AlphaZero Code review

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.
- Attributes:
  - num_actors: Determines how many parallel actors will be trained. This is a parallelization feature that we have little use for. It used in the `alphazero()` function. It will most likely be removed.
  - num_sample_moves: Controls the trade-off between exploration and exploitation. If less than num_sample_moves have been made through the game, the action is sampled using a softmax function. If more than that amount of moves has been performed, the action taken is the one with the maximum number of visits, favouring exploitation. Used in the `select_action()`
  - max_moves: This parameter determines how many moves a game can have during the self-play phase. It is used in the `play_game()` function. With this, we can limit the amount of decisions we assign to an episode.

  - max_simulations: Determines the number of times the tree search is run every step. That is, the number of simulations that occur from the root state to either a terminal state or to a number of steps greater than the limit. Used in `run_mcts()`.
  - root_dirichlet_alpha and root_exploration_fraction: These paramaters are used in `add_exploration_noise()` to control the amount of exploration of the action space.
  - pb_c_base and pb_c_init: used in the Upper confidence bound formula, in function `ucb_score()`.
  - Training parameters:
    - training_steps: The total number of training steps to perform.
    - checkpoint_interval: Interval at which to save network checkpoints during training.
    - window_size: Maximum size of the replay buffer.
    - batch_size: Size of batches sampled from the replay buffer during each training step.
    - weight_decay: Weight decay coefficient applied during weight updates.
    - momentum: Momentum parameter used in the MomentumOptimizer.
    - learning_rate_schedule: A dictionary defining the learning rate schedule over different training steps.

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

  def value(self) -> float:
    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. It will need to be modified to include the state of the simulation.
- Attributes:
  - `visit_count`: Number of times this node has been visited. Used to calculate the averagevalue of the node over multiple simulations in the `value()` method.
  - `to_play`: Player to make a move at this node. Either -1 or 1. This will most likely be removed, as we do not have as of yet a formulation of decision making in autonomous driving in terms of an adversarial task.
  - `prior`: Prior probability assigned by the neural network. It is obtained from the policy (neural network output for the state associated to this node). It is used in the UCB formula (`ucb_score()`) to balance exploration and exploitation.
  - `value_sum`: Sum of values encountered during simulations.

    - `children`: Dictionary of child nodes representing possible actions. Keys are the possible actions from this node, and values are instances of the Node class that represent future states.
- Methods:
    - `expanded()`: Checks if the node has been expanded (has children).
    - `value()`: Returns the average value of the node.

Game Class

```
class Game(object):
  """
  Represents the state of the game.

  Attributes:
    history (List[int]): List of actions representing the game history.
        It records the sequence of actions taken during the game.
    child_visits (List[List[float]]): Stores the visit count
distribution
        of child nodes for each state in the game.
    num_actions (int): Represents the size of the action space for the
game.
        It is the total number of possible actions that can be taken by
a player.
  """
  def __init__(self, history: List[int] = None):
    """
    Initializes a new Game instance.

    Args:
        history (List[int], optional): List of actions representing the
game history.
            Defaults to an empty list.
    """
    self.history = history or []
    self.child_visits = []
    self.num_actions = 4672  # action space size for chess; 11259 for
shogi, 362 for Go

  def terminal(self) -> bool:
    """
    Checks if the game is in a terminal state.

    Returns:
        bool: True if the game is in a terminal state, False otherwise.
    """
    # Game specific termination rules.
    pass

  def terminal_value(self, to_play: int) -> float:
    # Game specific value.
    """
    Returns the reward associated with the terminal state of the
current game.

```

```
        Args:
            to_play (int): The player to play at the terminal state.

        Returns:
            float: The terminal value indicating the outcome or score of
    the game.
        """
        pass

    def legal_actions(self) -> List[int]:
        # Game specific calculation of legal actions.
        """
        Returns legal actions at the current state.

        Returns:
            List[int]: List of legal actions.
        """
        return []

    def clone(self) -> 'Game':
        """
        Creates a copy of the game state.

        Returns:
            Game: A new instance representing a copy of the game state.
        """
        return Game(list(self.history))

    def apply(self, action: int):
        """
        Applies an action to the game state.

        Args:
            action (int): The action to be applied.

        Notes:
            This method interacts with the Carla client to execute the
    action
            and updates the game state based on the client's response.
        """
        #self.history.append(action)
        pass

    def store_search_statistics(self, root: 'Node'):
        """
        Stores visit statistics for child nodes.

        Args:
            root (Node): The root node of the search tree.
        """
        sum_visits = sum(child.visit_count for child in
    root.children.itervalues())
        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) -> List[numpy.array]:
    """
    Constructs a game-specific feature representation.

    Args:
        state_index (int): The index of the current game state.

    Returns:
        List[numpy.array]: List of feature planes representing the game
state.
    """
    # Game specific feature planes.
    return []

  def make_target(self, state_index: int) -> Tuple[float, List[float]]:
    """
    Constructs a target tuple for training.

    Args:
        state_index (int): The index of the current game state.

    Returns:
        Tuple[float, List[float]]: Target value and policy for training
the neural network.
    """
    return (self.terminal_value(state_index % 2),
            self.child_visits[state_index])

  def to_play(self) -> int:
    return len(self.history) % 2
```

- Represents the state of the game.
- Attributes:
  - `history`: List of actions representing the game history. It works here because the state of a game of chess can be recreated from the initial state (which is always the same) and a list of movements performed by either player. In our case, this history should be a `List['Node']`, such that the game can be backtracked from the current state to any other that came previously.
  - `child_visits`: Stores the visit count distribution of child nodes for each state in the game. This information is used during training to guide the network towards actions with higher visit counts.
  - `num_actions`: Represents the size of the action space for the game. It is the total number of possible actions that can be taken by a player. In our case, it will hover between 3 and 5.
- Methods:
  - `terminal()`: Checks if the game is in a terminal state. To be implemented -or subclassed-. In our case, termination is reached when the ego vehicle has travelled a certain distance or when a collision occurs.
  - `terminal_value(to_play)`: Returns the reward associated to the terminal state of the current game. The `to_play` variable will disappear.

- `legal_actions()`: Returns legal actions at the current state. A possible refactor of this method is to substitute it with an attribute, as in principle every action will be legal in any state. However, the logic to allow lane shifting can be implemented here: if on the left lane, disallow left lane changes.
- `clone()`: Creates a copy of the game state.
- `apply(action)`: Applies an action to the game state. This function will need to include the necessary logic to enforce an action by the agent. Thus, it will send the action to the corresponding logic in the carla client, and receive a new state, which will be appended to the `history` attribute.
- `store_search_statistics(root)`: Stores visit statistics for child nodes. It records a history of visit statistics, to show how the visit distribution of the actions -and child states- evolves with the game.
    - Correlation to Game States:
        - The child_visits list accumulates visit count distributions for different game states during the self-play phase. Each entry in this list corresponds to a specific game state and how the agent perceived the value of different actions from that state.
    - Training the Neural Network:
        - During the training phase, you can sample batches of these distributions along with their corresponding game states to train the neural network.
        - The neural network is trained to predict both the value (expected outcome) and the policy (probability distribution over actions) based on the input game state.
    - Guiding Training with Exploration History:
        - The historical information in child_visits guides the training process by emphasizing actions that were explored more frequently during the self-play phase.
        - Actions with higher visit counts are considered more reliable or desirable based on the agent's exploration and evaluation of the game tree.
- `make_image(state_index)`: Constructs a game-specific feature representation. This fits nicely with our implementation of an autoencoder that extracts potential field information from the scene. However, it is not clear where this function needs to be called, and moreover, it can lead to an excessive increase of memory usage, as instead of storing an array of X by Y entries, as dictated by the encoder, we need to store the potential fields themselves.
- `make_target(state_index)`: Constructs a target value for training. It collects the value associated to a given state, as well as the policy (child visits). In our case, we need not include the modulo operator to choose between players, since the ego vehicle is the only entity from which we collect experiences.
- `to_play()`: Returns the player to play at the current state. This is not useful for our application

ReplayBuffer Class

```
class ReplayBuffer(object):
  """
```

```
  A replay buffer for storing and sampling self-play game data.

  Attributes:
    window_size (int): The maximum size of the replay buffer.
        When the buffer exceeds this size, old games are discarded.
    batch_size (int): The size of batches to be sampled during
training.
    buffer (List[Game]): A list to store self-play games.
  """

  def __init__(self, config: 'AlphaZeroConfig'):
    """
    Initializes a new ReplayBuffer instance.

    Args:
      config (AlphaZeroConfig): Configuration object containing
parameters.
    """
    self.window_size = config.window_size
    self.batch_size = config.batch_size
    self.buffer = []

  def save_game(self, game: 'Game'):
    """
    Saves a self-play game to the replay buffer.

    Args:
      game (Game): The self-play game to be saved.

    Notes:
      If the buffer exceeds the maximum window size, old games are
discarded.
    """
    if len(self.buffer) > self.window_size:
      self.buffer.pop(0)
    self.buffer.append(game)

  def sample_batch(self) -> List[Tuple[List[numpy.array], Tuple[float,
List[float]]]]:
    """
    Samples a batch of self-play game data for training.

    Returns:
      List[Tuple[List[numpy.array], Tuple[float, List[float]]]]:
          A list of tuples containing game states (images) and their
target values (value, policy).
    """
    # Sample uniformly across positions.
    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.
- Attributes
  - `window_size`: Maximum size of the replay buffer.
  - `batch_size`: Size of batches to sample during training.
  - `buffer`: List of stored games.
- Methods
  - `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.
    - It calculates the total number of moves across all games in the buffer using sum(len(g.history) for g in self.buffer). This sum represents the total number of possible positions in all stored games.
    - It uses `numpy.random.choice()` to randomly select `batch_size` number of games from the buffer. The probability of selecting each game is proportional to the number of moves it has made, ensuring a uniform sampling across positions.
    - For each sampled game, it randomly chooses a position (index) within the game's history.
    - It constructs a list of tuples, where each tuple contains the game state (image) and the corresponding target values (value, policy). These tuples are generated using the `make_image()` and `make_target()` methods of the `Game` class.

Network Class

```
class Network(object):
  """
  A placeholder for the neural network used in AlphaZero.

  Methods:
    inference(image: List[numpy.array]) -> Tuple[float, List[float]]:
        Performs inference on the input image and returns the value and
policy.

    get_weights() -> List:
        Returns the weights of the neural network.
  """

  def inference(self, image: List[numpy.array]) -> Tuple[float,
List[float]]:
    """
    Performs inference on the input image and returns the value and
policy.

    Args:
      image (List[numpy.array]): The input image, a representation of
the game state.

    Returns:
      Tuple[float, List[float]]:
          A tuple containing the predicted value (expected outcome) and
```

```
policy (action probabilities).
    """
    return (-1, [])  # Placeholder for the actual implementation.

  def get_weights(self) -> List:
    """
    Returns the weights of the neural network.

    Returns:
      List: The weights of the neural network.
    """
    # Placeholder for the actual implementation.
    return []
```

- This class serves as a placeholder for the Network class that eventually represents the model used to learn the relationship between game states, values and policies. We already have a class that performs a somewhat similar function, the `AutoEncoder` class, that with minimal refactoring can accomplish this function. It remains to be seen if the network needs to be trained in Tensorflow, or if we can use Keras as a replacement.
- `inference(image)`: Performs a forward pass of the input image through the network. It should return a tuple containing the value associated to the state as predicted by the network, and a tuple of length num_actions that represents the probability distribution over the action space for said state. In a way, the network produces both the value of the state and the q-values of the state-action pairs. The actual implementation of the neural network is not provided in the pseudocode, so it returns a placeholder value of -1 for the predicted value and an empty list [] for the policy. In the actual implementation, this method would use the trained neural network to generate predictions.
- `get_weights()`: Returns the weights of the network. The actual implementation of obtaining weights from the neural network is not provided in the pseudocode, so it returns an empty list []. In practice, this method would retrieve the current weights of the neural network during training.

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`.