HerculeX

AlphaZero for Hex, but cheap and not working

Iancu Onescu*, Mihail Feraru†,
*value head, †policy head

I. Introduction

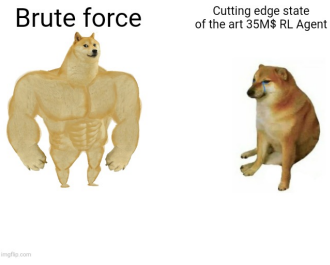
Hex is a two player abstract strategy board game in which players attempt to connect opposite sides of a hexagonal board by placing stones on empty cells. The most common board size in competitions and causal play is 11x11 but 13x13 or 19x19 sizes are also popular. Despite the simple rules, hex has a higher branching factor than most popular board games such as Go or Chess and comparable or even more elaborate tactics. Due to its interesting mathematical properties it is an active research subject in both computer science and mathematics.

The game has a small amount of rules which are also very simple:

1) Each player has a colour and owns two opposing sides of the board. (usually Red and Blue, and Red owns top and bottom)
2) Each turn, a stone is placed in an empty hexagon by the current player.
3) If a player connects their owned sides with a chain of stones, they win.
4) On their first turn, the second player can choose to swap the placed piece with their own. (the Swap Rule)

The Swap Rule is used to counteract the strategic advantage of the player having the first turn. In our game implementation we ignored the Swap Rule for convenience.

We implemented a proof-of-concept bot powered by Deep Reinforcement Learning and inspired by Google's AlphaZero bot, which we trained (on a PC cheaper than 35MM$) using self-play and benchmarked against a random player on a board so small that it was already solved by brute-force. In the following paragraphs we will describe the architecture, technical implementation and results of our project.

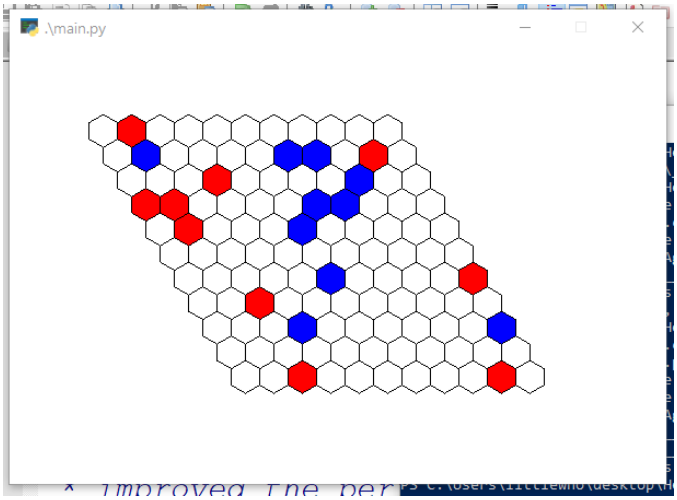


II. Method

As a guarantee for reusability, we developed our bot on top of a OpenAI Gym environment. The main libraries used for computations and machine learning components were Numpy and Tensorflow. For parallelizing different stages of work we used Python's multiprocessing module.

A. Gym environment

The environment allows playing the game in CvC mode, two agents interacting each one being unaware of the other one. We support RGB display and a debugging mode for displaying information about the internal state of the environment.



An agent interacting with environment expects to see observations from a space, and produce actions, which we will describe below. Given the set of possible states of a cell $C = \{RED,\ BLUE,\ EMPTY\}$, and a board size of n:

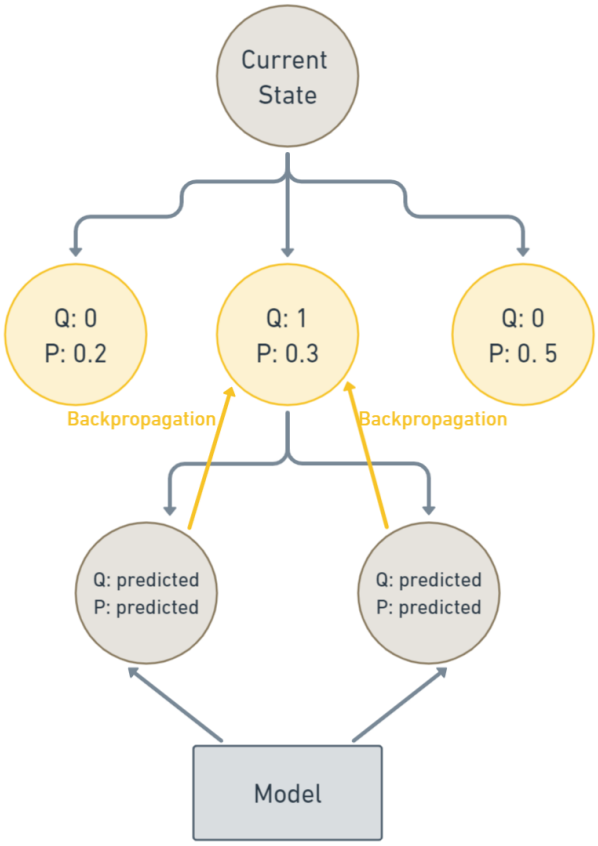
$$S = C^{n \times n} \quad A = \{(x, y) \mid S_{ij} = EMPTY,\ i, j \in \overline{0, n-1}\}$$

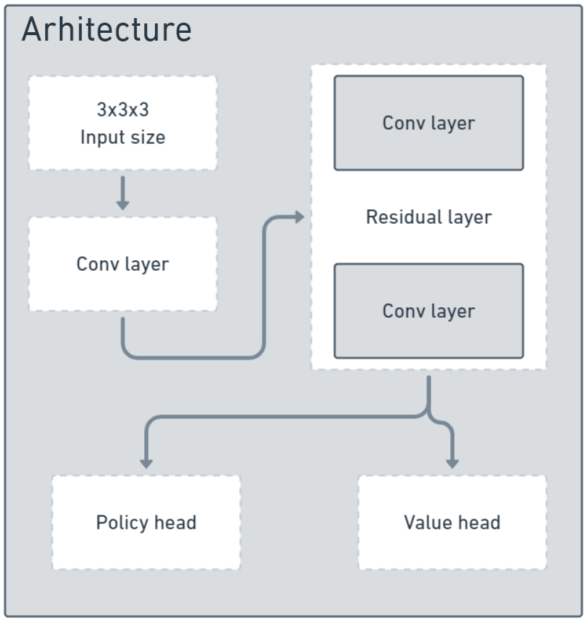
B. Alpha-Zero architecture and implementation

Our implementation follows the true AlphaZero closely.We use a deep network consisting of convolutional and residual layers with two value heads in order to predict a correct policy and value for a specified position on the board. The input shape is 3x3x3, because we break the board into two 2D-vectors, consisting in the moves of red and blue respectively followed by a layer filled with the value of the player that needs to make a move.

The output is used inside a Monte-Carlo Search Tree in order to determine the best move starting from a current position. The following steps are applied: Selection, that expands the tree by finding the best leaf, Expansion, that creates another layer of leaves and Backpropagation that perpetuates the changes in the tree. The method is slightly altered from the classical MCTS because instead of playing out the game from a chosen leaf we use the network to predict what we think will be the outcome. The best leaf is chosen using the following formula:

And finally, in order to chose the best move we take the path that has been traveled the most (meaning that it has the most potential). The training of the model improves prediction making our agent almost unbeatable.

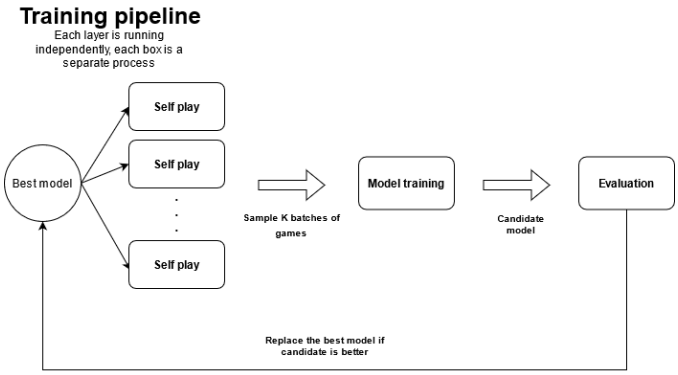




C. Parallel training pipeline

To speed up the training process, we split the independent components into separate processes, asynchronously consuming the products of each other. The layers of our pipeline are:

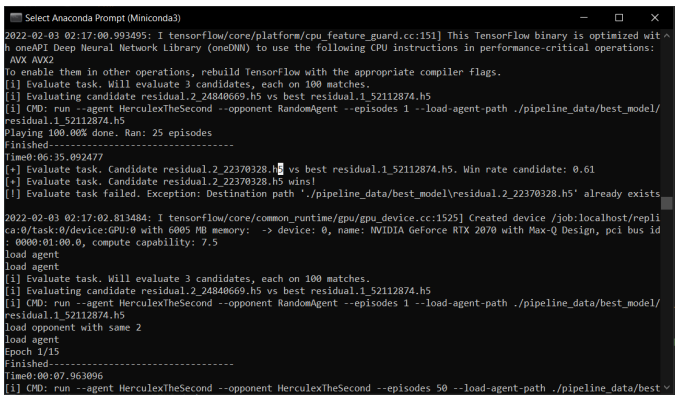
1) The self-play layer - N processes playing matches (episodes) using the current best model against itself and saving the acquired experience
2) Training layer - samples K batches of episodes and retrains the best model using them, the output is a candidate model
3) Evaluation layer - takes a candidate model and plays a series of matches against the current best model, if the candidate model wins more than 50% of matches, it will become the best model



III. Results

Continuous training has shown interesting results. The model has a big jump in the beginning where it manages a whooping 68% winrate over the random model, and then proceeds to slowly improve each generation up until when it his a plateau. Because of the size of the board a decent model playing with the red hexes (the player that goes first) has a significant advantage and is more prone to win. So in the end it all comes down to how many episodes we benchmark on in order to get a better figure.

Here is an example of a model that manages to outclass his predecessor.



This is definitely proof of concept for the fact that with enough training and a solid pipeline the model can learn to overcome any situation presented before it.

IV. Challenges

During the development of our project, we faced a series of challenges, part of which we solved, others remaining unaddressed. We will talk about the ones we could not solve as a reference for further improvements.

One of the main concerns when it comes to training AI systems are performance and efficient use of computational resources. Besides using more powerful and specialized hardware, such as using GPUs or TPUs instead of CPUs, there are some bottlenecks present in the design and implementation which we will discuss below:

1) Tensorflow v2 performance drop - there are active discusses about cases where Tensorflow v2 performs worse than v1 and PyTorch being a faster alternative
2) Tensorflow v2 bugs in predict - model predictions in for loops tend to be slow and require a lot of low level optimizations
3) Mixing tensors and arrays decreases performance - mixing Tensorflow with numpy has to be done carefully and with clear boundaries between the two, otherwise context switching highly impacts performance
4) Python can't be any faster - implementing the MCTS or the Hex game simulator in C++ would drastically speed up computations
5) Hex game win condition - checking if a board is in a winning state requires using a data structure which supports Unite and Find operations for sets which is computationally expensive
6) Unit testing is the greatest invention of all time, use it extensively

V. Conclusions

If somebody reads a paper on an ultra high tech, state of the art model architecture and a goofy way some people used to put it into practice maybe they should respect their work and keep training linear models. :)