Homework 5

CMU 10-403: Deep Reinforcement Learning (Spring 2023)

OUT: November 15, 2023
DUE: December 4, 2023 by 11:59pm ET

Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. However, no more than 3 grace days may be applied to a single assignment. Any assignment submitted after 3 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy for more information about grace days and late submissions [2]
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 5". Additionally, export your code ([File → Export .py (if using Colab notebook)]) and upload it the GradeScope assignment titled "Homework 5: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.

[1] `https://www.cmu.edu/policies/`

[2] `https://cmudeeprl.github.io/703website_f23/logistics/`

Collaborators

Please list your name and Andrew ID, as well as those of your collaborators.

Problem 1: MuZero (100 pts)

In this problem you will implement a non-parallelized version of the MuZero algorithm[3] [1] and solve the OpenAI Gym `CartPole-v0` environment. Because a full implementation of MuZero requires considerable effort this handout will provide you with a coding template and asks you to implement important key functions. Our template uses the TensorFlow library to implement and train the different neural networks that are used by MuZero to model policy, states and environment dynamics (Figure 1). Most of the TensorFlow code has been implemented, so you will not need in-depth knowledge of the framework. When downloading the code template you will see a text file that describes the different files in more detail. Your will only implement functions in *mcts.py* and *networks.py*, but knowledge about the other files will help you understand the overall flow of the algorithm. Here is an overview of all files in our implementation:

- *main.py*: In this file, we initialize the MuZero networks for training, the replay buffer, the configuration for the environment, and the environment itself. After this the script will start the self-play learning process. Later in this homework you will study the effects of different hyperparameter choices on MuZero by passing different command line arguments to *main.py*.
- *self_play.py*: Here we control the experience collection and model training iterations of MuZero. In the original MuZero paper, the algorithm is inherently distributed and multiple threads collect experience for the replay buffer in parallel. Another training thread is used to update the networks using this experience. To avoid the difficulties of asynchronous distributed training, for this assignment we only use a *single* thread that *alternates* between collecting experience and training the network. At each epoch we will first collect experience from 20 roll-outs and then perform 30 training steps.
- *config.py*: Here we define some utility functions, model hyperparameters and the training configuration for MuZero.
- *game.py*: This file contains different helper functions for playing a game and storing the final statistics computed during the Monte Carlo Tree Search (MCTS). After MCTS, the game object is passed to the replay buffer and stored for future network updates.
- *replay.py*: Here we implement the replay buffer and the sampling of training data.
- *mcts.py*: Here we place all functions needed for the MCTS. **You will implement multiple key functions in this file.**
- *networks_base.py*: In this file, we specify some abstract Python classes which are used to manage the different MuZero networks.

[3] `https://arxiv.org/pdf/1911.08265.pdf`

- *networks.py*: Here we define the network architectures for the different models and training loop. **You will implement the *update_weights* function in this file.**

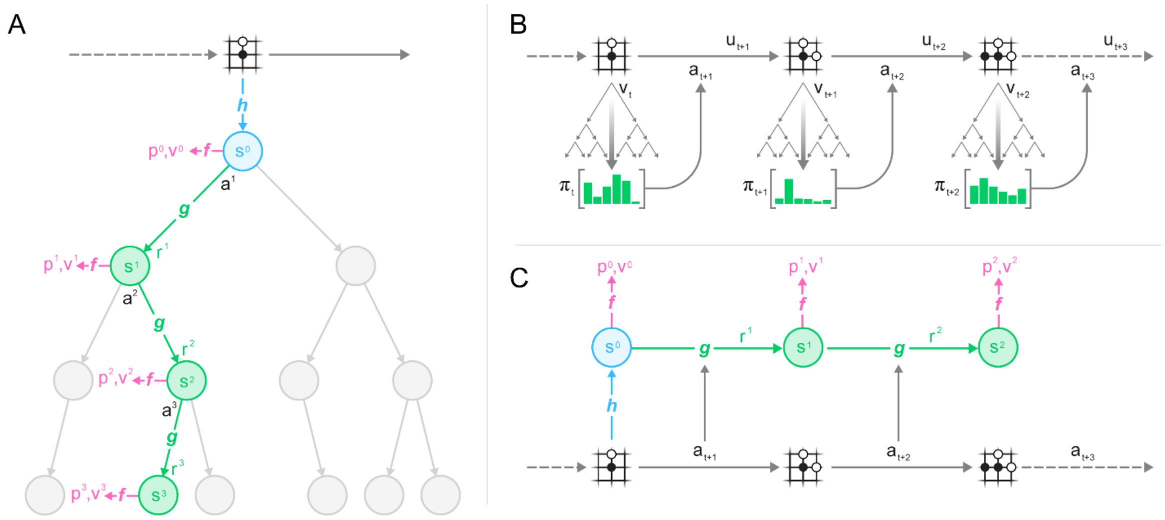

Figure 1: **Planning, acting, and training with a learned model.** (**A**) How MuZero uses its model to plan. The model consists of three connected components for representation, dynamics and prediction. Given a previous hidden state s^{k-1} and a candidate action a^k, the dynamics function g produces an immediate reward r^k and a new hidden state s^k. The policy p^k and value function v^k are computed from the hidden state s^k by a prediction function f. The initial hidden state s^0 is obtained by passing the past observations (e.g. the Go board or Atari screen) into a representation function h. (**B**) How MuZero acts in the environment. A Monte-Carlo Tree Search is performed at each timestep t. An action a_{t+1} is sampled from the search policy π_t, which is proportional to the visit count for each action from the root node. The environment receives the action and generates a new observation o_{t+1} and reward u_{t+1}. At the end of the episode the trajectory data is stored into a replay buffer. (**C**) How MuZero trains its model. A trajectory is sampled from the replay buffer. For the initial step, the representation function h receives as input the past observations $o_1, \ldots, o_t$ from the selected trajectory. The model is subsequently unrolled recurrently for K steps. At each step k, the dynamics function g receives as input the hidden state s^{k-1} from the previous step and the real action a_{t+k}. The parameters of the representation, dynamics and prediction functions are jointly trained, end-to-end by backpropagation-through-time, to predict three quantities: the policy $p^k \approx \pi_{t+k}$, value function $v^k \approx z_{t+k}$, and reward $r_{t+k} \approx u_{t+k}$, where z_{t+k} is a sampled n-step return. Figure and caption taken from MuZero paper [1]

The original MuZero paper models state transitions and rewards both with a single function g. Similarly, MuZero uses a function f which jointly models policy and state values. To stabilize the training process our code template models each individual quantity using a separate network. In *networks.py* we specify the following networks:

- *representation_network*: This network takes as input the observed state from the environment o and maps it to a latent state representation s.
- *dynamic_network*: This network takes as input a latent state representation s and an action a and predicts the latent state representation of the next state s'.

- *value_network*: This network takes as input a latent state representation s and estimates the corresponding state value v_s.
- *policy_network*: This network takes as input a latent state representation s and maps it to the corresponding policy p_s (use a softmax on the output of the policy network to recover the action distribution).
- *reward_network*: This network takes as input a latent state representation s and an action a and predicts the corresponding reward $r_{s,a}$.

We *strongly* recommend that before you start writing your own code, you carefully read the MuZero paper and go through the different files in the code template.

Problem 1.1: Implementation (60 pts)

You will now implement different key functions in the MuZero code template. Copy your implementation of the individual functions into the homework solution template. Your MuZero implementation will form the basis for the further analysis in problem 1.2 and 1.3. If not specified otherwise rely on the hyperparameter choices specified in *config.py*.

1.1.1: MCTS child selection (10 pts)

Implement the *select_child* function in *mcts.py*. During the MCTS search, MuZero uses UCB scoring over min-max normalized Q-values to determine which child node to expand next. The implementation of this function should only require a few lines of code and you can make use of the existing *ucb_score* function.

1.1.2: MCTS expand root/child (20 pts)

Implement the *expand_root* and *expand_child* functions in *mcts.py*. These two functions are important for the MCTS search tree construction. The *expand_root* function should make use of the network's *initial_inference* function to map the raw observation from the environment to a latent state representation using the representation network. Internally, *initial_inference* passes this latent state representation to the value and policy networks to generate the value estimate and and *policy logits* for the current state. Use the returned variables to set the hidden representation and reward attribute of the root node. Set the root node's policy attribute to match the *action selection distribution*. Initialize child nodes for each action by passing the corresponding action selection probability to the Node constructor. Finally, mark the root node as expanded. The *expand_child* function can be implemented analogously by using the network's *recurrent_inference* function instead of *initial_inference*.

1.1.3: MCTS backpropagation (5 pts)

Implement the *backpropagate* function in *mcts.py*. This function updates the value sum and visit counts attributes of all nodes on the path. Traverse through the nodes in reverse order and keep track of the discounted trajectory value. In the MCTS search the value sum determines the UCB exploration and the visit count attribute will guide the final action selection. Use the provided code to update the min_max_stats object with the node value.

1.1.4: MCTS softmax sampling (5 pts)

Implement the *softmax_sample* function in *mcts.py.* MuZero samples the next action that should be executed proportional to the MCTS visit counts. It is important to note that MuZero's softmax sampling *is different* from the regular softmax function. MuZero uses a temperature parameter T to control the degree of exploration. If $T = 0$, MuZero always chooses the action with the highest visit count. For $T > 0$, MuZero selects action $a \in A$ with visit count $N(a)$ with probability

$$p_a = \frac{N(a)^{1/T}}{\sum_{b \in A} N(b)^{1/T}}.$$

1.1.5: Network weight updates (20 pts)

Implement the *update_weights* function in *networks.py.* During training, the MuZero network is unrolled for *num_unroll_steps* steps and its predictions are aligned to sequences sampled from the trajectories previously generated by the MCTS actor. The individual sequences are selected by sampling a states from a game in the replay buffer, then unrolling from that state. Each observation o_t along the sequence has a corresponding MCTS policy π_t, value estimate v_t and environment reward u_t. At each unrolled step k, the MuZero network occurs losses for the value, policy and reward targets which are aggregated to produce the total network loss (see Algorithm 1).

Loss Function Overview:

- *Value Loss*: Cross-Entropy loss between value targets and value outputs from the network's *initial model* or *recurrent model.* We use a cross-entropy loss since the value network outputs a categorical representation of the value functions (this keeps gradients equal regardless of the value size). We also scale the value loss by 0.25. This causes the value loss to be on a similar scale than the reward and policy loss. *Hint*: *tf.nn.softmax_cross_entropy_with_logits* will be useful.

- *Reward Loss*: Mean-Squared Error loss between the predicted rewards and true environment rewards. MuZero does not compute a reward loss for the initial state inference.

- *Policy Loss*: Cross-Entropy loss between policy targets and policy outputs from the network's *initial model* or *recurrent model.*

To maintain gradients of similar magnitude across different unrolling steps, we scale the *true* gradients in two separate locations:

- We scale the gradient of the loss (value + reward + policy) by $\frac{1}{\mathit{num\_unroll\_steps}}$ at each step. This ensures that the total gradient has similar magnitude regardless of how many steps we unroll for.

- We also scale the gradient at the start of the dynamics function by $\frac{1}{2}$. This ensures that the total gradient applied to the dynamics function is of constant magnitude.

To improve the learning process and bound the activations, we add a *tanh* activation to the representation network output. While standard MuZero performs min-max scaling, we use *tanh* for simplicity.

Finally, carefully look at what the replay buffer returns when sampled. It returns a 4-tuple of (*state_batch*, *targets_init_batch*, *targets_recurrent_batch*, *actions_batch*). This tuple entries describe the initial environment states, the initial targets for those states, the targets for the recurrent states, and the corresponding actions taken.

Algorithm 1 Training MuZero

```
procedure UPDATE_WEIGHTS(config $C$, network $N$, optimizer $O$, batch $B$)
    Get representations $R$, values $V$, policies $\pi$ for initial state batch with *initial model*.
    Compute value loss $\ell_v$ and policy loss $\ell_p$ with targets. Let $\ell = 0.25\ell_v + \ell_p$.
    for actions, targets in zip(actions_batch, targets_recurrent_batch):
        Concatenate current latent state representation $R$ and one hot encoded actions $A$ to form matrix $[R|A]$.
        Get new representation $R$, rewards $r$, values $V$, policies $\pi$ from *recurrent model*.
        Compute value loss $\ell_v$, policy loss $\ell_p$ and reward loss $\ell_r$ with targets
        Sum losses to form overall loss for step $\ell_{step} = 0.25\ell_v + \ell_p + \ell_r$
        Scale gradient of $\ell_{step}$ by $\frac{1}{num\_unroll\_steps}$
        Add current step loss $\ell_{step}$ to overall loss $\ell$.
        Scale gradient of latent representation $R$ by $\frac{1}{2}$
    Take a gradient step w.r.t the total loss for given batch with optimizer $O$
end procedure
```

Problem 1.2: Running MuZero (20 pts)

After you completed your implementation, run MuZero with the provided hyperparameters (i.e., you do not need to pass any additional information to *main.py*). All hyperparameters can be found in *main.py* and *config.py*. Train MuZero for 50 epochs or until convergence (the training will stop automatically if the average test reward reaches 195). At each epoch our implementation first collects 20 trajectories using MCTS, then trains the networks for 30 steps and then runs 10 test episodes. Our solution convergence in about ≤ 30 epochs, but the exact time may vary due to randomness in the collected trajectories. Create three plots for (i) the test reward, (ii) the value, reward and policy losses and (iii) the total loss. As x-axis use the number of training steps in the loss plots and the number of test episodes in the reward plot. The plotting code is provided in the *TestResults* and *TrainResults* classes. Feel free to modify the plotting code if you like. Optionally, you can apply some smoothing to the reward plot.

Do not be surprised if your policy loss is rather constant. The `CartPole-v0` environment has many states in which both possible actions can lead to a successful trajectory. Reason how this is connected to the behavior of the policy loss.

Problem 1.3: Effects of Hyperparameters (10 pts)

MuZero requires us to set a large number of hyperparameters that define the training process. In this problem, we will study the effects of the *num_simulations* parameter. The role of the *num_simulations* parameter is to control the number of simulations which is performed during the MCTS search. This is a vital hyperparameter, since the MCTS provides the targets we use to train the networks. Experiment with *num_simulations* $\in \{10, 50, 100\}$. For each hyperparameter value, train MuZero for 50 epochs or until convergence (the training will stop automatically if the average test reward reaches 195). Then, for each hyperparameter setting, create a separate set of three plots for (i) the test reward, (ii) the value, reward and policy losses and (iii) the total loss. As a result, you should have 9 plots in total. Describe how the *num_simulations* parameter affects what you observe in the plots and explain the behavior.

To change the hyperparameter used by MuZero, you can pass the *--num_simulations* argument over the command line to `main.py`.

Problem 1.4: Conceptual Questions (10 pts)

1. Describe the main differences between AlphaZero and MuZero. Give an example for a problem where you would prefer MuZero over AlphaZero.
2. In the MuZero paper the authors describe an additional technique called Reanalyze. Describe what Reanalyze does and how it can improve sample efficiency. How would you implement this in the given code?
3. MuZero has no constraints to learn a consistent hidden state representation. Specifically consider a state o_t, and a state o_{t+1}. When embedded with the initial inference, this gives hidden states s_t and s_{t+1}. When we apply our dynamics to s_t, we should expect that the result $\hat{s}_{t+1}$ aligns with s_{t+1}. How might we enforce this constraint during training?

Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

References

[1] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.