A Evaluation Hyperparameters

A.1 Hyperparameters for Evolution Strategies

Hyperparameter	Value
Noise standard deviation	0.02
Adam stepsize	0.01
L2 coefficient	0.005
Episodes per batch	5000

A.2 Hyperparameters for Proximal Policy Optimization

Hyperparameter	Value
Timesteps per batch	320000
SGD minibatch size	32768
SGD epochs per iteration	20
Adam stepsize	1e-4
PPO clip param	0.2
GAE parameter (λ)	0.95
Discount (γ)	0.995

B Case Study Pseudocode

B.1 AlphaGo Zero

```
class AlphaGoZero(Algorithm):
    """Outline of a top-level RLlib algorithm for AlphaGo Zero.

    This models the various distributed components in AlphaGo Zero as Ray
    actors. The train() call processes results from these components, routing
    data and launching new component actors as needed.

    Helper classes:
        AlphaGoEvaluator - Evaluator for self-play, holding the AlphaGo model.
        AlphaGoOptimizer - Optimizer wrapping SyncLocalOptimizer.
        SharedReplayBuffer - Evaluator backed by a replay buffer.

    Helper functions:
        elo_estimate - Estimates the strength of the given model.
        compare_models - Compares two models, returning the victory margin.
    """

    def __init__(self):
        self.best_model = AlphaGoEvaluator()
        self.replay_buffer = SharedReplayBuffer.remote()

        # AlphaGoOptimizers pull from the shared replay buffer to improve their
        # model. Note that AlphaGo Zero has many concurrent optimizers.
        self.optimizers = [
            AlphaGoOptimizer.remote(self.replay_buffer) for _ in range(20)]
        self.optimizer_tasks = set()
        weights = self.best_model.get_weights()
        # Initialize and kick off optimizer tasks
        for opt in self.optimizers:
            opt.set_weights.remote(weights)
```

```
            self.optimizer_tasks.append(opt.step.remote())

        # Self-play evaluators constantly evaluate the current best model and
        # add samples into the shared replay buffer.
        self.self_play_evaluators = [
            AlphaGoEvaluator.remote() for _ in range(20)]
        self.self_play_tasks = set()
        for ev in evaluators:
            self.self_play_tasks.append(ev.sample.remote())

        # When optimizers finish an optimization round, their model is compared
        # with the current best model to see if it can replace it. This set
        # tracks those concurrent comparison tasks.
        self.compare_tasks = set()

    def train(self):
        """Implements the train() method required for all RLlib algorithms."""

        result = elo_estimate.remote(self.best_model)
        start = time.time()

        # Return when enough time has passed and our new ELO estimation
        # finishes (the model will always be slightly ahead of the ELO).
        while time.time() - start < 60 or not ray.wait(result, timeout=0.0):
            self._step()

        return result

    def _step(self):
        """One step of the event loop that coordinates AlphaGo components."""

        # For simplicity, assume ray.wait() also returns the worker that ran
        # the task. As of Ray 0.2 you would need to track this in a dict.
        result_id, result_worker, _ = ray.wait(
            self.optimizer_tasks + self.self_play_tasks + self.compare_tasks)

        if result_id in self.optimizer_tasks:
            # Launch a compare task to see if we can beat the best model
            self.compare_tasks.append(
                compare_models.remote(best_model, result_id))
            # Continue optimization on this worker
            self.optimizer_tasks.append(result_worker.step.remote())
            self.optimizer_tasks.remove(result_id)

        if result_id in self.self_play_tasks:
            # Tell the replay buffer to incorporate the new rollouts
            self.replay_buffer.add_samples.remote(result_id)
            # Continue self play on this worker
            self.self_play_tasks.append(result_worker.sample.remote())
            self.self_play_tasks.remove(result_id)

        if result_id in self.compare_tasks:
            self.compare_tasks.remove(result_id)
            result = ray.get(result_id)
            # If it beats the current best model, broadcast new weights to
            # all evaluators and optimizers.
            if result.win_ratio >= 0.55:
                self.best_model = ray.get(result.model)
                weights = ray.put(self.best_model.get_weights())
```

```
                for ev in self.self_play_evaluators:
                    ev.set_weights.remote(weights)
                for opt in self.optimizers:
                    opt.set_weights.remote(weights)
                ray.kill(self.compare_tasks)
                self.compare_tasks = []


@ray.remote
class AlphaGoEvaluator(Evaluator):
    def sample(self):
        # return self-play rollouts based on current policy. Note that
        # self-play itself may be parallelized, launching further remote tasks.

    # standard update methods around the current policy
    def grad(self, samples): ...
    def weights(self): ....
    def set_weights(self, weights): ...


@ray.remote
class AlphaGoOptimizer(SyncLocalOptimizer):
    def __init__(self, replay_buffer):
        self.evaluator = AlphaGoEvaluator()
        SyncLocalOptimizer.__init__(self, self.evaluator, [replay_buffer])

    def set_weights(self, weights):
        self.evaluator.set_weights(weights)


@ray.remote
class SharedReplayBuffer(Evaluator):
    def add_samples(self, samples):
        # adds the given samples to the internal buffer

    def sample(self):
        # return batch of experience from internal buffer

    # don't need to implement these since AlphaGoOptimizer never calls them
    def grad(self, samples): pass
    def weights(self): pass
    def set_weights(self, weights): pass


@ray.remote
def compare_models(current_best, candidate):
    # compares the models, returning the candidate win ratio


@ray.remote
def elo_estimate(model):
    # returns an ELO estimate of the given model
```