Deep Q Learning with 2048

Bill Qian and Rilyn Han
Professor: James Glenn
CPSC 474 - Computational Intelligence for Games
2048 is a classic game where given a set of nxm squares, and move up down left and right, to try to combine squares and achieve the highest score possible. The game is infinitely flexible, and changes to the rules can alter the best strategy, but the fundamental gameplay will stay the same.

We are inspired by AlphaZero, which is developed by Google to handle turn-based games with perfect information. The algorithm has been used extensively to dominate in chess (Leela Chess Zero), Go (AlphaGo), and we thought we might try to incorporate into a simpler game like 2048.

Note that the game 2048 is a one player game with a stochastic element, which differs from the 2-player strategy games that AlphaZero is designed for. Our idea here is to treat the stochastic process (eg. square generation) sort of like it's own "player" of sorts. The stochastic player (player B) will have a probability distribution to generate the moves such that result in the worst possible position for the player moving the tiles (player A), and player A will play moves such that player B cannot, even in the worst possible scenarios, end the game with a sequence of unlucky (from player A's perspective) moves.

The reason we expect this to work is because we are assuming that 2048 is a game that stays winning until the player starts losing. What we mean by this is that as long as the strategy is avoiding the worst case, then it will likely continue playing for as long as possible. The strategy doesn't necessarily need to be efficient, but it must refrain from losing for as long as possible. If you think of it another way, since the sum of the squares on the board is equal to the sum of all generated tiles, there is no "fast" way to get to 2048 (or higher), but there is a way to lose all progress and start over again.

Let's first define what we are using for this algorithm. First, we use MCTS to search for new positions in the game. MCTS will start from the root node, and continue to expand the game tree by visiting every combination of moves and observing all the actions. Note, however, that there is an element of randomness in the form of generating moves, which is what the MCTS is good at tackling.

We define a simple MCTS algorithm as follows:

```
While we have more time
    node = root node
    while node is not a terminal node:
        node = choose_node(node)

    if node is terminal:
        reward = some value for win/loss
    else:
        reward = (play to the end/result of game)

    update all nodes in the path to the root with
        node.visit +=1
        node.reward +=1

When time is up:
    Return choose_node(root)
```

choose_node() can be defined using the UCT algorithm, which is defined as

$$\text{val} = Q(s,a) + c \cdot \sqrt{\frac{\log N(s))}{N(s,a)}}$$

(Note that $N(s)$ means visits to node s, while $N(s,a)$ is the number of times action a has been taken from s (aka the child of s))
Now note that the state space could be huge; if we played to the end of the game every time, and achieved 2048, then that would be *at least* 2048/4=512 iterations. For every node! That seems like a huge waste of computing power, so what if we could approximate the reward via some function instead?

Suppose we had a function $P(s,a)$ that would give us some values for the 4 possible actions that I might take at position a. Instead of purely relying on a logarithm of the number of visits ($\log N(s)$), which is a very stale function and hence might not accurately reflect the best move, we add a learnable weight onto it instead. Therefore, this might look like the following:

$$\text{val} = Q(s,a) + c \cdot P(s,a) \cdot \sqrt{\frac{\log N(s))}{N(s,a)}}$$

In this case we would no longer need the square root, so we could expand to something like

$$\text{val} = Q(s,a) + c \cdot P(s,a) \cdot \sqrt{\frac{N(s)}{N(s,a)}}$$

Empirically, it seems that in the Google AlphaZero paper, they found it better to use the following (removing the square root from the denominator), as it encourages more exploration,

decays nicely as action is explored, and keeps balance stable across different games/domains. We now pick the node with the highest value of:

$$\text{val} = Q(s,a) + c \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{N(s,a)}$$

This means that our MCTS has essentially been reduced to

```
While we have more time
    node = root node
    while node is not a terminal node:
        node = choose_node(node)

    if node is terminal:
        reward = some value for win/loss
    else:
        **reward = V(node) <- NEW**

    update all nodes in the path to the root with
        node.visit +=1
        node.reward +=1

When time is up:
    Return choose_node(root)
```

Now, it's just time to figure out how to get the functions $P(s,a)$ and $V(s)$. In this project we proposed a neural network to handle this. Let the game state be G, with $G_{i,j}$ denoting the value of the square at row i, column j, where 0 represents that the cell is currently empty.

Note that the output will be 5 items: the first 4 part of the probability distribution for the 4 possible moves given the state s, and the fifth one being an estimate of the value of the current state s. In a more visual form,

$$[p_1, p_2, p_3, p_4, v_1]$$

We have the following options to design the architecture:

1. Classic CNN

In a $n \times m$ grid, let each $k_{i,j} = \log x_{i,j}$ be the characteristic value of each cell. Then, encode the matrix with $X_{i,j} = k_{i,j}$ and feed into the neural network. The remaining layers would just be:

Input $n \times m$
-> CNN with 2x2 stride -> $((n-1) \times (m-1))$

-> Flatten -> 9x1
-> Linear -> 10x1
-> Linear -> 4x1 + 1x1

2. One-hot encoding

In an $n \times m$ grid, let each $k_{i,j}$ represent the one-hot-encoding state of each cell. Then, let the maximum representable number for now be $t = 20$. Then, we create a $n \times m \times t$ tensor with one-hot encoding with the t dimension. The remaining layers might be

Self-Play

To train the neural network, we utilize the self-play framework, which acts roughly as follows:

```
w = initialize weights
for _ in num_epochs:
    # basically start an empty game and play it to finish
    state = empty game
    data = []
    while state is not terminal:
        run mcts on state with old policy
            (here, also pick a move generator, either adverserial or random)
            the current state is the root. but basically the rest is the
same
            return the policy (calculated via the number of visits of the 4
children of the root).
            also return the expected value of the root (value of root/num
visits).
        policy, z = [p1, p2, p3, p4], [z1]
        data.append(state, policy, z) # basically a slightly better version
of what should have happened
        state = generate_move(state, ...)

    with these state, (policy, z) expected pairs, we can train the NN with
supervised learning. Use cross entropy for policy, mse for z (can you do
this?)
```

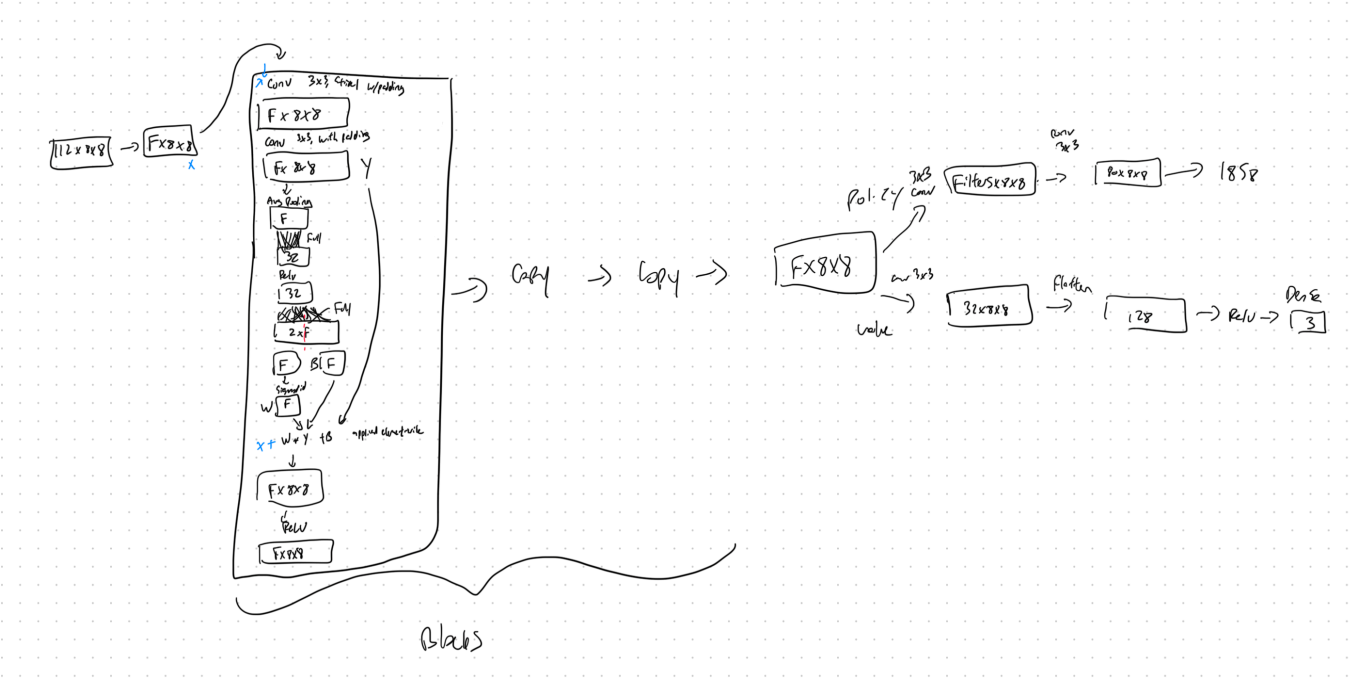
The main idea of self-play is that the current policy fully plays out a few games, however on each move, a MCTS search with the current policy is added. Theoretically, by using MCTS with the existing policy instead of just the policy itself (eg. selecting the best move out of the 4-move probability distribution, every move the policy "thinks" for itself using its current knowledge, and

with some added help it makes a more polished decision. Basically it's like compressing the knowledge of thousands of "if this then that's" into a single score function, and using it there.)

Therefore, we train the self-play NN, where the training processes uses the exact MCTS function that is the actual final finished product.

To further motivate our search, we explore the architecture powering Leela Chess Zero here .

Here is a very, very rough sketch of the architecture. It takes in 112 8x8 planes, each of which is the representation of a piece, the history, castling rights, etc.



We can kind of replicate this to 2048. Consider the following transformation:

- Plane 0 is $n \times m$, $a_{ij} = 1$ if it is empty
- Plane 1 is $n \times m$, $a_{ij} = 1$ if value is 2
- Plane 2 is $n \times m$, $a_{ij} = 1$ if value is 4
- Plane 3 is $n \times m$, $a_{ij} = 1$ if value is 8
- ...
- Plane 20 is $n \times m$, $a_{ij} = 1$ if value is $2^{20} = 1048576$
- Plane k (configurable parameter) is $n \times m$, $a_{ij} = 1$ if value is 2^k

(Parameters FILTERS, BLOCKS)
Then, we copy the LC0 network format:

- Input $k \times n \times m$
- Conv2D with 3x3 size, stride 1, padding=1 -> $F \times n \times m$
- Conv2D with 3x3 size, stride 1, padding=1 -> $F \times n \times m$ (call this **x**)

- BEGIN RESIDUAL BLOCK WITH INPUT x
  - Conv2D with 3x3 size, stride 1, padding 1 -> $F \times n \times m$
  - Conv2D with 3x3 size, stride 1, padding 1 -> $F \times n \times m$
  - Global Average pooling $n \times m$ into 1: -> $F \times 1$
  - Fully connected: 32×1
  - Relu -> 32×1
  - Fully connected: $2F \times 1$
  - Logical split into W and B (no operation): $\dim W = F \times 1$, $\dim B = F \times 1$
  - Sigmoid to W, nothing to B: $\dim W = F \times 1$, $\dim B = F \times 1$
  - Combined = $x + (W \times y) + B$ (broadcast, apply W elementwise) -> $F \times n \times m$
  - Relu -> $F \times n \times m$
  - Output is of dimension $F \times n \times m$
- After residual blocks with output $F \times n \times m$,
  - For policy head:
    - Conv2D with 3x3 size, stride 1, padding=1 -> $F \times n \times m$
    - Conv2D with 3x3 size, stride 1, padding=1 -> $40 \times n \times m$
    - Flatten -> $Fnm \times 1$
    - Linear + Softmax-> 4×1
  - For value head:
    - Conv2D with 3x3 size, stride 1, padding=1 -> $16 \times n \times m$ (reduced from LC0)
    - Conv2D with 3x3 size, stride 1, padding=1 -> $8 \times n \times m$
    - Flatten: $8nm \times 1$
    - ReLU
    - Fully Connected -> score (float value) 1×1

Now, let's implement all of this!

The simplified2048 class has the following methods:

- Simplified2048(height, width, spawn_rates, num_spawn_tiles_per_move, num_initial_tiles)
- get_board() -> 2D Numpy array
- get_score() -> current score
- is_game_over() -> returns true if there are no moves left
- move(direction: [Simplified2048.UP/DOWN/LEFT/RIGHT]) -> returns score gain, total score
- clone() -> returns a new copy
- get_valid_moves() -> List
- generate_tiles()-> None

Note that the state Q(s,a) should indicate the state after the move is played, but before the squares have been generated.

Some notes we made along the way:

- We optimized the 2048 into a Bitboard to achieve maximum performance. We could have switched to C++, however that might have resulted in us having difficulty compiling against the libtorch libraries.

Distributed Training

Note that as cited in LC0, AlphaZero, and many other papers using similar implementations of this algorithm, this algorithm requires tens of thousands of games to be played before any meaningful performance improvement will be shown. To combat this, we designed a distributed training architecture to accelerate the self-play framework, which is the main bottleneck compared to the training process.

We had first tried a parallel, synchronous approach, but this proved too complex as we wanted to implement all of the technologies from scratch (synchronous distributed computing is very tough, requiring distributed locking, distributed replicated state machines, etc), so we eventually settled on a parameter-server-like approach. It works as follows:

A main server revolves around the following few states:

```
While True:
        send weights out to clients by placing it in R2
        wait for enough games to be generated by clients and received from
server
        tell clients to wait while the model gets trained
        propagate the model back out to the clients
        repeat
```

The client works as follows

```
While True:
        Wait for model and task sent from server
        If there are weights and argumetns
```

With this architecture, we were about to scale the training to about 70 GPUs across a variety of platforms that all point to the parameter server. We estimate (based on system load) that we could connect thousands of GPUs to the main server and not have it crash.

Note that this is functionally identical to training synchronously, as each of the MCTS/2048 games are independent of each other. Therefore, we are able to do this without any performance loss.

Our training hyperparameters were as follows:

- Self play
  - Simulations per move: 800
  - Minimum number of self-play games: 2048 → 384 (human-changed at revision 6)
  - Dirichlet Alpha: 0.3
  - Dirichlet Epsilon: 0.25
  - Maximum moves: 1000
  - Temperature: 1.0
- Training
  - Epochs: 10
  - Batch Size: 64
  - Weight decay: 1e-4
  - Momentum: 0.9
  - Scheduler step size: 20
  - Scheduler gamma: 0.5
- Dataset
  - Num workers: 4
  - Validation split: 0.1
- Network
  - Filters: 128
  - Block: 10
  - k: 16

Due to time constraints we actually used a minimized network called ZeroNetworkMini. This is a 4.1 million parameter network with the following architecture:

```
class ZeroNetworkMini(ZeroNetworkMain):
    def __init__(self, n: int, m: int, k: int, filters=128, dense_out=64,
blocks = None, infer_device='cpu'):
        super(ZeroNetworkMini, self).__init__(n, m, k, filters=filters,
infer_device=infer_device)
        self.n = n
        self.m = m
        self.k = k
        self.filters = filters
```

```
        self.dense_out = dense_out

        self.conva1 = nn.Conv2d(in_channels=k, out_channels=filters,
kernel_size=(3, 3), padding='same')
        self.conva2 = nn.Conv2d(in_channels=filters, out_channels=filters,
kernel_size=(3, 3), padding='same')

        self.convb1 = nn.Conv2d(in_channels=k, out_channels=filters,
kernel_size=(3, 3), padding='same')
        self.convb2 = nn.Conv2d(in_channels=filters, out_channels=filters,
kernel_size=(3, 3), padding='same')

        self.flat = nn.Flatten()

        flattened_size = filters * n * m

        self.densea = nn.Linear(in_features=flattened_size,
out_features=dense_out)
        self.denseb = nn.Linear(in_features=flattened_size,
out_features=dense_out)

        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.softmax = nn.Softmax(dim=1)
        self.policy_linear = nn.Linear(in_features=dense_out,
out_features=4)

        self.value_linear = nn.Linear(in_features=dense_out, out_features=1)

        self.infer_device = infer_device
        self.to(infer_device)
    def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        a = self.relu(self.conva1(x))
        a = self.relu(self.conva2(a))
        a_flat = self.flat(a)
        a_dense = self.relu(self.densea(a_flat))

        b = self.relu(self.convb1(x))
        b = self.relu(self.convb2(b))
        b_flat = self.flat(b)
        b_dense = self.relu(self.denseb(b_flat))

        merged = a_dense * b_dense

        policy_logits = self.policy_linear(merged)
        policy_out = self.softmax(policy_logits)
```

```
        value_out = self.value_linear(merged)
        # value_out = self.tanh(value_out)

        return policy_out, value_out
```

This network is still inspired by the AlphaZero paper, however we left out parts of the network that we believe would only minimally contribute to position evaluation.

Minimax

We use expectimax, which is minimax with chance nodes to look several piles ahead and choose the move with the highest expected score.

At every turn, we generate the four successor boards obtained by moving up/right/down/left. For each legal successor board, we run an expectimax search to a fixed depth d. Here, max node is denoted as the alternative player, who choose the move that maximizes, using our customized heuristic function. chance nodes are the natural nodes; they branch over all empty squares and spawn a 2 (90%) or a 4 (10%) and weight the child values by those probabilities.

We then pick the direction whose expectimax value is highest.

Our Heuristic at depth-0 leaves is defined as:

`H(b) = −0.1 ∗ Smoothness + 1.0 ∗ MonotonicityPenalty + 2.7 ∗ EmptyTiles + 3.5 ∗ MergePotential`

`H(b) = α Smoothness + β MonotonicityPenalty + γ EmptyTiles + δ MergePotential`

Baseline Algorithms

We create 2 baseline algorithms to compare against: The first one is (Algorithm A):

```
For every step, pick a completely random move
```

The second one is (Algorithm B):

```
For every step
    For each of the 4 directions to move
        find the number of merges that this move would make
        to handle tiebreaks, first prioritize the score, then prioritize the
first move chosen
    pick the move with the most number of merges
```

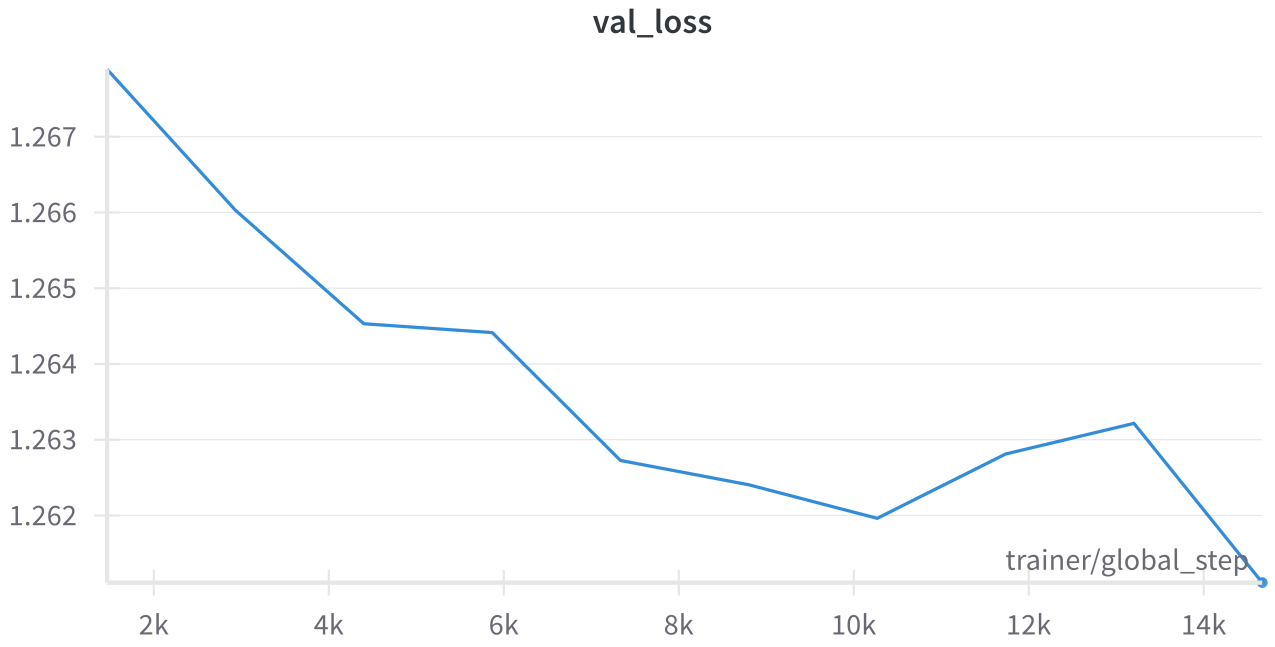
Results

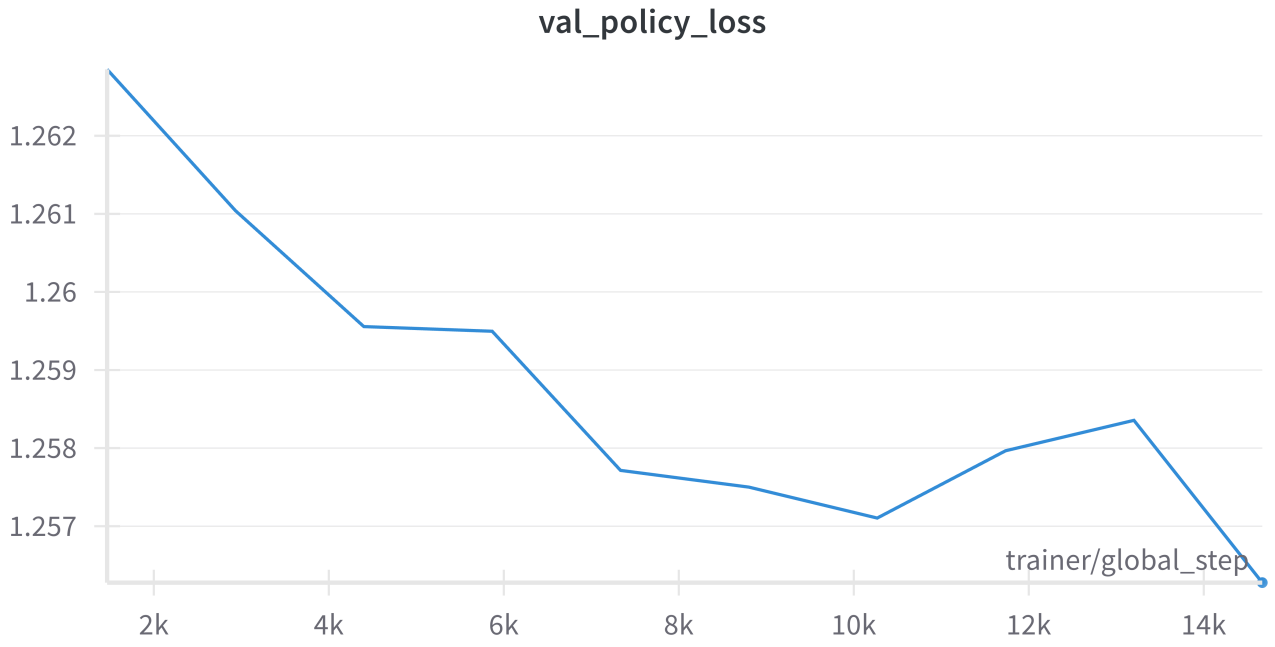
To test the performance, we run each simulation 100 times, capturing the percentage of games with the highest tile being 1024, 2048, 4096, etc. Note that the hyperparameters from the distributed training are kept for the evaluation, and the results are from the 7th revision of this algorithm. If curious, the full training run can

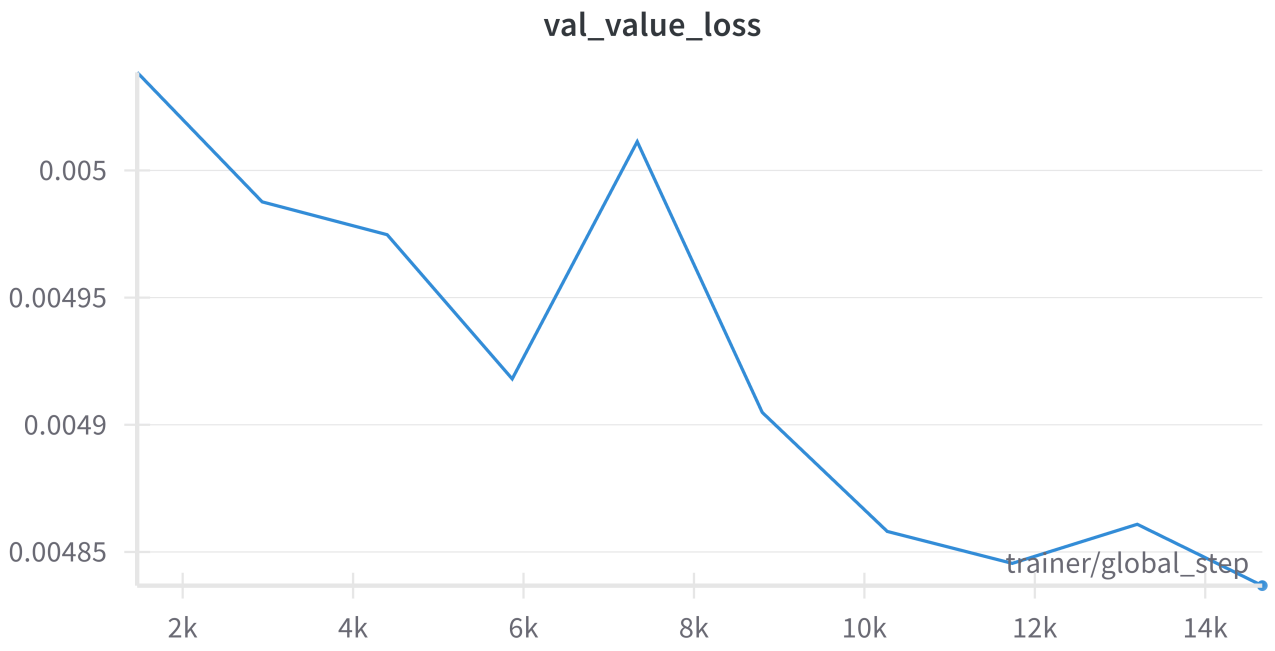
Max Tile	Algorithm A	Algorithm B	Minimax	AlphaZero-like
512				
1024				
2048				
4096				
8192				

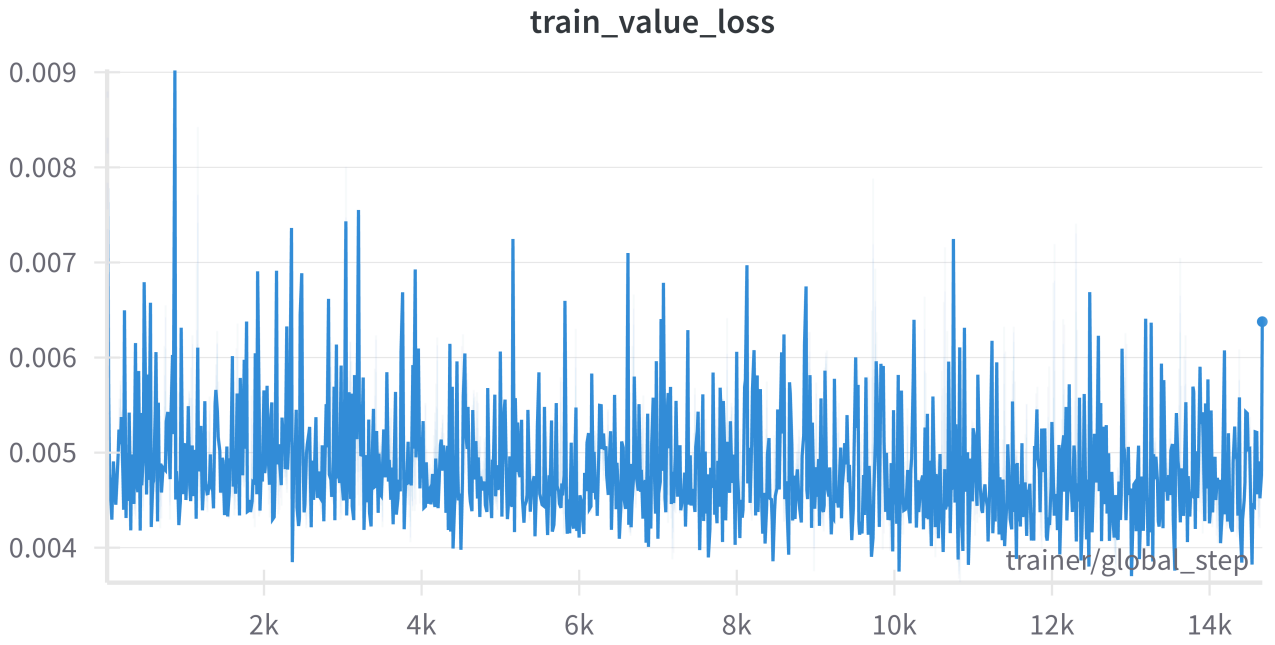
While our results for the AlphaZero-like architecture were a little disappointing, we are confident that a large part of this can be attributed to our compute limitations. We are seeing policy loss and value loss both decreasing during the training runs, and are confident that the trend will continue. While we would like to utilize architectural ideas from the original AlphaZero and Leela Chess Zero implementations, the reality is that those setups either have thousand-GPU datacenter farms, or have an army of community contributors that far outclass what we could supply. Even though we gathered around a 70 GPU cluster for our training, and we used a significantly reduced version of our main net, we would still need weeks to achieve good performance with our setup. Therefore, we massively underestimate the compute time that this would take. However, based on the current trajectory we do believe that eventually our algorithm and systems will achieve very solid performance once it converges, and we do not

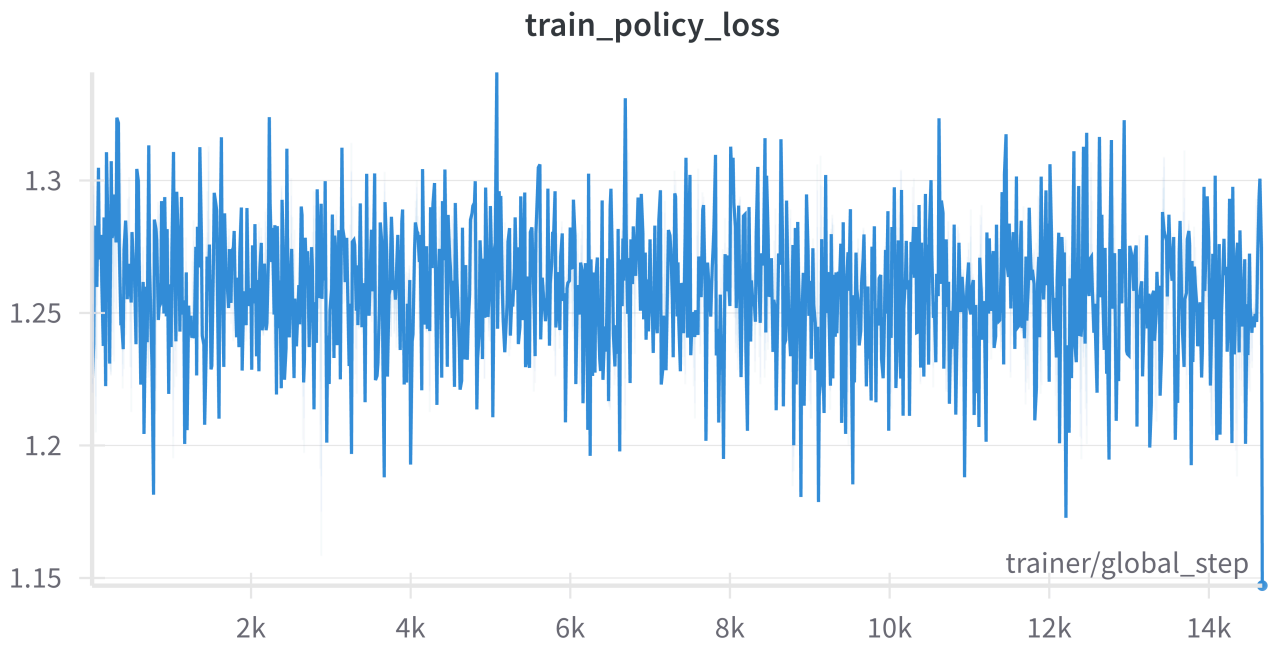
anticipate any other major changes to the code to achieve this, just more compute time.

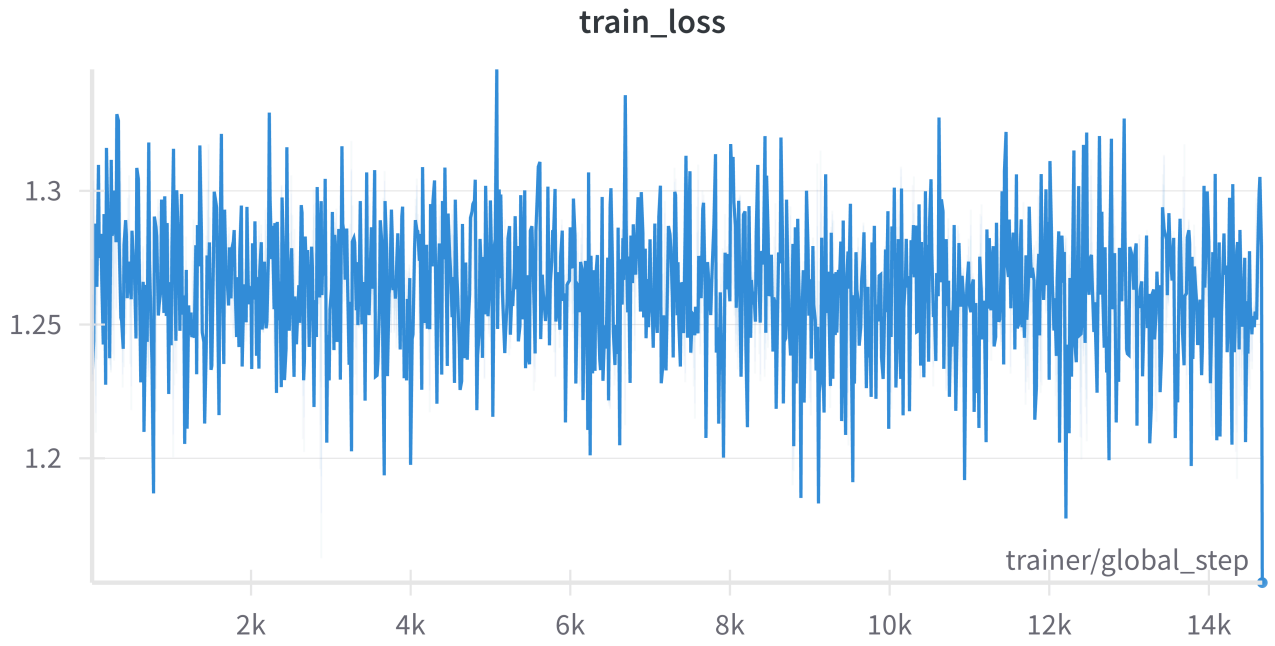





val_value_loss
0.005
0.00495
0.0049
0.00485
trainer/global_step
2k
4k
6k
8k
10k
12k
14k


train_value_loss
0.009
0.008
0.007
0.006
0.005
0.004
trainer/global_step
2k
4k
6k
8k
10k
12k
14k





In the coming few days, out of our own curiosity, we will continue this training process to see how much performance we can get, as the loss is still converging.