': 'categorical_crossentropy', 'value': 'binary_crossentropy'},
        loss_weights={'policy': 0.5, 'value': 1.5},
        metrics={'policy': 'categorical_accuracy', 'value': 'mse'}
    )
    return model

optimal_resnet = optimal_resnet(
    filters=32,
    nb_layers=4
)

def SE_Block(input_tensor, filters, ratio=16): # Ratio is generally 16
    se_shape = (1, 1, filters)
    se = layers.GlobalAveragePooling2D()(input_tensor)
    se = layers.Reshape(se_shape)(se)
    se = layers.Dense(filters // ratio, activation='silu', use_bias=False)(se)
    se = layers.Dense(filters, activation='sigmoid', use_bias=False)(se)
    x = layers.Multiply()([input_tensor, se])
    return x

def mixnet_block(x, expand=filters, squeeze=64):
    m = layers.Conv2D(expand, (1,1),
                      kernel_regularizer=regularizers.l2(0.0001),
                      use_bias=False)(x)
    m = layers.BatchNormalization()(m)
    m = layers.Activation("silu")(m)

    # MixNet: Use multiple kernel sizes
    m1 = layers.DepthwiseConv2D((3,3), padding="same",
                               kernel_regularizer=regularizers.l2(0.0001),
                               use_bias=False)(m)
    m2 = layers.DepthwiseConv2D((5,5), padding="same",
                               kernel_regularizer=regularizers.l2(0.0001),
                               use_bias=False)(m)
    m = layers.Concatenate()([m1, m2])

    m = layers.BatchNormalization()(m)
    m = layers.Activation("silu")(m)

    # SE block between DW conv and projection (Mobilenet V3)
    # m = SE_Block(m, 2*expand) # *2 because mix net concatenation

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

    # Add Squeeze-and-Excitation block
    m = SE_Block(m, squeeze)
    return layers.Add()([m, x])

def mobilenet(blocks, filters=256, trunk=64, policy_weight=1.0, value_weight=1.0):
    input = layers.Input(shape=(19, 19, 31), name="board")
    x = layers.Conv2D(trunk, 1, padding="same",
                      kernel_regularizer=regularizers.l2(0.0001))(input)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("silu")(x)

    for i in range(blocks):
        x = mixnet_block(x, filters, trunk)

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

    # Value Head
    value_head = layers.GlobalAveragePooling2D()(x)
    value_head = layers.Dense(64, activation="silu",
                              kernel_regularizer=regularizers.l2(0.0001))(value_head)
    value_head = layers.Dense(1, activation="sigmoid", name="value",
                              kernel_regularizer=regularizers.l2(0.0001))(value_head)

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

    model.summary()
    model.compile(
        optimizer=Adam(),
        loss={'policy': 'categorical_crossentropy', 'value': 'binary_crossentropy'},
        loss_weights={'policy': policy_weight, 'value': value_weight},
        metrics={'policy': 'categorical_accuracy', 'value': 'mse'}
    )
    return model
```

A.1.2 Schedulers

```python
def cosine_annealing(epoch, curr_epoch, n_max = 0.005, n_min = 0):
    Ti = epochs,
    Tcur = curr_epoch % Ti
    return n_min + (n_max - n_min) * (1 + np.cos((Tcur/Ti)*np.pi)) / 2

class ResnetLearningRateScheduler(keras.callbacks.Callback):
    """
    Implementing the algorithm used to update the training rate in the paper:
        T. Cazenave, \Residual networks for computer go,"
        Available: https://doi.org/10.1109/TCIAIG.2017.2681042

    The reason we are subclassing keras.callbacks.Callback and not keras.callbacks.LearningRateScheduler
    is because we have to update it during epochs and not at the end of each epochs according to the pape.

    The paper defines:
        - A step as 5 000 training examples
        - An epoch as 5 000 000 training examples
        - Update rate of learning rate every 1000 steps

    In our case we have (for N = 10000, batch_size = 128):
        - We consider 1 batch as one training example
        - A step as one batch (we have to stick with the project implementation)
        - An epoch as N / batch_size = 79 steps

    Therefore, by a simple cross multiplication, we have:
        - Update rate of learning rate every  (1000 * batch_size) / 5000  = 26 steps i.e. every 26 batch
    """
    def __init__(self, initial_lr=0.2, batch_size = 128):
        super(ResnetLearningRateScheduler, self).__init__()
        self.initial_lr = initial_lr
        self.batches_per_step = (1000 * batch_size) // 5000
        self.error_history = []
        self.last_lr_update_step = 0
        self.current_step = 0

    def on_train_batch_end(self, batch, logs=None):
        error = logs.get("loss")
        self.error_history.append(error)

        # Increment step counter every "batches_per_step"
        if len(self.error_history) % self.batches_per_step == 0:
            self.current_step += 1

            if self.current_step > 4: # Learning rate does not update at least for 4 step
              # Compute recent and previous errors
              error_recent = sum(self.error_history[-self.batches_per_step:]) / (self.batches_per_step)
              error_previous = sum(self.error_history[-self.batches_per_step*2:-self.batches_per_step]) / (self.batches_per_step)

              # Check if learning rate should be reduced
              if (self.current_step - self.last_lr_update_step) > 3 and (error_recent >= error_previous):
                  new_lr = self.model.optimizer.learning_rate.numpy() / 2
                  if new_lr < 0.01: new_lr = 0.01
                  self.model.optimizer.learning_rate.assign(new_lr)
                  self.last_lr_update_step = self.current_step
                  print(f"\nStep {self.current_step}: Learning rate reduced to {new_lr:.5f}")

    def on_epoch_begin(self, epoch, logs=None):
        # Print the current learning rate at the start of each epoch
        current_lr = self.model.optimizer.learning_rate.numpy()
        print(f"Current learning rate is {current_lr:.5f}")
```

A.2 Tournaments scores

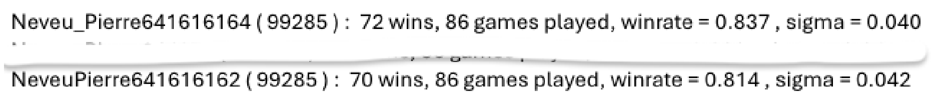

```
Neveu_Pierre641616164 ( 99285 ) :  72 wins, 86 games played, winrate = 0.837 , sigma = 0.040
NeveuPierre641616162 ( 99285 ) :  70 wins, 86 games played, winrate = 0.814 , sigma = 0.042
```

Figure 7: Stats of two models with different value weight loss (last digit): one with 4 and one with 2.

A.3 More Graphics

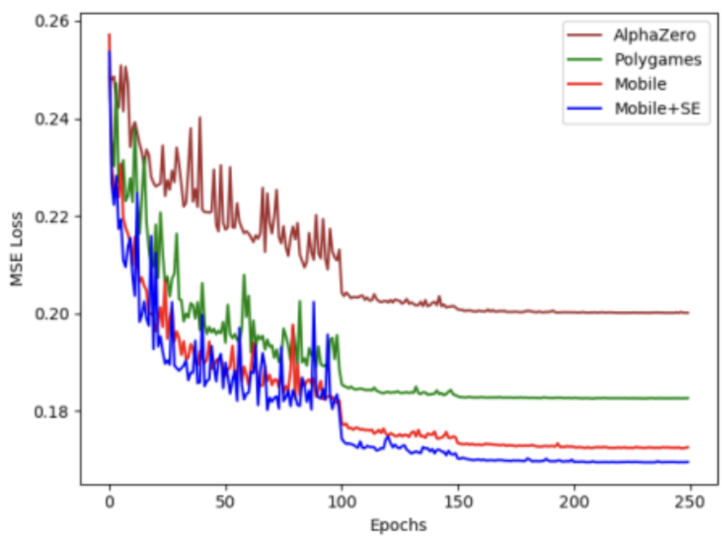


Figure 8: **MSE Loss**: Mobilenet vs Mobilenet with Squeeze excitation blocks vs AlphaZero

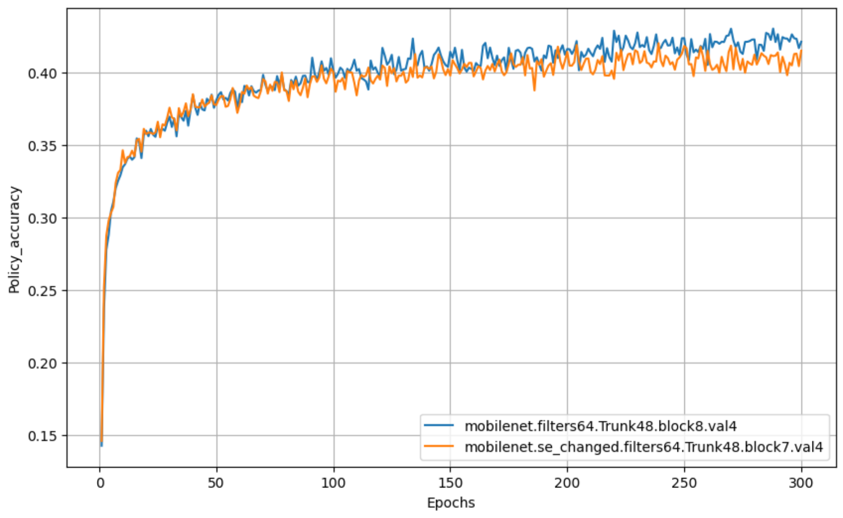


Figure 9: **Policy accuracy**: Mobilenet v2 vs Mobilenet v3 with squeeze excitation block at different positions

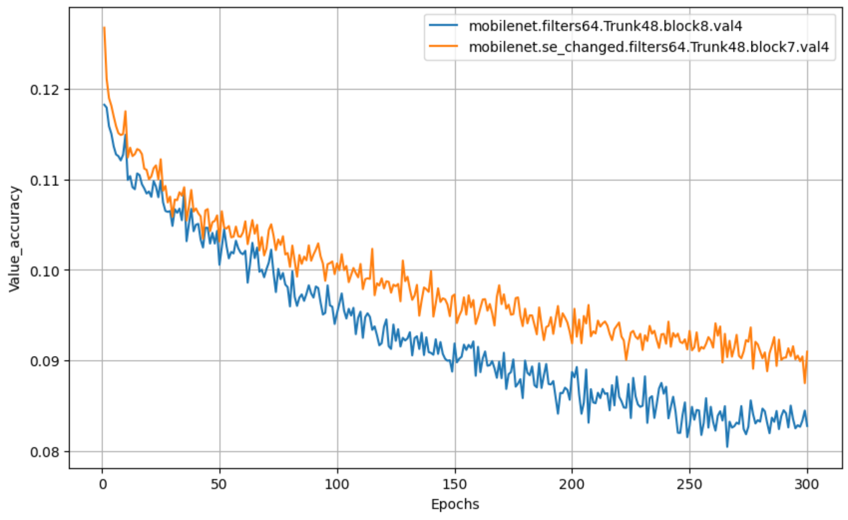


Figure 10: **Value MSE**: Mobilenet v2 vs Mobilenet v3 with squeeze excitation block at different positions

.

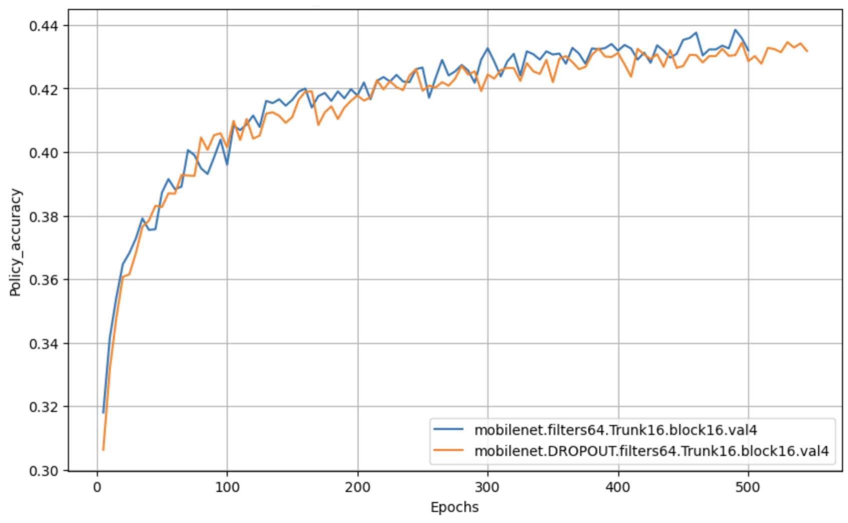


Figure 11: **Policy accuracy**: With or without dropout in the value head and last block

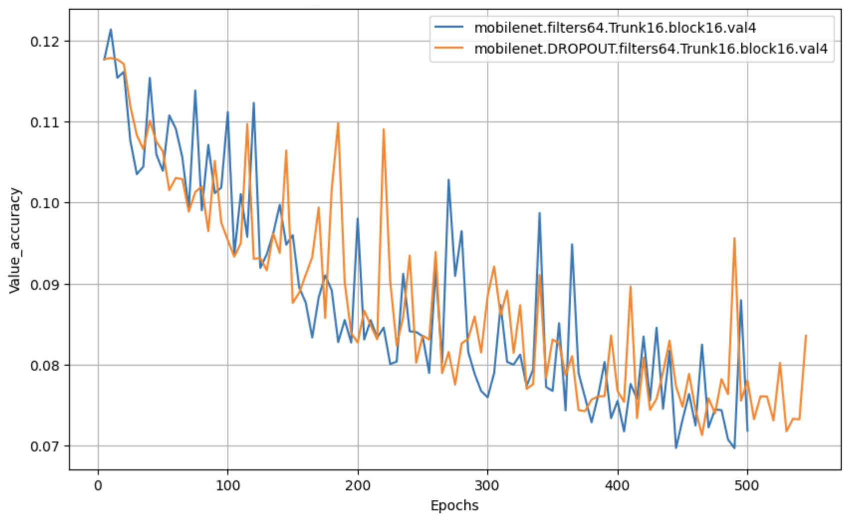


Figure 12: **Value MSE**: With or without dropout in the value head and last block

.

References

[1] T. Cazenave, *Residual Networks for Computer Go*, 2018.

[2] D. Silver, Aja Huang, Chris J. Maddison, *Mastering the game of Go with deep neural networks and tree search*, 2016.

[3] T. Cazenave, *Spatial average pooling for computer go*, 2018.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, *Deep Residual Learning for Image Recognition*, 2016.

[5] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson, *How transferable are features in deep neural networks?* , 2014.

[6] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017.

[7] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, *MobileNetV2: Inverted Residuals and Linear Bottlenecks*, 2018.

[8] Andrew Howard, Mark Sandler, Grace Chu1, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, Hartwig Adam, *Searching for MobileNetV3*, 2019.

[9] T. Cazenave, *Mobile Networks for Computer Go*, 2020.

[10] T. Cazenave, *Improving model and search for computer Go*, 2021.

[11] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu, *Squeeze-and-Excitation Networks*, 2017.

[12] T. Cazenave, *Cosine annealing, mixnet and swish activation for computer Go*, 2021.

[13] Mingxing Tan, Quoc V. Le, *MixConv: Mixed Depthwise Convolutional Kernels*, 2016.

[14] Ilya Loshchilov, Frank Hutter, *SGDR: Stochastic Gradient Descent with Warm Restarts*, 2016.

[15] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson, *How transferable are features in deep neural networks?*, 2016.