AlphaZeroConfig Class

```
class AlphaZeroConfig(object):
  def __init__(self):
    # Self-Play
    self.num_actors = 5000
    self.num_sampling_moves = 30
    self.max_moves = 512
    self.num_simulations = 800
    self.root_dirichlet_alpha = 0.3
    self.root_exploration_fraction = 0.25
    self.pb_c_base = 19652
    self.pb_c_init = 1.25

    # Training
    self.training_steps = int(700e3)
    self.checkpoint_interval = int(1e3)
    self.window_size = int(1e6)
    self.batch_size = 4096
    self.weight_decay = 1e-4
    self.momentum = 0.9
    self.learning_rate_schedule = {
        0: 2e-1,
        100e3: 2e-2,
        300e3: 2e-3,
        500e3: 2e-4
    }
```

- Configuration class for AlphaZero algorithm.
- Defines hyperparameters for both self-play and training phases.
- Specifies the number of actors (self-play processes), exploration parameters, and training-related parameters such as steps, intervals, window size, batch size, weight decay, momentum, and learning rate schedule.

Node Class

```
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 = {}

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

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

- Represents a node in the Monte Carlo Tree Search (MCTS) algorithm.
- `visit_count`: Number of times this node has been visited.
- `to_play`: The player to make a move at this node.
- `prior`: The prior probability assigned by the neural network.
- `value_sum`: The sum of values encountered during simulations.
- `children`: Dictionary of child nodes representing possible actions.
- `expanded()`: Checks if the node has been expanded (has children).
- `value()`: Calculates the average value of the node.

Game Class

```
class Game(object):
  def __init__(self, history=None):
    self.history = history or []
    self.child_visits = []
    self.num_actions = 4672

  def terminal(self):
    pass

  def terminal_value(self, to_play):
    pass

  def legal_actions(self):
    return []

  def clone(self):
    return Game(list(self.history))

  def apply(self, action):
    self.history.append(action)

  def store_search_statistics(self, root):
    sum_visits = sum(child.visit_count for child in root.children.values())
    self.child_visits.append([
        root.children[a].visit_count / sum_visits if a in root.children else 0
        for a in range(self.num_actions)
    ])

  def make_image(self, state_index: int):
    return []

  def make_target(self, state_index: int):
    return (self.terminal_value(state_index % 2),
            self.child_visits[state_index])
  def to_play(self):
    return len(self.history) % 2
```

- Represents the state of the game.
- `history`: List of actions representing the game history.
- `child_visits`: Records visit counts of child nodes during MCTS.
- `num_actions`: Number of possible actions in the game.
- Methods:
    - `terminal()`: Checks if the game is in a terminal state.
    - `terminal_value(to_play)`: Returns the value of the terminal state.
    - `legal_actions()`: Returns legal actions at the current state.
    - `clone()`: Creates a copy of the game state.
    - `apply(action)`: Applies an action to the game state.
    - `store_search_statistics(root)`: Stores visit statistics for child nodes.
    - `make_image(state_index)`: Constructs a game-specific feature representation.
    - `make_target(state_index)`: Constructs a target value for training.
    - `to_play()`: Returns the player to play at the current state.

ReplayBuffer Class

```
class ReplayBuffer(object):
  def __init__(self, config: AlphaZeroConfig):
    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)

  def sample_batch(self):
    move_sum = float(sum(len(g.history) for g in self.buffer))
    games = numpy.random.choice(
        self.buffer,
        size=self.batch_size,
        p=[len(g.history) / move_sum for g in self.buffer])
    game_pos = [(g, numpy.random.randint(len(g.history))) for g in games]
    return [(g.make_image(i), g.make_target(i)) for (g, i) in game_pos]
```

- Manages a replay buffer of past games for training.
- `window_size`: Maximum size of the replay buffer.
- `batch_size`: Size of batches to sample during training.
- `buffer`: List of stored games.
- `save_game(game)`: Adds a game to the replay buffer and removes the oldest game if the buffer exceeds the window size.
- `sample_batch()`: Samples a batch of games uniformly across positions.

Network Class

```
class Network(object):
  def inference(self, image):
    return (-1, {})

  def get_weights(self):
    return []
```

- Represents the neural network used for evaluation.
- `inference(image)`: Returns a tuple containing the value and policy logits for a given input image.
- `get_weights()`: Returns the weights of the network.

SharedStorage Class

```
class SharedStorage(object):
  def __init__(self):
    self._networks = {}

  def latest_network(self) -> Network:
    if self._networks:
      return self._networks[max(self._networks.keys())]
    else:
      return make_uniform_network()

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

- Maintains a collection of network snapshots during training.
- `_networks`: Dictionary mapping training step to the corresponding network snapshot.
- `latest_network()`: Returns the latest network snapshot.
- `save_network(step, network)`: Saves a network snapshot at a specific training step.

AlphaZero Function

```
def alphazero(config: AlphaZeroConfig):
  storage = SharedStorage()
  replay_buffer = ReplayBuffer(config)

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

  train_network(config, storage, replay_buffer)

  return storage.latest_network()
```

- Orchestrates the self-play and training phases of the

AlphaZero algorithm.

- `storage`: Shared storage for network snapshots.
- `replay_buffer`: Replay buffer for training data.
- Launches self-play processes and then trains the network.
- Returns the latest trained network.

Self-Play Functions

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

def play_game(config: AlphaZeroConfig, network: Network):
  game = Game()
  while not game.terminal() and len(game.history) < config.max_moves:
    action, root = run_mcts(config, game, network)
    game.apply(action)
    game.store_search_statistics(root)
  return game

def run_mcts(config: AlphaZeroConfig, game: Game, network: Network):
  root = Node(0)
  evaluate(root, game, network)
  add_exploration_noise(config, root)

  for _ in range(config.num_simulations):
    node = root
    scratch_game = game.clone()
    search_path = [node]

    while node.expanded():
      action, node = select_child(config, node)
      scratch_game.apply(action)
      search_path.append(node)

    value = evaluate(node, scratch_game, network)
    backpropagate(search_path, value, scratch_game.to_play())
  return select_action(config, game, root), root

def select_action(config: AlphaZeroConfig, game: Game, root: Node):
  visit_counts = [(child.visit_count, action)
                  for action, child in root.children.items()]
  if len(game.history) < config.num_sampling_moves:
    _, action = softmax_sample(visit_counts)
  else:
    _, action = max(visit_counts)
  return action

def select_child(config: AlphaZeroConfig, node: Node):
  _, action, child = max((ucb_score(config, node, child), action, child)
                         for action, child in node.children.items())
  return action, child

def ucb_score(config: AlphaZeroConfig, parent: Node, child: Node):
  pb_c = math.log((parent.visit_count + config.pb_c_base + 1) /
                  config.pb_c_base) + config.pb_c_init
  pb_c *= math.sqrt(parent.visit_count) / (child.visit_count + 1)

  prior_score = pb_c * child.prior
  value_score = child.value()
  return prior_score + value_score

def evaluate(node: Node, game: Game, network: Network):
  value, policy_logits = network.inference(game.make_image(-1))
  node.to_play = game.to_play()
  policy = {a: math.exp(policy_logits[a]) for a in game.legal_actions()}
  policy_sum = sum(policy.values())
  for action, p in policy.items():
    node.children[action] = Node(p / policy_sum)
  return value

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

def add_exploration_noise(config: AlphaZeroConfig, node: Node):
  actions = list(node.children.keys())
  noise = numpy.random.gamma(config.root_dirichlet_alpha, 1, 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
```

- The self-play functions that generate game data through MCTS simulations.
- `run_selfplay`: Runs an infinite loop where it retrieves the latest network and generates a game using MCTS, saving it to the replay buffer.
- `play_game`: Generates a single game using MCTS until a terminal state or maximum moves are reached.
- `run_mcts`: Core MCTS algorithm to decide on actions.
- `select_action`: Selects an action based on visit counts during MCTS.
- `select_child`: Selects the child node with the highest UCB score.
- `ucb_score`: Calculates the UCB score for a child node.
- `evaluate`: Uses the neural network to obtain a value and policy prediction for a given game state.
- `backpropagate`: Propagates the evaluation up the tree to update visit counts and values.
- `add_exploration_noise`: Adds Dirichlet noise to the prior of the root to encourage exploration.

Training Functions

```
def train_network(config: AlphaZeroConfig, storage: SharedStorage,
replay_buffer: ReplayBuffer):
  network = Network()
  optimizer = tf.train.MomentumOptimizer(config.learning_rate_schedule,
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()
    update_weights(optimizer, network, batch, config.weight_decay)
  storage.save_network(config.training_steps, network)

def update_weights(optimizer: tf.train.Optimizer, network: Network, batch,
weight_decay: float):
  loss = 0
  for image, (target_value, target_policy) in batch:
    value, policy_logits = network.inference(image)
    loss += (
        tf.losses.mean_squared_error(value, target_value) +
        tf.nn.softmax_cross_entropy_with_logits(
            logits=policy_logits, labels=target_policy))

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

  optimizer.minimize(loss)
```

- The training functions that update the neural network weights using training data.
- `train_network`: Main training loop that iterates over training steps, saves network checkpoints, samples batches from the replay buffer, and updates the weights.
- `update_weights`: Updates the network weights based on the loss calculated from value and policy predictions, and applies weight decay.

Stubs

```
def softmax_sample(d):
  return 0, 0

def launch_job(f, *args):
  f(*args)

def make_uniform_network():
  return Network()
```

- Stubs for functions that are not explicitly defined in the pseudocode but are mentioned, such as `softmax_sample`, `launch_job`, and `make_uniform_network`.