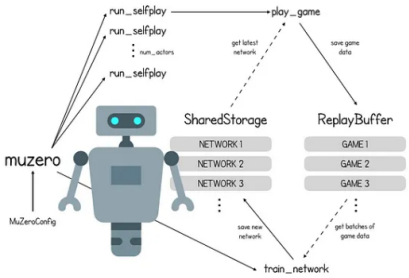
Published in Applied Data Science

David Foster
Dec 2, 2019 · 9 min read · Listen



MuZero: The Walkthrough (Part 3/3)

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 third and final part in a series that walks through the pseudocode released by DeepMind for their groundbreaking reinforcement learning model, MuZero.

👉 Part 1

👉 Part 2

. . .

Last time, we walked through the `play_game` function and saw how MuZero makes a decision about the next best move at each turn. We also explored the MCTS process in more detail.

In this post, we'll take a look at the training process for MuZero and understand the loss function that it is trying to minimise.

I'll 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.

Training (train_network)

The final line of the original entrypoint function (remember that from Part 1?) launches the `train_network` process that continually trains the neural networks using data from the replay buffer.

instances of MuZero's three neural networks) and sets the learning rate to decay based on the number of training steps that have been completed. We also create the gradient descent optimiser that will calculate the magnitude and direction of the weight updates at each training step.

The last part of this function simply loops over `training_steps` (=1,000,000 in the paper, for chess). At each step, it samples a batch of positions from the replay buffer and uses them to update the networks, which is saved to storage every `checkpoint_interval` batches (=1000).

There are therefore two finals parts we need to cover — how MuZero creates a batch of training data and how it uses this to update the weights of the three neural networks.

Creating a training batch (replay_buffer.sample_batch)

The ReplayBuffer class contains a `sample_batch` method to sample a batch of observations from the buffer:

```python
class ReplayBuffer(object):
  def __init__(self, config: MuZeroConfig):
    self.window_size = config.window_size
    self.batch_size = config.batch_size
    self.buffer = []

  def sample_batch(self, num_unroll_steps: int, td_steps: int):
    games = [self.sample_game() for _ in range(self.batch_size)]
    game_pos = [(g, self.sample_position(g)) for g in games]
    return [(g.make_image(i), g.history[i:i + num_unroll_steps],
             g.make_target(i, num_unroll_steps, td_steps, g.to_play()))
            for (g, i) in game_pos]

  ...
```

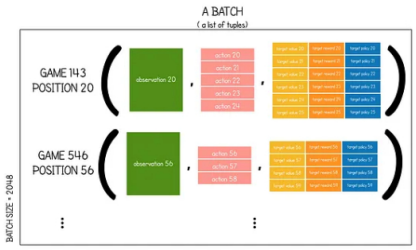
pseudocode.py hosted with ❤ by GitHub view raw

The default `batch_size` of MuZero for chess is 2048. This number of games are selected from the buffer and one position is chosen from each.

A single `batch` is a list of tuples, where each tuple consists of three elements:

- `g.make_image(i)` — the observation at the chosen position
- `g.history[i:i + num_unroll_steps]` — a list of the next `num_unroll_steps` actions taken after the chosen position (if they exist)
- `g.make_target(i, num_unroll_steps, td_steps, g.to_play()` — a list of the targets that will be used to train the neural networks. Specifically, this is a list of tuples: `target_value`, `target_reward` and `target_policy`.

A diagram of a example batch is shown below, where `num_unroll_steps` = 5 (the default value used by MuZero):



An example batch

You may be wondering why multiple future actions are required for each observation. The reason is that we need to train our dynamic network and the only way of doing this is to train on small streams of sequential data.

For each observation in the batch, we will be 'unrolling' the position `num_unroll_steps` into the future using the actions provided. For the initial position, we will use the `initial_inference` function to predict the value, reward and policy and compare these to the target value, target reward and target policy. For subsequent actions, we will use the `recurrent_inference` function to predict the value, reward and policy and compare to the target value, target reward and target policy. This way, all three networks are utilised in the predictive process and therefore the weights in all three networks will be updated.

Let's now look in more detail at how the targets are calculated.

```python
class Game(object):
  """A single episode of interaction with the environment."""

  def __init__(self, action_space_size: int, discount: float):
    self.environment = Environment()  # Game specific environment.
    self.history = []
    self.rewards = []
    self.child_visits = []
    self.root_values = []
    self.action_space_size = action_space_size
    self.discount = discount

  def make_target(self, state_index: int, num_unroll_steps: int, td_steps: int,
                  to_play: Player):
```

```
    # The value target is the discounted root value of the search tree N steps
    # into the future, plus the discounted sum of all rewards until then.
    targets = []
    for current_index in range(state_index, state_index + num_unroll_steps + 1):
      bootstrap_index = current_index + td_steps
      if bootstrap_index < len(self.root_values):
        value = self.root_values[bootstrap_index] * self.discount**td_steps
      else:
        value = 0

      for i, reward in enumerate(self.rewards[current_index:bootstrap_index]):
        value += reward * self.discount**i  # pytype: disable=unsupported-operands

      if current_index < len(self.root_values):
        targets.append((value, self.rewards[current_index],
                        self.child_visits[current_index]))
      else:
        # States past the end of games are treated as absorbing states.
        targets.append((0, 0, []))
    return targets

  ...
```

pseudocode.py hosted with ♥ by GitHub view raw

The `make_target` function uses ideas from TD-learning to calculate the target value of each state in positions from `state_index` to `state_index + num_unroll_steps`. The variable `current_index`.

TD-learning is a common technique used in reinforcement learning — the idea is that we can update the value of a state using the estimated discounted value of a position `td_steps` into the near future plus the discounted rewards up until that point, rather than just using the total discounted rewards accrued by the end of the episode.

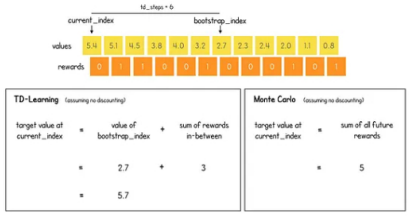
When we are updating an estimate based on an estimate, we say we are **bootstrapping**. The `bootstrap_index` is the index of the position `td_steps` into the future that we will be using to estimate the true future rewards.

The function first checks if the `bootstrap_index` is after the end of the episode. If so, `value` is set to 0, otherwise `value` is set to the discounted predicted value of the position at the `bootstrap_index`.

Then, the discounted rewards accrued between the `current_index` and `bootstrap_index` are added onto the `value`.

Lastly, there is a check to make sure that the `current_index` is not after the end of the episode. If so, empty target values are appended. Otherwise, the calculated TD target value, true reward and policy from the MCTS are appended to the target list.

For chess, `td_steps` is actually set to `max_moves` so that `bootstrap_index` always falls after the end of the episode. In this scenario, we are actually using Monte Carlo estimation of the target value (i.e. the discounted sum of all future rewards to the end of the episode). This is because the reward for chess is only awarded at the end of the episode. The difference between TD-Learning and Monte Carlo estimation is shown in the diagram below:



The difference between TD-Learning method and Monte Carlo method for target value setting

Now that we have seen how the targets are built, we can see how they fit in to the MuZero loss function and finally, see how they are used in the `update_weights` function to train the networks.

The MuZero loss function

The loss function for Muzero is as follows:

$$l_t(\theta) = \sum_{k=0}^{K} l^r(u_{t+k}, r_t^k) + l^v(z_{t+k}, v_t^k) + l^p(\pi_{t+k}, \mathbf{p}_t^k) + c||\theta||^2$$

Here K is the [illegible] variable. In other words, there are three

Here, `k` is the `num_unroll_steps` variable. In other words, there are three losses we are trying to minimise:

1. The difference between the predicted **reward** `k` steps ahead of turn `t` `(r)` and the actual reward `(u)`
2. The difference between the predicted **value** `k` steps ahead of turn `t` `(v)` and the TD target value `(z)`
3. The difference between the predicted **policy** `k` steps ahead of turn `t` `(p)` and the MCTS policy `(pi)`

These losses are summed over the rollout to generate the loss for a given position in the batch. There is also a regularisation term to penalise large weights in the network.

Updating the three MuZero networks (`update_weights`)

```
        image)
    predictions = [(1.0, value, reward, policy_logits)]

    # Recurrent steps, from action and previous hidden state.
    for action in actions:
      value, reward, policy_logits, hidden_state = network.recurrent_inference(
          hidden_state, action)
      predictions.append((1.0 / len(actions), value, reward, policy_logits))

      hidden_state = tf.scale_gradient(hidden_state, 0.5)

    for prediction, target in zip(predictions, targets):
      gradient_scale, value, reward, policy_logits = prediction
      target_value, target_reward, target_policy = target

      l = (
          scalar_loss(value, target_value) +
          scalar_loss(reward, target_reward) +
          tf.nn.softmax_cross_entropy_with_logits(
              logits=policy_logits, labels=target_policy))

      loss += tf.scale_gradient(l, gradient_scale)

  for weights in network.get_weights():
    loss += weight_decay * tf.nn.l2_loss(weights)

  optimizer.minimize(loss)
```

pseudocode.py hosted with ❤ by GitHub view raw

The `update_weights` function builds the loss piece by piece, for each of the 2048 positions in the batch.

First the initial observation is passed through the `initial_inference` network to predict the `value`, `reward` and `policy` from the current position. These are used to create the `predictions` list, alongside a given weighting of 1.0.

Then, each action is looped over in turn and the `recurrent_inference` function is asked to predict the next `value`, `reward` and `policy` from the current `hidden_state`. These are appended to the `predictions` list with a weighting of `1/num_rollout_steps` (so that the overall weighting of the `recurrent_inference` function is equal to that of the `initial_inference` function).

We then calculate the loss that compares the `predictions` to their corresponding target values — this is a combination of `scalar_loss` for the `reward` and `value` and `softmax_crossentropy_loss_with_logits` for the `policy`.

The optimises then uses this loss function to train all three of the MuZero networks simultaneously.

So…that's how you train MuZero using Python.

Summary

In summary, AlphaZero is hard-wired to know three things:

- What happens to the board when it makes a given move. For example, if it takes the action 'move pawn from e2 to e4' it knows that the next board position is the same, except the pawn will have moved.

- What the legal moves are in a given position. For example, AlphaZero knows that you can't move 'queen to c3' if your queen is off the board, a piece is blocking the move, or you already have a piece on c3.
- When the game is over and who won. For example, it knows that if the opponent's king is in check and cannot move out of check, it has won.

In other words, AlphaZero can imagine possible futures because it knows the rules of the game.

MuZero is denied access to these fundamental game mechanics throughout the training process. What is remarkable is that by adding a couple of extra neural networks, it is able to cope with not knowing the rules.

In fact, it flourishes.

Incredibly, MuZero actually improves on AlphaZero's performance in Go. This may indicate that it is finding more efficient ways to represent a position through its hidden representation than AlphaZero can find when using the actual board positions. The mysterious ways in which MuZero is embedding the game in its own mind will surely be an active area of research for DeepMind in the near future.

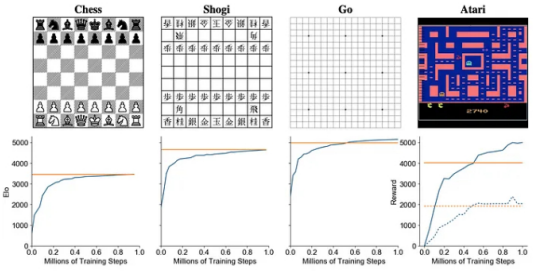


Figure 2: **Evaluation of *MuZero* throughout training in chess, shogi, Go and Atari.** The x-axis shows millions of training steps. For chess, shogi and Go, the y-axis shows Elo rating, established by playing games against *AlphaZero* using 800 simulations per move for both players. *MuZero*'s Elo is indicated by the blue line, *AlphaZero*'s Elo by the horizontal orange line. For Atari, mean (full line) and median (dashed line) human normalized scores across all 57 games are shown on the y-axis. The scores for R2D2 [21], (the previous state of the art in this domain, based on model-free RL) are indicated by the horizontal orange lines. Performance in Atari was evaluated using 50 simulations every fourth time-step, and then repeating the chosen action four times, as in prior work [25].

Summary of MuZero's performance across chess, shogi, Go and Atari games.

To end, I would like to give a brief summary of why I think this development is hugely important for AI.

Why this is a kind of a big deal

AlphaZero is already considered one of the greatest achievements of AI to date, after achieving superhuman prowess at a range of games, with no human expertise required as input.

On the surface, it seems strange then to put so much extra effort into showing that the algorithm isn't hampered by denying it access to the rules. It's a bit like becoming chess world champion, then playing all future competitive matches with your eyes closed. Is this just a party trick?

The answer is that this has never really been about Go, Chess or any other board game for DeepMind. This is about intelligence itself.

When you learnt to swim, you weren't given the rulebook to fluid dynamics first. When you learnt to build towers with blocks, you weren't prepped with Newton's laws of gravity. When you learnt to speak, you did so without knowing any grammar and would probably still struggle to explain all the rules and quirks of language to a non-native speaker, even today.

The point is, life learns without a rulebook.

How this works remains one of the universe's greatest secrets. That's why it is so important that we continue to explore methods for reinforcement learning that do not require direct knowledge of the environment mechanics to plan ahead.

There are similarities between the MuZero paper and the equally impressive WorldModels paper (Ha, Schmidhuber). Both create internal representations of the environment that exist only within the agent and are used to imagine possible futures in order to train a model to achieve a goal. The way in which the two papers achieve this goal is different, but there are some parallels to

be drawn:

- MuZero uses the representation network to embed the current observation, WorldModels uses a variational autoencoder.
- MuZero uses the dynamics network to model the imagined environment, WorldModel uses a recurrent neural network.
- MuZero uses MCTS and a prediction network to select an action, World Models uses an evolutionary process to evolve an optimal action controller.

It's usually a good sign when two ideas that are both groundbreaking in their own way, achieve similar goals. It usually means that there is some deeper underlying truth that both have uncovered — perhaps the two shovels have just hit different parts of the treasure chest.

. . .

This is the end of the three part series 'How To Build Your Own MuZero AI Using Python'. I hope you've enjoyed it.

Please do leave some claps and follow us for more walkthroughs of cutting edge AI.

. . .

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.

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

671 5

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