Implementing AlphaZero algorithm from scratch

for the ICGA computer olympiad

Student : Enzo Durand

Tutor : Jean-Noël Vittaut

Table of contents

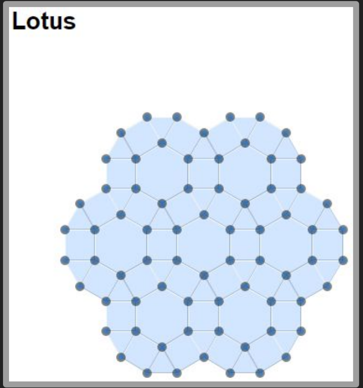
1. Introduction

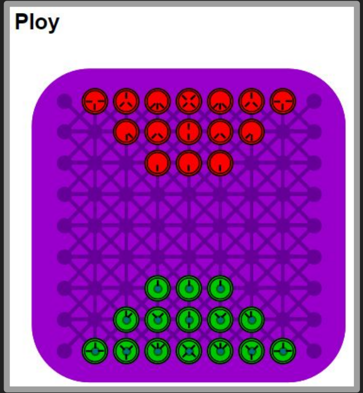
→ 6 games

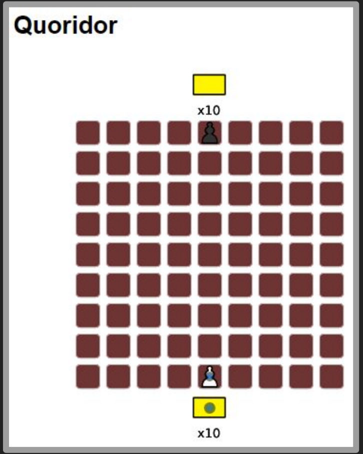
→ 1 algorithm

→ Gameplays are very different

→ Action and observation space are also really different







2. Monte-Carlo tree search

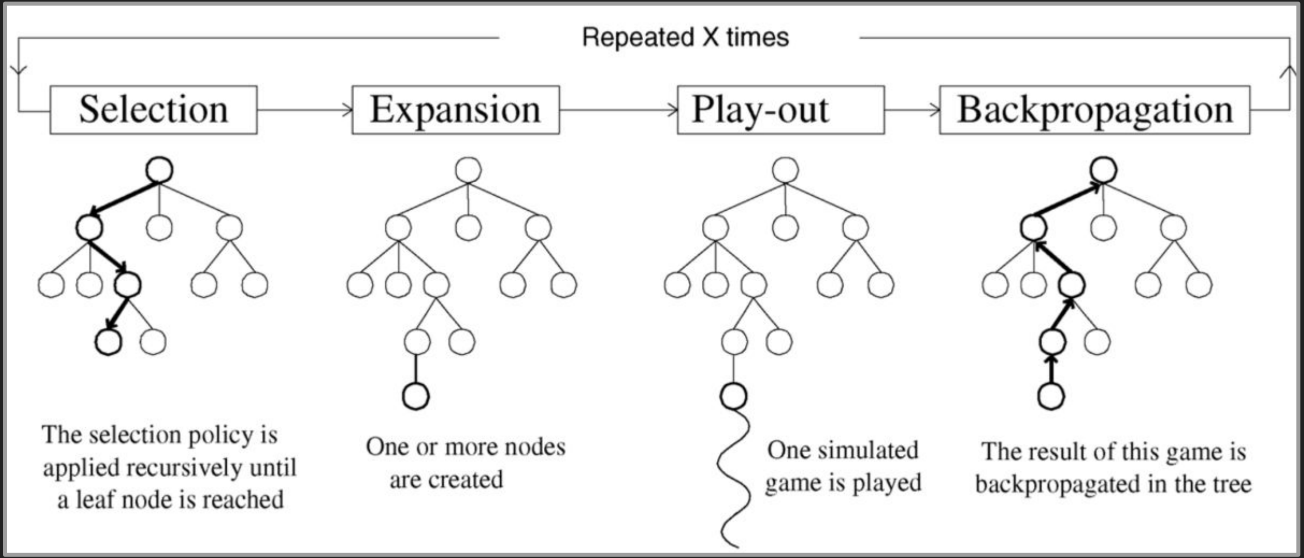
2.1. Algorithm

→ Selection policy can be improved

→ Slow because of the play-out

→ Playing randomly might not be the best way to evaluate a node

→ Let's use deep learning to estimate a policy and a value !



→ Final decision : root action leading to the state with most visits

2. Monte-Carlo tree search

2.2. Selection policy

→ w_i : number of wins

→ n_i : number of simulations

→ N_i : total number of simulations

→ c : exploration hyper-parameter

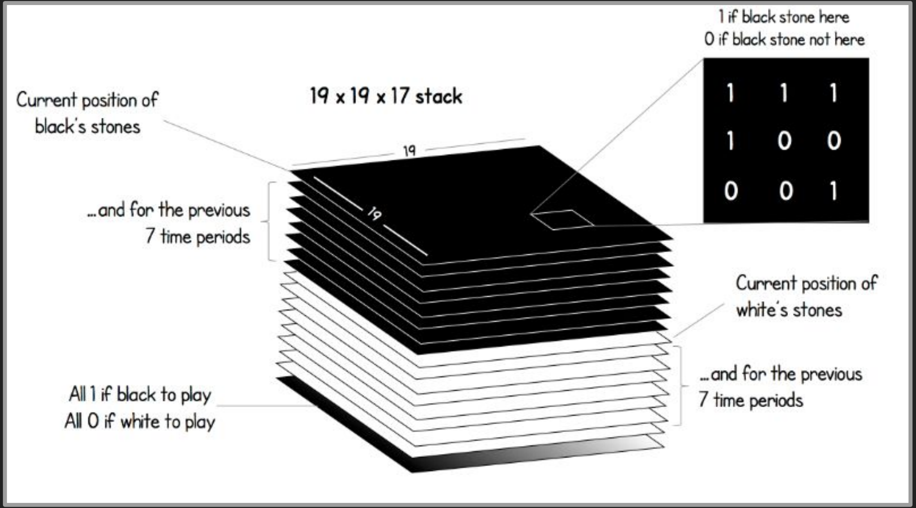
Exploitation **Exploration**

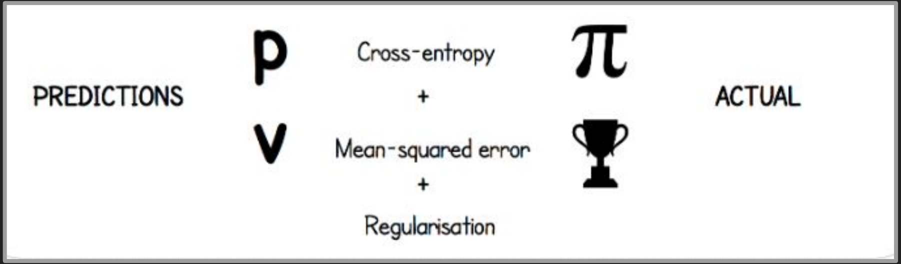
$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

3. Deep learning

3.1. Input & outputs

→ We want to predict a policy and a value for each state during the MCTS iterations





→ p : estimated policy by the model

→ v : estimated value by the model

→ Pi : what was the actual policy coming from the MCTS

→ Cup : which player actually won the current game

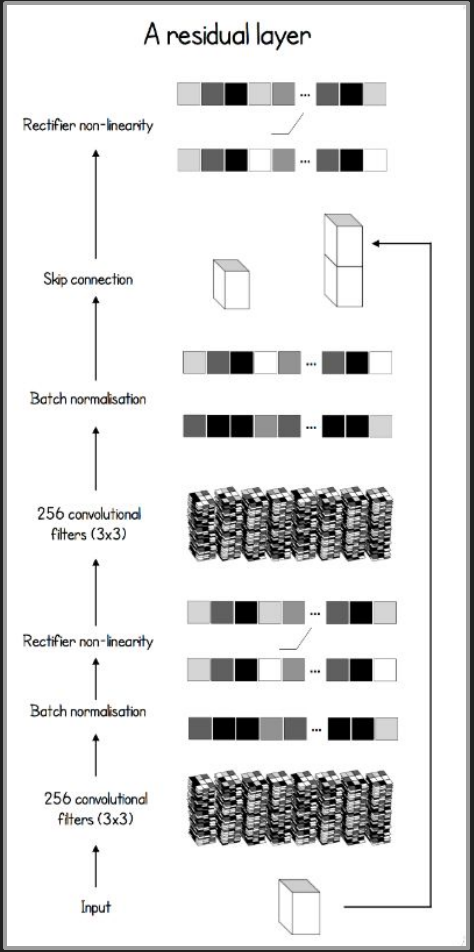
3. Deep learning

3.2. Backbone

→ This 40 residual layers backbone extracts the features from the game state

→ Residual layers are efficient in very deep convolutional neural networks (40 layers in the paper)

→ Skip connection to prevent information loss and vanishing gradient



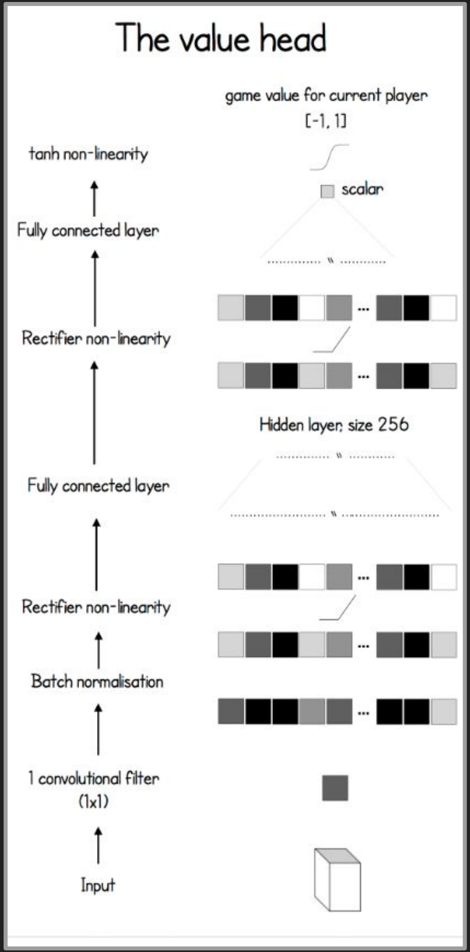
3. Deep learning

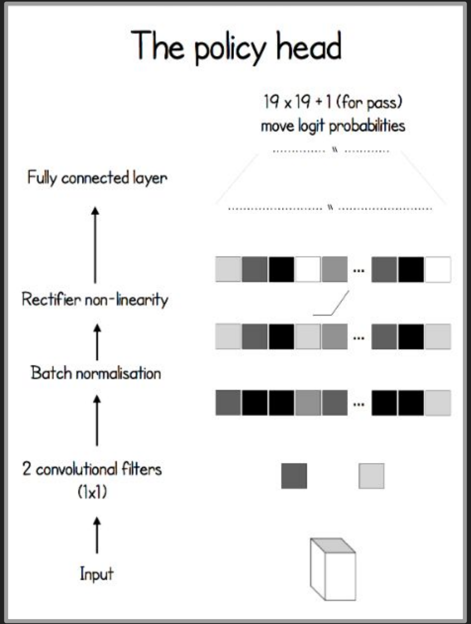
3.3. Value & policy heads

→ Multi-head network

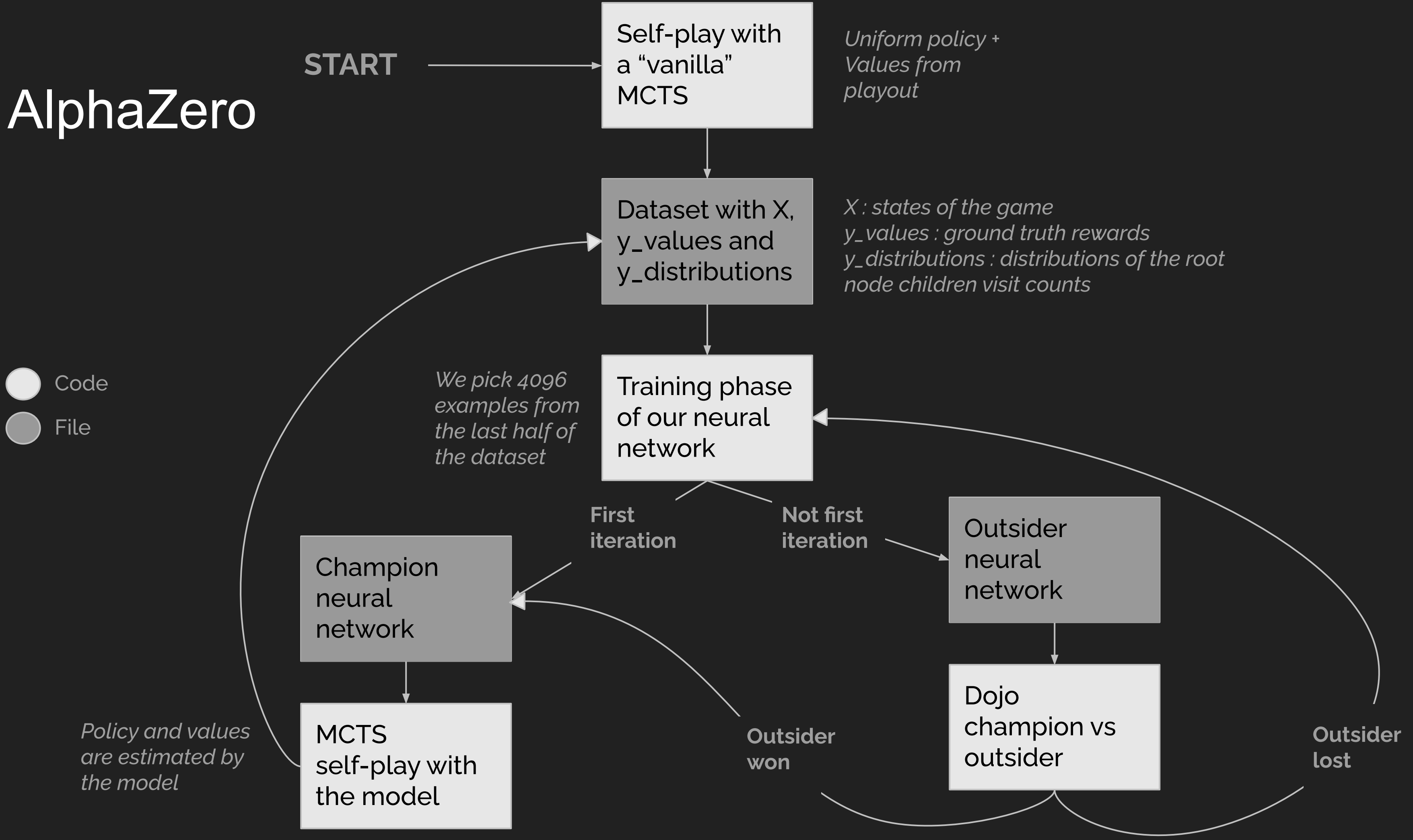
→ Estimates a scalar value between -1 and 1

→ Estimates a policy tensor that can be multidimensional depending on the action space





4. AlphaZero


START
Self-play with a "vanilla" MCTS
Uniform policy + Values from playout
Dataset with X, y_values and y_distributions
X : states of the game
y_values : ground truth rewards
y_distributions : distributions of the root node children visit counts
Training phase of our neural network
We pick 4096 examples from the last half of the dataset
Code
File
First iteration
Not first iteration
Champion neural network
Outsider neural network
MCTS self-play with the model
Policy and values are estimated by the model
Dojo champion vs outsider
Outsider won
Outsider lost

5. Interesting points

5.1 Software engineering

→ Python can be slow

→ Java-Python bridge because the game models are in Java (Ludii library)

→ Multiple files such as .pkl, .h5, .onnx, .sm, .txt etc

→ Lots of bash and python scripts for the clusters and core algorithm

```
AlphaZeroICGA/
    ├── src/
    |       ├── main/
    |       |     ├── agents/            (Contains the jar files of the final agents)
    |       |     ├── bin/               (Contains the binary files compiled from src_java)
    |       |     ├── datasets/          (Contains the (state,distrib,value) datasets)
    |       |     ├── final_model/       (Contains the final weights of the best models)
    |       |     ├── libs/              (Contains the librairies such as JPY/Ludii...)
    |       |     ├── models/            (Contains the current models)
    |       |     ├── src_java/          (Contains all the source code in java)
    |       |     ├── src_python/        (Contains all the source code in python)
    |       |     |     ├── brain/        (Contains the deep learning part)
    |       |     |     ├── mcts/         (Contains the vanilla MCTS and AlphaZero MCTS)
    |       |     |     ├── optimization/ (Contains the optimization part such as precomputations)
    |       |     |     ├── other/        (Contains utility files)
    |       |     |     ├── run/          (Contains files runned by java files such as dojo, trials...)
    |       |     |     ├── scripts/      (Contains all the scripts such as merge_datasets.py)
    |       |     |     ├── settings/     (Contains the hyperparameters and games settings)
    |       |     |     └── utils.py      (File containing the utility functions)
    |       |     ├── cluster_scripts/   (Contains the script to run alphazero on multiple nodes)
    |       |     ├── cluster_logs/      (Contains all the logs output from the cluster scripts)
    |       |     ├── alphazero.py       (Script running the whole AlphaZero algorithm)
    |       |     ├── alphazero_m.py     (Same but only playing model vs model)
    |       |     ├── alphazero_mm.py    (Same but running on multiple nodes if on cluster)
    |       |     ├── build.xml          (Build file helping us run java commands, clean...)
    |       |     └── notes.txt          (Some notes I left while doing that project)
    |       └── test/                    (Some Ludii tutorials and tests)
    ├── alphazero_env.yml                (Conda environment save)
    ├── README.md
    └── LICENSE
```

5. Interesting points

5.2. Multi-processing & Multi-node

→ Not that hard to implement

→ Very hard to make it work in reasonable time

→ Need to parallelize everything we can

→ Multi-threading fails because of the Java-Python bridge

→ Multi-processing is a solution

→ Multi-node on the CPU/GPU clusters aswell

```
def run_trials(n_workers, n_nodes):
        print("*******************************************************************************************")
        print("**************************************** RUNNING TRIALS ****************************************")
        print("*******************************************************************************************")
        Popen("sbatch cluster_scripts/run_trials.sh "+ str(n_nodes) + " " + str(n_workers), shell=True).wait()

        while True:
                n_files = len([f for f in listdir(DATASET_PATH) \
                                                if isfile(join(DATASET_PATH, f)) \
                                                and any(char.isdigit() for char in join(DATASET_PATH, f))])
                if n_files >= n_nodes * n_workers:
                        print("*******************************************************************************************")
                        print("*************************************** MERGING DATASETS ***************************************")
                        print("*******************************************************************************************")
                        Popen("python3 src_python/scripts/merge_datasets.py", shell=True).wait()
                        break
```

5. Interesting points

5.3. Code optimization

→ Huge number of python function calls in the self-play part

→ Avoid using loops and try to optimize with numpy

→ Pre-compute every possible functions at the start

```python
def precompute_get_coord():
    n_returns = 4
    pre_coords = np.zeros((N_ROW*N_COL, N_ROW*N_COL, n_returns), dtype=int)
    for from_ in range(N_ROW*N_COL):
        for to_ in range(N_ROW*N_COL):
            for index_return in range(n_returns):
                pre_coords[from_][to_][index_return] = get_coord(from_, to_)[index_return]
    return pre_coords
```

```python
def precompute_get_3D_coord():
    n_returns = 3
    pre_3D_coords = np.zeros((N_ROW*N_COL*N_ACTION_STACK, n_returns), dtype=int)
    for value in range(N_ROW*N_COL*N_ACTION_STACK):
        for index_return in range(n_returns):
            pre_3D_coords[value][index_return] = get_3D_coord(value)[index_return]
    return pre_3D_coords
```

5. Interesting points

5.4. Hyper-parameters

→ Huge number of hyper-parameters

→ Need to tune those hyper-parameters to get nice performances

→ Trade-off between performance and training time

```
######### TIME CONSUMING VARIABLES #########

ONNX_INFERENCE = True # ONNX inference should be False if using GPU

N_EPOCHS = 100
EARLY_STOPPING_PATIENCE = 10

NUM_EPISODE = 10 # Number of self play games by worker
VANILLA_EPISODE_MULTIPLIER = 5 # Factor by which we multiply the number o
MAX_ITERATION_AGENT = 100 # Max number of nodes discovered by the MCTS
THINKING_TIME_AGENT = -1 # Max number of seconds for the MCTS to run

NUM_DOJO = 4
MAX_ITERATION_AGENTS_DOJO = 100
THINKING_TIME_AGENTS_DOJO = -1

N_BATCH_PREDICTION = 5 # Number of batch per MCTS simulation
MINIMUM_QUEUE_PREDICTION = MAX_ITERATION_AGENT//N_BATCH_PREDICTION + 1 #
```

```
######### MCTS PARAMETERS #########

CSTE_PUCT = 2 # Exploration constant
MAX_SAMPLE = 10_000 # Can decide the max size of the datas
WEIGHTED_SUM_DIR = 0.75 # this value comes from the paper
DIRICHLET_ALPHA = 10/N_MOVES_TYPICAL_POSITION_CONNECTFOUR
TEMPERATURE = 1 # 1 -> no change, 0 -> argmax
```

```
######### NN parameters #########

TRAIN_SAMPLE_SIZE = 4096
RANDOM_SEED = 42
BATCH_SIZE = 512
VERBOSE = 1
VALIDATION_SPLIT = 0.25

MAIN_ACTIVATION = "relu"
FILTERS = 64
KERNEL_SIZE = (3,3)
FIRST_KERNEL_SIZE = (3,3)
USE_BIAS = True
N_RES_LAYER = 5
NEURONS_VALUE_HEAD = 128 # Number of neurons in last dense layer

OPTIMIZER = "sgd"
LEARNING_RATE_DECAY_IT = 5 # LR decay every 5 alphazero iteration
LEARNING_RATE_DECAY_FACTOR = 2 # Divided by 2 each time
BASE_LEARNING_RATE = 0.1
MOMENTUM = 0.9
REG_CONST = 1e-5 # L2 reg

LOSS_WEIGHTS = [0.5, 0.5]
```

5. Interesting points

5.5. Modifying the algorithm

→ Learning is too slow

→ But working on TicTacToe !

→ We are not DeepMind (super-computers + research engineers)

→ Need to modify the algorithm

→ Find the bottlenecks

→ Find a solution

→ Divided training time by a huge factor

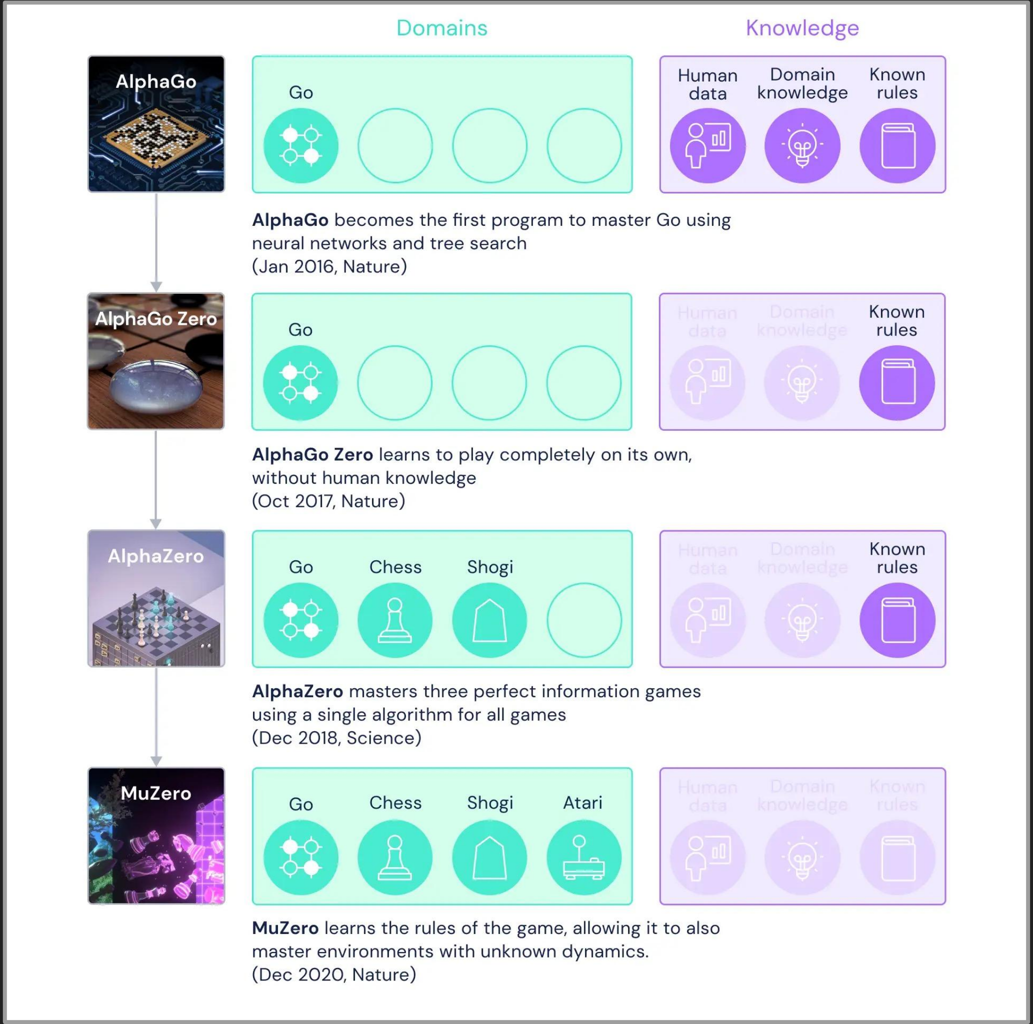
```python
# Adding the current node to the predict queue list in order to estimate the values later
predict_queue.append(current)
# Here we predict values if the queue length is higher than a minimum value or if it's the
# last iteration in order to avoid missing values before the final decision
if len(predict_queue) >= MINIMUM_QUEUE_PREDICTION or num_iterations == max_its - 1:
    # Predict the values of the whole queue
    utils, policy_preds = self.predict_values(predict_queue)

    # Check if we can compute some ground truth utils
    utils = self.check_ground_truth(predict_queue, utils)

    # Backpropagated the utility scores
    self.backpropagate_predicted_values(predict_queue, utils, policy_preds)

    # Empty the predict queue
    predict_queue = []
# If it's not time to estimate the values then we put all the values to 0, we don't need to
# backpropagate the values. This makes us save time and the MCTS becomes pessimistic
else:
    current.value_pred = 0
    current.value_opp_pred = 0

# Here for each node we backpropagate the visit counts
self.backpropagate_visit_counts(current)
```

6. Can we do better ?

→ Optimize the inference time with GPU parallelization approach

→ Transfer learning between games to avoid starting learning from scratch

→ Curriculum learning



Thanks for listening !

Any questions ?

Code available @ github.com/hanzopgp/AlphaZeroICGA