Lab14 Report

st122314

MuZero

1. MuZero function

In [ ]:

```
def muzero(config: MuZeroConfig):
    storage = SharedStorage()
    replay_buffer = ReplayBuffer(config)

    for _ in range(config.num_actors):
        launch_job(run_selfplay, config, storage, replay_buffer)

    train_network(config, storage, replay_buffer)

    return storage.latest_network()
```

The entry point functiion muzero is passed a MuZeroConfig object, which stores important inforamtion about parameter settings such as the action_space_size(number of possible actions) and num_actors (the number of parallel game simulations to run). There are two independent parts to the MuZero algorithm, self-play (creating game data) and training (producing improved versions of the neural network models). The SharedStorage and ReplayBuffer objects can be accessed by both halves of the algorithm in order to store neural network versions and game data.

2. Shared Storage and the Replay Buffer

In [ ]:

```
class SharedStorage(object):

    def __init__(self):
        self._networks = {}

    def latest_network(self) -> Network:
        if self._networks:
            return self._networks[max(self._networks.keys())]
        else:
            # policy -> uniform, value -> 0, reward -> 0
            return make_uniform_network()

    def save_network(self, step: int, network: Network):
        self._networks[step] = network
```

```
The SharedStorage object contains methods for saving a version of the neural
network and retrieving the latest neural network from the store.
```

3. Replay Buffer

In [ ]:

```
class ReplayBuffer(object):

    def __init__(self, config: MuZeroConfig):
        self.window_size = config.window_size
        self.batch_size = config.batch_size
        self.buffer = []

    def save_game(self, game):
        if len(self.buffer) > self.window_size:
            self.buffer.pop(0)
        self.buffer.append(game)

    ...
```

The ReplayBuffer stores data from previous games. The ReplayBuffer class contains a sample_batch method to sample a batch of observations from the buffer.

4. Self-play(run-selfplay)

In [ ]:

```
def run_selfplay(config: MuZeroConfig, storage: SharedStorage,
                 replay_buffer: ReplayBuffer):
    while True:
        network = storage.latest_network()
        game = play_game(config, network)
        replay_buffer.save_game(game)
```

The method plays thousands of games against itself. In the process, the games are saved to a buffer, and then training utilizes the data from those games. This step is the same as AlphaZero.

5. Monte Carlo tree search(MCTS)

In [ ]:

```
class NetworkOutput(typing.NamedTuple):
    value: float
    reward: float
    policy_logits: Dict[Action, float]
    hidden_state: List[float]


class Network(object):

    def initial_inference(self, image) -> NetworkOutput:
        # representation + prediction function
        return NetworkOutput(0, 0, {}, [])

    def recurrent_inference(self, hidden_state, action) -> NetworkOutput:
        # dynamics + prediction function
        return NetworkOutput(0, 0, {}, [])

    def get_weights(self):
        # Returns the weights of this network.
        return []

    def training_steps(self) -> int:
        # How many steps / batches the network has been trained for.
        return 0
```

In terms of the pseudocode, there are two key inference functions used to move through the MCTS tree making predictions:

- initial_inference for the current state. Calls h followed by f.
- recurrent_inference for moving between states inside the MCTS tree. Calls g followed by f.

6. Playing a game

In [ ]:

```
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
```

A game is a loop. The game ends when a terminal condition is met or the maximum number of moves is reached.

When a new game is started, MCTS must be started over at the root node.

In [ ]:

```
class Node(object):

    def __init__(self, prior: float):
        self.visit_count = 0
        self.to_play = -1
        self.prior = prior
        self.value_sum = 0
        self.children = {}
        self.hidden_state = None
        self.reward = 0

    def expanded(self) -> bool:
        return len(self.children) > 0

    def value(self) -> float:
        if self.visit_count == 0:
            return 0
        return self.value_sum / self.visit_count
```

In [ ]:

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

Then we request the game to return the current observation

In [ ]:

```
def expand_node(node: Node, to_play: Player, actions: List[Action],
                network_output: NetworkOutput):
    node.to_play = to_play
    node.hidden_state = network_output.hidden_state
    node.reward = network_output.reward
    policy = {a: math.exp(network_output.policy_logits[a]) for a in actions}
    policy_sum = sum(policy.values())
    for action, p in policy.items():
        node.children[action] = Node(p / policy_sum)
```

Next, we 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.

In [ ]:

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

In [ ]:

```
def add_exploration_noise(config: MuZeroConfig, node: Node):
    actions = list(node.children.keys())
    noise = numpy.random.dirichlet([config.root_dirichlet_alpha] * len(actions))
    frac = config.root_exploration_fraction
    for a, n in zip(actions, noise):
        node.children[a].prior = node.children[a].prior * (1 - frac) + n * frac
```

We need to add exploration noise to the root node actions, to ensure that MCTS explores a range of possible actions rather than only exploring the action which it currently believes to be optimal.

In [ ]:

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

7. MCTS run function

In [ ]:

```
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
```

In [ ]:

```
def backpropagate(search_path: List[Node], value: float, to_play: Player,
                  discount: float, min_max_stats: MinMaxStats):
    for node in search_path:
        node.value_sum += value if node.to_play == to_play else -value
        node.visit_count += 1
        min_max_stats.update(node.value())

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

In [ ]:

```
def run_mcts(config: MuZeroConfig, root: Node, action_history: ActionHistory,
             network: Network):
    min_max_stats = MinMaxStats(config.known_bounds)

    for _ in range(config.num_simulations):
        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)
```

In [ ]:

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

In [ ]:

```
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.
```

8. Training

In [ ]:

```
def train_network(config: MuZeroConfig, storage: SharedStorage,
                  replay_buffer: ReplayBuffer):
    network = Network()
    learning_rate = config.lr_init * config.lr_decay_rate**(
        tf.train.get_global_step() / config.lr_decay_steps)
    optimizer = tf.train.MomentumOptimizer(learning_rate, config.momentum)

    for i in range(config.training_steps):
        if i % config.checkpoint_interval == 0:
            storage.save_network(i, network)
        batch = replay_buffer.sample_batch(config.num_unroll_steps, config.td_steps)
        update_weights(optimizer, network, batch, config.weight_decay)
    storage.save_network(config.training_steps, network)
```

It first creates a new Network object (that stores randomly initialised 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.

9. MuZero Loss Function

In [ ]:

```
def update_weights(optimizer: tf.train.Optimizer, network: Network, batch,
                   weight_decay: float):
    loss = 0
    for image, actions, targets in batch:
        # Initial step, from the real observation.
        value, reward, policy_logits, hidden_state = network.initial_inference(
            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)
```

Conclusion

In this lab, I have leant about MuZero and AlphaZeo and Evaluaiotn of MiZero. AlphaZero (AZ) is a more generalized variant of the AlphaGo Zero (AGZ) algorithm, and is able to play shogi and chess as well as Go while MuZero (MZ) combines the AlphaZero (AZ) algorithm's high-performance planning with model-free reinforcement learning methodologies. The combination allows for more effective training in traditional planning regimes like Go, as well as domains with significantly more complex inputs at each level, such visual video games.

Form my understanding, the evaluation of Muzero are the following.

- AlphaGo becomes the first program to mater Go using meural networks and tree seach. (Jan 2016, Nature)
- AlphaGo Zero learns to play completely on its own, without human knowlwdege. (Oct 2017, Nature)
- AlphaZero maters three perfect imforamtion games using a single algorithm for all games. (Dec 2018, Science)
- MuZero learns the rules the game, allowing it to also master enviroments with unknown dynamics. (Dec 2020, Nature)

In [ ]:

```

```