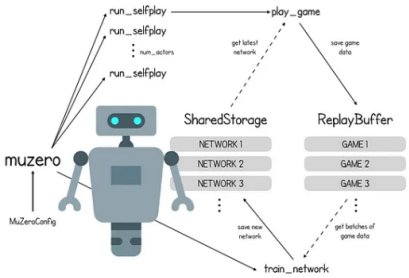
Published in Applied Data Science

David Foster

Dec 2, 2019 · 7 min read · Listen



MuZero: The Walkthrough (Part 2/3)

Teaching a machine to play games using self-play and deep learning... without telling it the rules ☢️

> *If you want to learn how one of the most sophisticated AI systems ever built works, you've come to the right place.*

This is the second part in the series that walks through the pseudocode released by DeepMind for their groundbreaking reinforcement learning model, MuZero.

👉 Part 1

👉 Part 3

. . .

Last time, we introduced MuZero and saw how it is different from its older brother, AlphaZero.

In the absence of the actual rules of chess, MuZero creates a new game inside its mind that it can control and uses this to plan into the future. The three networks (**prediction**, **dynamics** and **representation**) are optimised together so that strategies that perform well inside the imagined environment, also perform well in the real environment.

In this post, we'll walk through the `play_game` function and see how MuZero makes a decision about the next best move at each turn.

Playing a game with MuZero (play_game)

We're now going to step through the `play_game` function:

```
# Each game is produced by starting at the initial board position, then
# repeatedly executing a Monte Carlo Tree Search to generate moves until the end
# of the game is reached.
def play_game(config: MuZeroConfig, network: Network) -> Game:
  game = config.new_game()

  while not game.terminal() and len(game.history) < config.max_moves:
    # At the root of the search tree we use the representation function to
    # obtain a hidden state given the current observation.
    root = Node(0)
    current_observation = game.make_image(-1)
    expand_node(root, game.to_play(), game.legal_actions(),
                network.initial_inference(current_observation))
    add_exploration_noise(config, root)

    # We then run a Monte Carlo Tree Search using only action sequences and the
    # model learned by the network.
    run_mcts(config, root, game.action_history(), network)
    action = select_action(config, len(game.history), root, network)
    game.apply(action)
    game.store_search_statistics(root)
  return game
```

pseudocode.py hosted with ❤ by GitHub view raw

First, a new `Game` object is created and the main game loop is started. The game ends when a terminal condition has been met or if the number of moves is longer that the maximum allowed.

We start the MCTS tree with the root node.

```
root = Node(0)
```

Each Node stores key statistics relating to the number of times it has been visited `visit_count`, whose turn it is `to_play`, the predicted prior probability of choosing the action that leads to this node `prior`, the backfilled value sum of the node `node_sum`, its child nodes `children`, the hidden state it corresponds to `hidden_state` and the predicted reward received by moving to this node `reward`.

```
11
12    def expanded(self) -> bool:
13      return len(self.children) > 0
14
15    def value(self) -> float:
16      if self.visit_count == 0:
17        return 0
18      return self.value_sum / self.visit_count
```

pseudocode.py hosted with ♥ by GitHub view raw

Next we ask the game to return the current observation (corresponding to `o` in the diagram above)...

```
current_observation = game.make_image(-1)
```

...and expand the root node using the known legal actions provided by the game and the inference about the current observation provided by the `initial_inference` function.

```
expand_node(root, game.to_play(), game.legal_actions(),
network.initial_inference(current_observation))
```

```
1   # We expand a node using the value, reward and policy prediction obtained from
2   # the neural network.
3   def expand_node(node: Node, to_play: Player, actions: List[Action],
4                   network_output: NetworkOutput):
5     node.to_play = to_play
6     node.hidden_state = network_output.hidden_state
7     node.reward = network_output.reward
8     policy = {a: math.exp(network_output.policy_logits[a]) for a in actions}
9     policy_sum = sum(policy.values())
10    for action, p in policy.items():
11      node.children[action] = Node(p / policy_sum)
```

pseudocode.py hosted with ♥ by GitHub view raw

We also need to add exploration noise to the root node actions — this is important to ensure that the MCTS explores a range of possible actions rather than only exploring the action which it currently believes to be optimal. For chess, `root_dirichlet_alpha` = 0.3.

```
add_exploration_noise(config, root)
```

```
6     frac = config.root_exploration_fraction
7     for a, n in zip(actions, noise):
8       node.children[a].prior = node.children[a].prior * (1 - frac) + n * frac
```

pseudocode.py hosted with ♥ by GitHub view raw

We now hit the main MCTS process, which we will cover in the next section.

```
run_mcts(config, root, game.action_history(), network)
```

The Monte Carlo Search Tree in MuZero (run_mcts)

```
    history = action_history.clone()
    node = root
    search_path = [node]

    while node.expanded():
      action, node = select_child(config, node, min_max_stats)
      history.add_action(action)
      search_path.append(node)

    # Inside the search tree we use the dynamics function to obtain the next
    # hidden state given an action and the previous hidden state.
    parent = search_path[-2]
    network_output = network.recurrent_inference(parent.hidden_state,
                                                 history.last_action())
    expand_node(node, history.to_play(), history.action_space(), network_output)

    backpropagate(search_path, network_output.value, history.to_play(),
                  config.discount, min_max_stats)
```

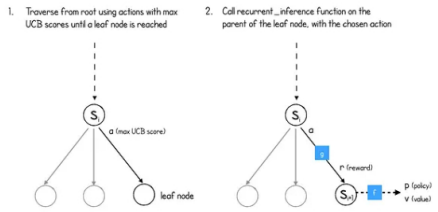
pseudocode.py hosted with ❤ by GitHub view raw

As MuZero has no knowledge of the environment rules, it also has no knowledge of the bounds on the rewards that it may receive throughout the learning process. The `MinMaxStats` object is created to store information on the current minimum and maximum rewards encountered so that MuZero can normalise its value output accordingly. Alternatively, this can also be initialised with known bounds for a game such as chess (-1, 1).

The main MCTS loop iterates over `num_simulations`, where one simulation is a pass through the MCTS tree until a leaf node (i.e. unexplored node) is reached and subsequent backpropagation. Let's walk through one simulation now.

First, the `history` is initialised with the of list of actions taken so far from the start of the game. The current `node` is the `root` node and the `search_path` contains only the current node.

The simulation then proceeds as shown in the diagrams below:



MuZero first traverses down the MCTS tree, always selecting the action with the highest UCB (Upper Confidence Bound) score:

```
# Select the child with the highest UCB score.
def select_child(config: MuZeroConfig, node: Node,
                 min_max_stats: MinMaxStats):
  _, action, child = max(
      (ucb_score(config, node, child, min_max_stats), action,
       child) for action, child in node.children.items())
  return action, child
```

pseudocode.py hosted with ❤ by GitHub view raw

The UCB score is a measure that balances the estimated value of the action `Q(s,a)` with a exploration bonus based on the prior probability of selecting the action `P(s,a)` and the number of times the action has already been selected `N(s,a)`.

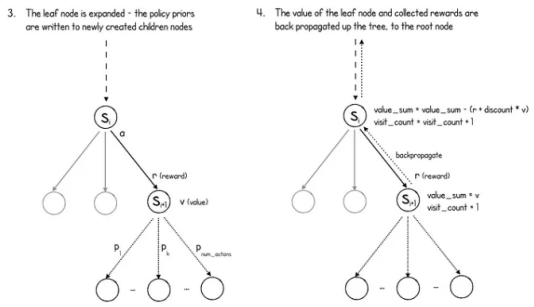
$$a^k = \arg\max_a \left[Q(s,a) + P(s,a) \cdot \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \left(c_1 + \log\left(\frac{\sum_b N(s,b) + c_2 + 1}{c_2} \right) \right) \right]$$

The action with the highest UCB score is chosen at each node of the MCTS tree.

Early on in the simulation, the exploration bonus dominates, but as the total number of simulations increases, the value term becomes more important.

Eventually, the process will reach a leaf node (a node that has not yet been expanded, so has no children).

At this point, the `recurrent_inference` function is called on the parent of the leaf node, in order to obtain the predicted reward and new hidden state (from the **dynamics** network) and policy and value of the new hidden state (from the **prediction** network).



The MCTS process (leaf expansion and backpropagation)

As shown in the diagrams above, the leaf node is now expanded by creating new children nodes (one for every possible action in the game) and assigning each node with its respective policy prior. Note that MuZero doesn't check which of these actions are legal or if the action results in the end of the game (it can't), so creates a node for every action whether it is legal or not.

Lastly, the value predicted by the network is back-propagated up the tree, along the search path.

Remember, since these are *predicted* rewards, rather than actual rewards from the environment, the collection of rewards is relevant even for a game like chess, where true rewards are only awarded at the end of the game. MuZero is playing its own imagined game, which may include interim rewards, even if the game it is modelled on, doesn't.

This completes one simulation of the MCTS process.

After `num_simulations` passes through the tree, the process stops and an action is chosen based on the number of times each child node of the root has been visited.

```python
def select_action(config: MuZeroConfig, num_moves: int, node: Node,
                  network: Network):
  visit_counts = [
      (child.visit_count, action) for action, child in node.children.items()
  ]
  t = config.visit_softmax_temperature_fn(
      num_moves=num_moves, training_steps=network.training_steps())
  _, action = softmax_sample(visit_counts, t)
  return action

def visit_softmax_temperature(num_moves, training_steps):
  if num_moves < 30:
    return 1.0
  else:
    return 0.0  # Play according to the max.
```

pseudocode.py hosted with ♥ by GitHub view raw

For the first 30 moves, the temperate of the softmax is set to 1, meaning that the probability of selection for each action is proportional to the number of times it has been visited. From the 30th move onwards, the action with the maximum number of visits is selected.

$$p_\alpha = \frac{N(\alpha)^{1/T}}{\sum_b N(b)^{1/T}}$$

softmax_sample: The probability of selecting action 'alpha' from the root node (N is the number of visits)

Though the number of visits may feel a strange metric on which to select the final action, it isn't really, as the UCB selection criteria within the MCTS process is designed to eventually spend more time exploring actions that it feels are truly high value opportunities, once it has sufficiently explored the alternatives early on in the process.

The chosen action is then applied to the true environment and relevant values are appended to the following lists in the `game` object.

- `game.rewards` — a list of true rewards received at each turn of the game
- `game.history` — a list of actions taken at each turn of the game
- `game.child_visits` — a list of action probability distributions from the root node at each turn of the game
- `game.root_values` — a list of values of the root node at each turn of the game

These lists are important as they will eventually be used to build the training data for the neural networks!

This process continues, creating a new MCTS tree from scratch each turn and using it to choose an action, until the end of the game.

All of the game data (`rewards`, `history`, `child_visits`, `root_values`) is saved to the replay buffer and the actor is then free to start a new game.

Phew.

. . .

This is the end of Part 2 , a complete tour of how MuZero plays games against itself and saves the game data to a buffer.

In Part 3, we'll look at how MuZero trains itself to improve, based on saved data from previous games. I'll also wrap up with a summary of why I think MuZero is a major advancement for AI and the implications for the future of the field.

Please clap if you've enjoyed this post and I'll see you in Part 3!

. . .

This is the blog of Applied Data Science Partners, a consultancy that develops innovative data science solutions for businesses. To learn more, feel free to get in touch through our website.

Artificial Intelligence | Machine Learning | Deep Learning | Data Science | Programming

670 3

Deep Learning 10 min read

Share your ideas with millions of readers. Write on Medium

Daniel Sharp · Jan 17, 2022

How To Build You Own Slack Bot

Full Stack Data Scientist: Part 7— How to set up a Slack Bot for instant, automatic notifications — Dashboards are the most popular way of presenting data in businesses — but they're not the only way!...

Data Science 5 min read

David Foster · Jan 26, 2018

How to build your own AlphaZero AI using Python and Keras

Teach a machine to learn Connect4 strategy through self-play and deep learning — Update! (2nd December 2019) I've just released a series on...

Artificial Intelligence 11 min read

David Foster · Sep 28, 2017

NEW R package that makes XGBoost interpretable

xgboostExplainer makes your XGBoost model as transparent and 'white-box' as a single decision tree — In this post I'm going to do three things: Show you how a single decision tree is not great at predicting,...

Machine Learning 9 min read

David Foster · Oct 29, 2017

AlphaGo Zero Explained In One Diagram

Download the AlphaGo Zero cheat sheet — Get the full cheat sheet here Update! (2nd December 2019) I've just released a series on MuZero — AlphaZero's younger and cooler brother. Check it out How to Build...

Machine Learning 2 min read

Read more from Applied Data Science