Homework 5 Template

Use this template to record your answers for Homework 5. Add your answers using LaTeXand then save your document as a PDF to upload to Gradescope. You are required to use this template to submit your answers. **You should not alter this template in any way** other than to insert your solutions. You must submit all 17 pages of this template to Gradescope. Do not remove the instructions page(s). Altering this template or including your solutions outside of the provided boxes can result in your assignment being graded incorrectly.

You should also export your code as a .py file and upload it to the **separate** Gradescope coding assignment. Remember to mark all teammates on **both** assignment uploads through Gradescope.

Instructions for Specific Problem Types

On this homework, you must fill in blanks for each problem. Please make sure your final answer is fully included in the given space. **Do not change the size of the box provided.** For short answer questions you should **not** include your work in your solution. Only provide an explanation or proof if specifically asked.

Fill in the blank: What is the course number?

10-703

Problem 0: Collaborators

Enter your team members' names and Andrew IDs in the boxes below. If you worked in a team with fewer than three people, leave the extra boxes blank.

Name 1:	Shrudhi Ramesh Shanthi	Andrew ID 1:	srameshs
Name 2:	Siddharth Ghodasara	Andrew ID 2:	sghodasa
Name 3:	Madhusha Goonesekera	Andrew ID 3:	mgoonese

Problem 1: MuZero (100 pt)

1.1.1: MCTS child selection (10 pt)

Insert code for MCTS child selection.

```python
def select_child(config, node, min_max_stats):
    """
    TODO: Implement this function
    Select a child in the MCTS
    This should be done using the UCB score, which uses the
    normalized Q values from the min max stats
    """

    ucb_scores = -np.inf

    for action, child in node.children.items():
        # ucb_scores.append(ucb_score(config, node, child, min_max_stats))
        ucb_curr = ucb_score(config, node, child, min_max_stats)
        if  ucb_curr > ucb_scores:
            ucb_scores =  ucb_curr
            ucb_action = action
            ucb_child = child

    return ucb_action, ucb_child
    raise NotImplementedError()
```

1.1.2: MCTS expand root/child (20 pts)

Insert code for root and child expansion.

```python
def expand_root(node, actions, network, current_state):
    """
    TODO: Implement this function
    Expand the root node given the current state

    This should perform initial inference, and calculate a softmax policy over children
    You should set the attributes hidden representation, the reward, the policy and children of the node
    Also, set node.expanded to be true
    For setting the nodes children, you should use node.children and instantiate
    with the prior from the policy

    Return: the value of the root
    """
    # get hidden state representation
    value, reward, policy_logits, hidden_rep = network.initial_inference(np.expand_dims(current_state, axis=0))

    node.hidden_representation = hidden_rep
    node.reward = reward

    # Extract softmax policy and set node.policy
    policy = tf.squeeze(tf.nn.softmax(policy_logits)).numpy()

    # instantiate node's children with prior values, obtained from the predicted policy
    for action in actions:
        node.children[action] = Node(policy[action])

    # set node as expanded
    node.expanded = True
    return value
    raise NotImplementedError()
```

```python
def expand_node(node, actions, network, parent_state, parent_action):
    """
    TODO: Implement this function
    Expand a node given the parent state and action
    This should perform recurrent_inference, and store the appropriate values
    The function should look almost identical to expand_root

    Return: value
    """

    # get hidden state representation
    # print("Parent State: ", parent_state)
    value, reward, policy_logits, hidden_rep = network.recurrent_inference(parent_state, parent_action)

    node.hidden_representation = hidden_rep
    node.reward = reward

    # Extract softmax policy and set node.policy
    policy = tf.squeeze(tf.nn.softmax(policy_logits)).numpy()

    # instantiate node's children with prior values, obtained from the predicted policy
    for action in actions:
        node.children[action] = Node(policy[action])

    # set node as expanded
    node.expanded = True

    return value
    raise NotImplementedError()
```

1.1.3: MCTS backpropagation (5 pts)

Insert code for MCTS backpropagation.

```python
def backpropagate(path, value, discount, min_max_stats):
    """
    Backpropagate the value up the path

    This should update a nodes value_sum, and its visit count

    Update the value with discount and reward of node
    """

    for node in reversed(path):
        # TODO: YOUR CODE HERE
        node.visit_count +=1

        node.value_sum += value
        value = value * discount + node.reward

        min_max_stats.update(node.value())
```

1.1.4: MCTS softmax sampling (5 pts)

Insert code for MCTS softmax sampling.

```python
def softmax_sample(visit_counts, temperature):
    """
    Sample an actions

    Input: visit_counts as list of [(visit_count, action)] for each child
    If temperature == 0, choose argmax
    Else: Compute distribution over visit_counts and sample action as in writeup
    """

    # TODO: YOUR CODE HERE
    if temperature == 0:
        result_idx = np.argmax([count for count, _ in visit_counts])
        return visit_counts[result_idx][1]

    else:
        visit_count = np.array([count for count, _ in visit_counts]) ** 1/temperature
        p = visit_count / np.sum(visit_count)
        result_idx = np.random.choice(len(p), p=p)
        return visit_counts[result_idx][1]
    # raise NotImplementedError()
```

1.1.5: Network weight updates (20 pts)

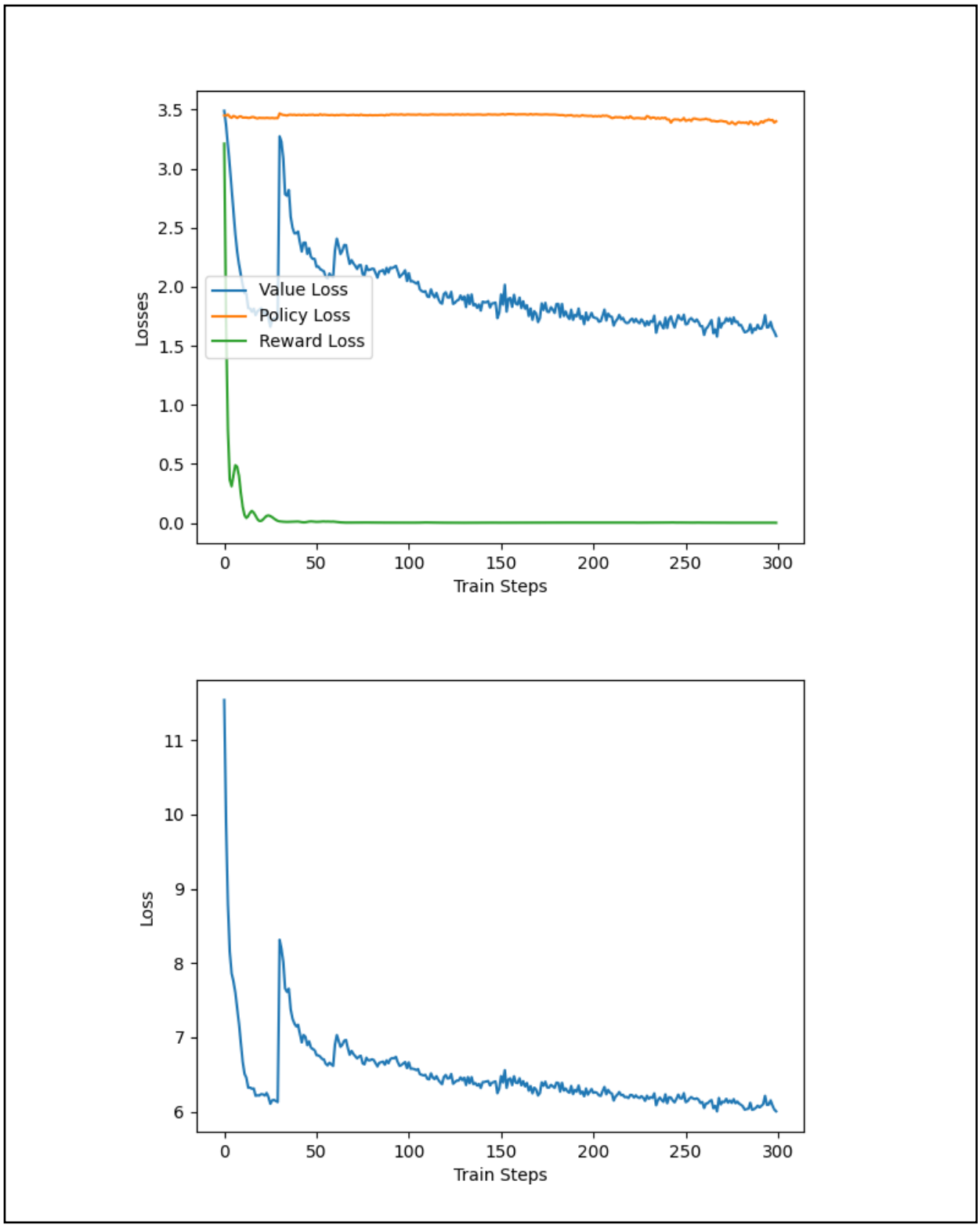
Insert code for network weight updates.

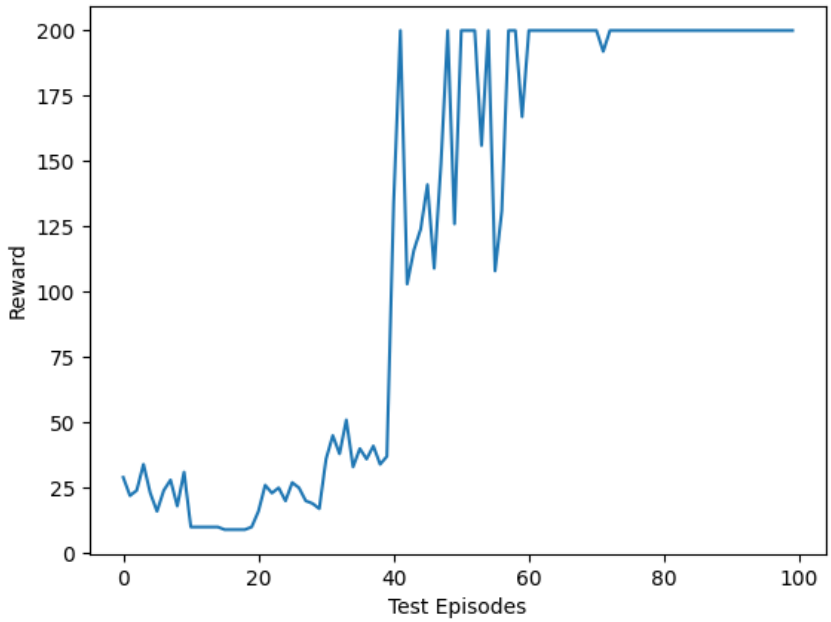
```python
def update_weights(config, network, optimizer, batch, train_results):
    """
    TODO: Implement this function
    Train the network_model by sampling games from the replay_buffer.
    config: A dictionary specifying parameter configurations
    network: The network class to train
    optimizer: The optimizer used to update the network_model weights
    batch: The batch of experience
    train_results: The class to store the train results

    Hints:
    The network initial_model should be used to create the hidden state
    The recurrent_model should be used as the dynamics, which unroll in the latent space.

    You should accumulate loss in the value, the policy, and the reward (after the first state)
    Loss Note: The policy outputs are the logits, same with the value categorical representation
    You should use tf.nn.softmax_cross_entropy_with_logits to compute the loss in these cases
    """
    # for every game in sample batch, unroll and update network_model weights
    def loss():
        loss = 0
        total_value_loss = 0
        total_reward_loss = 0
        total_policy_loss = 0
        (state_batch, targets_init_batch, targets_recurrent_batch,
         actions_batch) = batch

        # YOUR CODE HERE: Perform initial embedding of state batch
        hidden_representation, pred_values, policy_logits = network.initial_model.call(np.array(state_batch)) # called from networks_base
        target_value_batch, _, target_policy_batch = zip(
            *targets_init_batch)
        # Use this to convert scalar value targets to categorical representation
        # This now matches output of initial_model, and can be used with cross entropy loss
        target_value_batch = network._scalar_to_support(
            tf.convert_to_tensor(target_value_batch))
        # YOUR CODE HERE: Compute the loss of the first pass (no reward loss)
        # Remember to scale value loss!
        policy_loss = tf.nn.softmax_cross_entropy_with_logits(labels=target_policy_batch, logits=policy_logits)
        value_loss = tf.nn.softmax_cross_entropy_with_logits(labels=target_value_batch, logits=pred_values)

        policy_loss = tf.math.reduce_mean(policy_loss)
        value_loss = tf.math.reduce_mean(value_loss)

        loss = 0.25*(value_loss) + policy_loss

        for actions_batch, targets_batch in zip(actions_batch, targets_recurrent_batch):
            target_value_batch, target_reward_batch, target_policy_batch = zip(
                *targets_batch)
            # YOUR CODE HERE:
            # Create conditioned_representation: concatenate representations with actions batch
            # Recurrent step from conditioned representation: recurrent + prediction networks
            one_hot_encoded_actions = np.array([np.squeeze(action_to_one_hot(action, config.action_space_size)) for action in actions_batch])
            recurrent_input = tf.concat((hidden_representation, one_hot_encoded_actions), axis=1)
```

```
            hidden_representation, reward, pred_values, policy_logits = network.recurrent_model.call(recurrent_input) # called from networks_base

            # Same as above, convert scalar targets to categorical
            target_value_batch = tf.convert_to_tensor(target_value_batch)
            target_value_batch = network._scalar_to_support(target_value_batch)

            target_policy_batch = tf.convert_to_tensor(target_policy_batch)
            target_reward_batch = tf.convert_to_tensor(target_reward_batch)

            # YOUR CODE HERE: Compute value loss, reward loss, policy loss
            # Remember to scale value loss!
            policy_loss = tf.nn.softmax_cross_entropy_with_logits(labels=target_policy_batch, logits=policy_logits)
            value_loss = tf.nn.softmax_cross_entropy_with_logits(labels=target_value_batch, logits=pred_values)
            reward_loss = MSE(target_reward_batch, reward)

            policy_loss = tf.math.reduce_mean(policy_loss)
            value_loss = tf.math.reduce_mean(value_loss)
            reward_loss = tf.math.reduce_mean(reward_loss)

            total_policy_loss = total_policy_loss + policy_loss
            total_value_loss = total_value_loss + (0.25*value_loss)
            total_reward_loss = total_reward_loss + reward_loss

            loss_step = 0.25*value_loss + policy_loss + reward_loss

            # Add to total losses
            loss = loss + scale_gradient(loss_step, 1/config.num_unroll_steps)

            # YOUR CODE HERE: Half the gradient of the representation
            hidden_representation = scale_gradient(hidden_representation, 1/2)

            # YOUR CODE HERE: Sum the losses, scale gradient of the loss, add to overall loss
        train_results.total_losses.append(loss)
        train_results.value_losses.append(total_value_loss)
        train_results.policy_losses.append(total_policy_loss)
        train_results.reward_losses.append(total_reward_loss)
        return loss
    optimizer.minimize(loss=loss, var_list=network.cb_get_variables())
    network.train_steps += 1
    # raise NotImplementedError()
```

1.2: Running MuZero (20 pts)

Run MuZero, provide the three plots, reason about policy loss behavior.



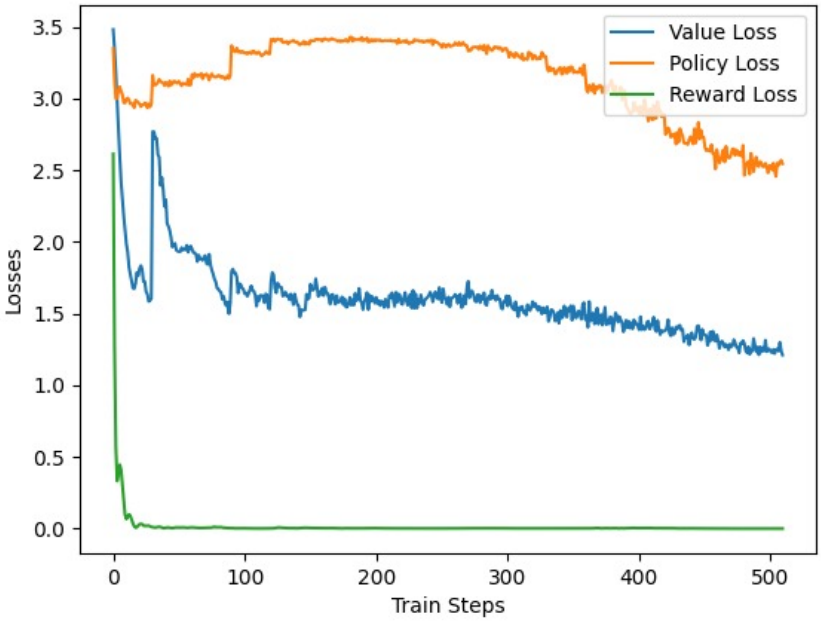


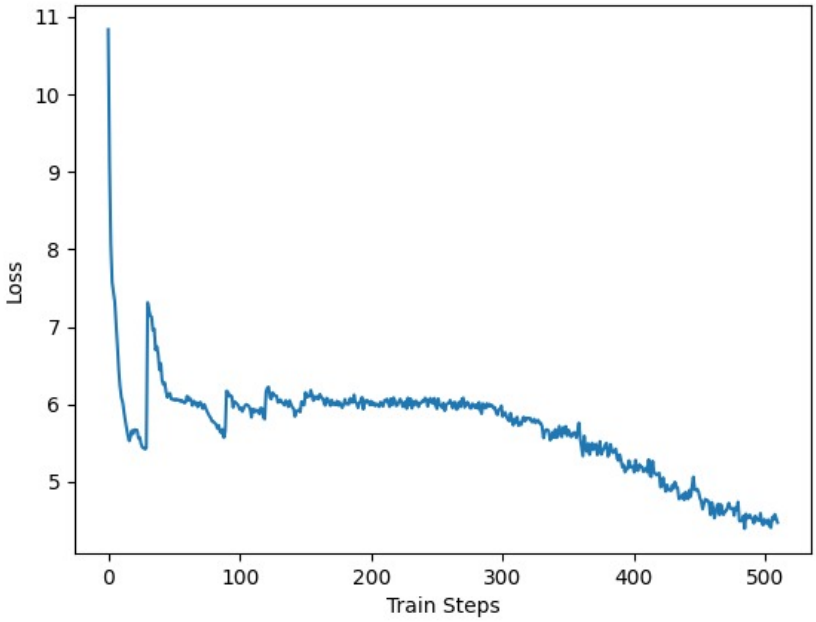
Policy Loss Behaviour:
The plots show that MuZero effectively reduces value and reward losses, with the reward loss reaching near zero quickly, indicating accurate reward prediction. The value loss decreases steadily, suggesting the model learns state-value functions effectively. However, the policy loss remains relatively high and stable, which might indicate difficulty in aligning the policy with optimal actions, possibly due to suboptimal exploration or hyperparameters. The reward plot shows increasing and eventually stabilizing rewards, confirming the model's success in learning a good policy. Further tuning of exploration or policy network capacity might improve the policy loss behavior.

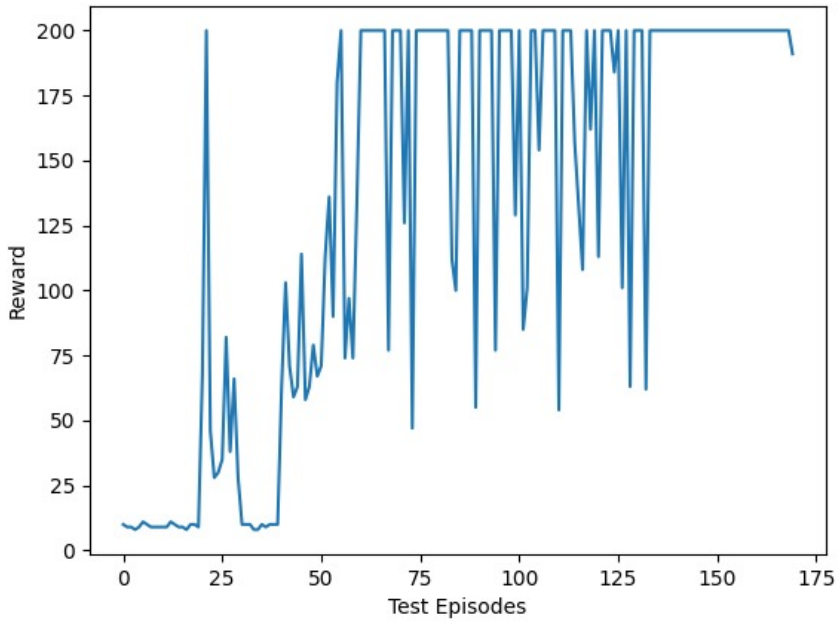
1.3: Effects of Hyperparameters (10 pts)

Run MuZero with different hyperparameters and provide three plots for each value (nine total). Describe and explain the effects of the parameter.

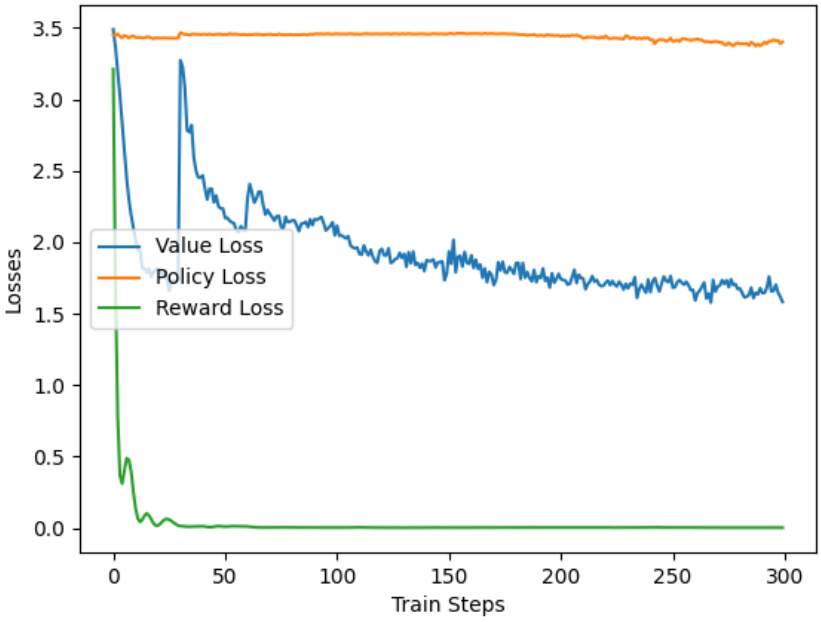
Number of Simulations: **10**

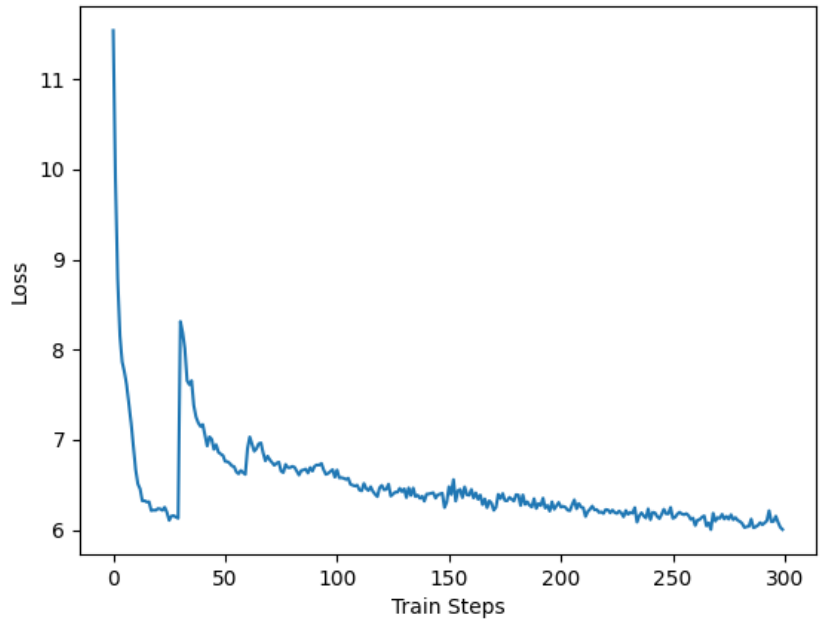


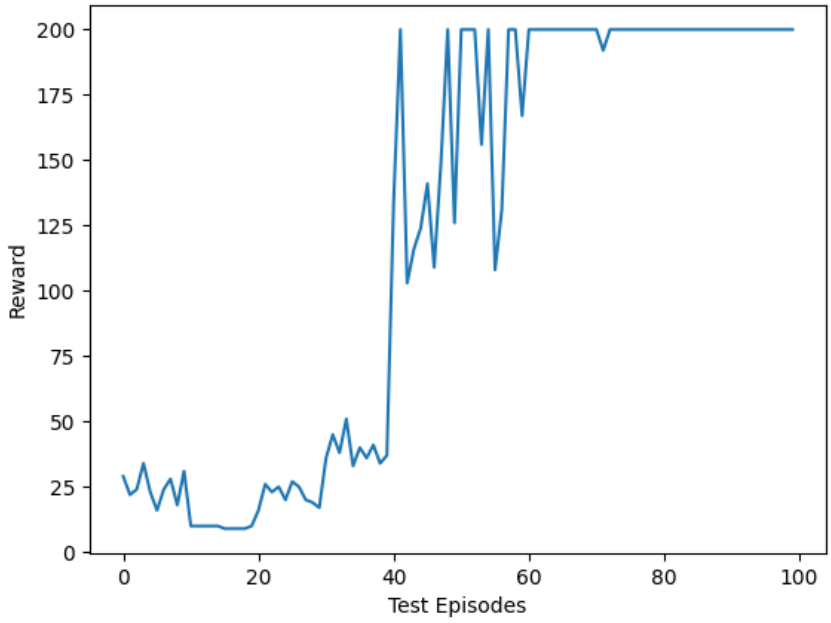




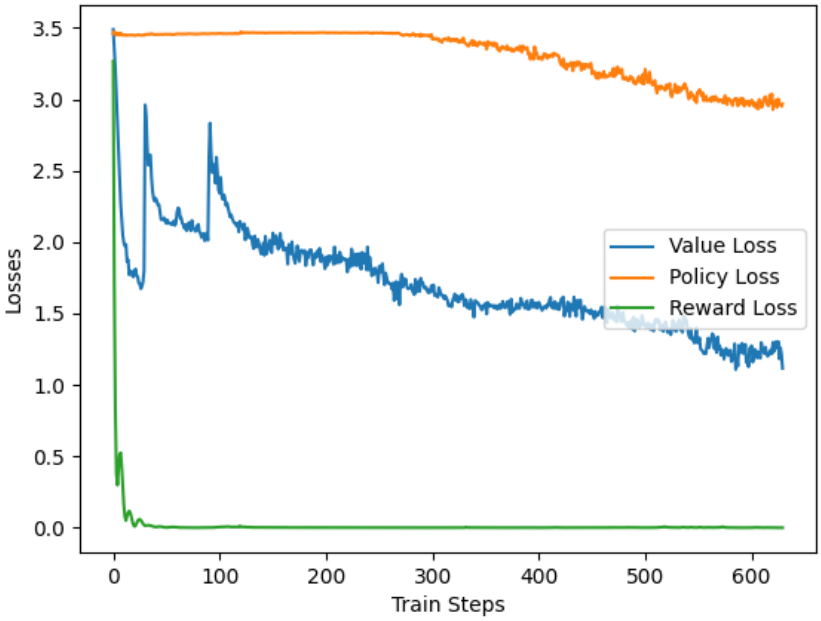
Number of Simulations: **50**

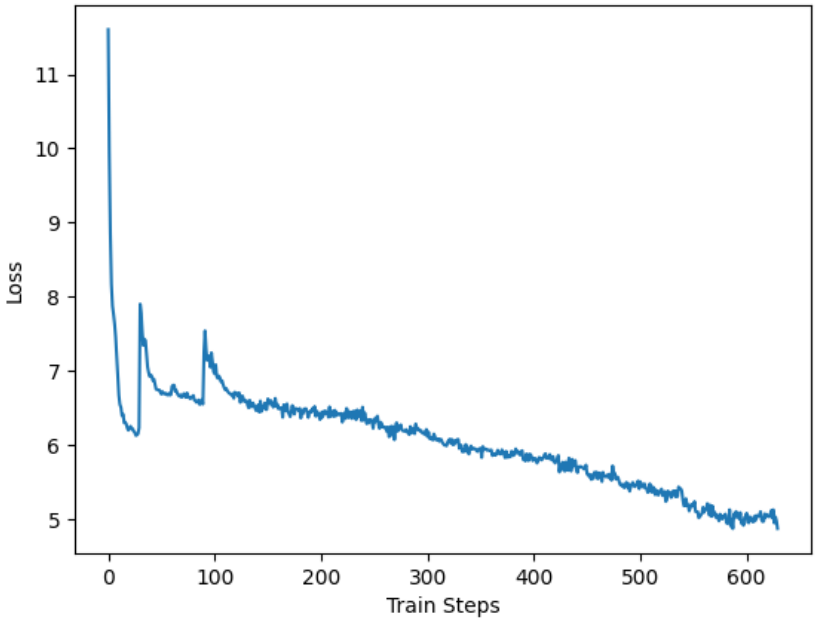


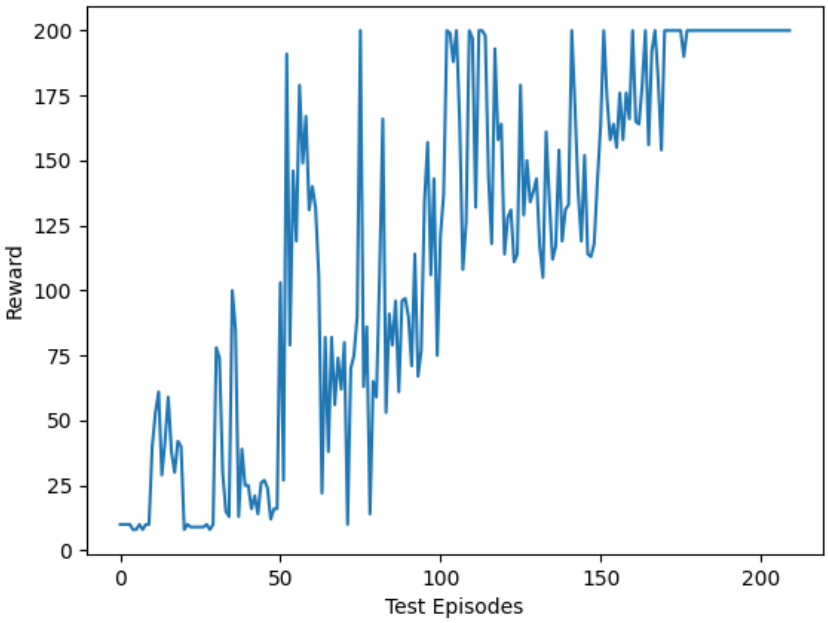

Loss
11
10
9
8
7
6
0
50
100
150
200
250
300
Train Steps


Reward
200
175
150
125
100
75
50
25
0
0
20
40
60
80
100
Test Episodes

Number of Simulations: **100**







Effects of the Number of Simulations Parameter:
Increasing the number of simulations improves MuZero's policy by enabling better-informed decisions, as seen in higher rewards and smoother loss reduction for 50 and 100 simulations. However, the benefits diminish beyond 50 simulations, with slower convergence and higher costs at 100. Hence, 50 simulations appear optimal, offering a balance of efficiency and performance.

1.4: Conceptual Questions (10 pts)

Respond to the three questions.

Q1.
AlphaZero is a reinforcement learning algorithm that relies on perfect knowledge of the environment's rules to predict outcomes and plan actions. In contrast, MuZero extends AlphaZero by learning the environment's dynamics and rules internally, making it applicable to problems where the rules are unknown or too complex to define explicitly. For example, MuZero is better suited for video game environments like Atari games, where explicit rules may not be available but can be learned through interaction. AlphaZero, however, excels in structured domains like chess or Go, where the rules are clearly defined.

Q2.
Reanalyze in MuZero improves sample efficiency by revisiting past experience stored in the replay buffer and re-running Monte Carlo Tree Search (MCTS) using the latest model parameters. This provides updated policy targets and value estimates, which can enhance the training process. To implement this, integrate a mechanism to periodically update stored trajectories in the replay buffer with the current network's outputs and use these refreshed targets during training for a subset of updates. This ensures that the learning process benefits from improved estimates without needing additional environment interactions.

Q3.

To enforce the constraint that the predicted hidden state $\hat{s}_{t+1}$ aligns with s_{t+1}, a consistency loss can be introduced during training. Specifically, after generating s_{t+1} using the initial inference and $\hat{s}_{t+1}$ via the dynamics function applied to s_t, you can minimize the difference between the two states using a loss function, such as mean squared error (MSE). This ensures that the dynamics model learns to transition consistently between hidden states.

Formally:

$$\mathcal{L}_{\text{consistency}} = \|\hat{s}_{t+1} - s_{t+1}\|^2$$

This loss term would be added to the overall training objective, guiding the model to maintain a consistent hidden state representation.

Feedback

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

Solution

Before you provide this assignment to the next batch of students, make sure to update to either provide an environment.txt or to update the assignment to work with the latest libraries that were required for this assignment. Debugging required library versions took the most time for getting everything to work.

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade. Please average your answer over all the members of your team.

Alone	4
With teammates	30
With other classmates	0
At office hours	0