nature portfolio

https://doi.org/10.1038/s41586-021-04357-7

Supplementary information

Outracing champion Gran Turismo drivers with deep reinforcement learning

In the format provided by the authors and unedited

This document is supplementary material to "Outracing champion Gran Turismo drivers with deep reinforcement learning."

Approximate Python Code

Here we provide approximate Python code that replicates the most critical aspects of our work. For the purposes of illustration, the code presented here is implemented in less computationally efficient ways than our actual implementation; has abstracted away many of the system implementation details such as remote process communication and check-pointing; and is written more directly than our actual implementation which supported wide configuration capabilities that otherwise detract from readability. Our implementation uses TensorFlow [1]; all references to 'tf.x' refer to an operation/class in the TensorFlow API, and references to 'keras.x' refers to an operation/class in the TensorFlow Keras API. A high-level overview of the training and rollout processes is provided in Figure 1.

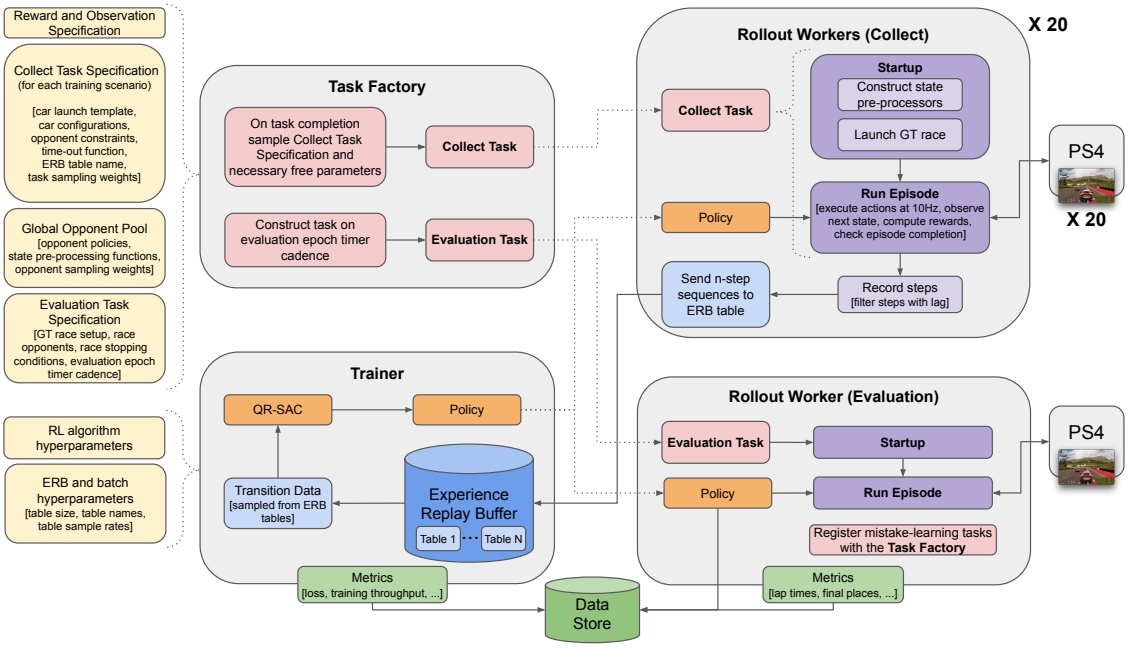

Figure 1: Diagram of modules in the GT Sophy training process

```
def train_loop():
    models = Models(
        policy=make_policy_network(),
        af1=make_qr_q_network(),
        af2=make_qr_q_network(),
        af1_target=make_qr_q_network(),
        af2_target=make_qr_q_network(),
    )
    # Sync prediction and target networks before start
    copy_weights(models.af1.trainable_params, models.af1_target.trainable_params)
    copy_weights(models.af2.trainable_params, models.af2_target.trainable_params)

    opts = Optimizers(
        policy=keras.optimizers.Adam(learning_rate=2.5e-5),
        critic=keras.optimizers.Adam(learning_rate=5.0e-5, global_clipnorm=10.0),
    )

    param_server = start_parameter_server(models.policy)
    replay_server = start_experience_replay_server()
    table_datasets = [
        make_dataset_for_table(replay_server, table_name)
        for table_name in TABLE_NAMES
    ]
    task_server = start_task_factory_server(warm_up=True)

    # Wait for MINIMUM_STEPS to be recorded to replay before we start any training
    replay_server.block_and_wait(MINIMUM_STEPS)

    while True:
        for _ in range(BATCHES_PER_EPOCH):
            transition_data = sample_data(table_datasets)
            # see QR-SAC code listing for implementation
            losses = step_qr_sac(models, opts, transition_data)
            write_metrics(losses)
        param_server.update_params(models.policy)
        # warmup over after first epoch
        task_server.warm_up = False

```

```
def sample_data(table_datasets: List[tf.data.Dataset]) -> Transition:
    # For simplicity, this code omits handling the edge case when only a subset of
    # the tables have data to sample.
    # Before concat, each dataset has a different batch size based on the table
    # weights
    sub_traj_batches = [next(iter(dataset)) for dataset in table_datasets]
    sub_traj_batches = nested_concat(sub_traj_batches, axis=0)
    # sub_traj_batches has fields obs, action, reward, done
    # Each tensor has two batch dimensions NT and 1 channel dimension, where N is
    # batch and T is time. The obs T dim is one longer than other T dims.

    discounting = tf.pow(DISCOUNT, tf.range(N_STEP, dtype=tf.float32))
    discounting = tf.reshape(discounting, (1, N_STEP, 1))
    discounted_reward = tf.math.reduce_sum(
        sub_traj_batches.reward * discounting,
        axis=1,
    )  # (N, 1)
    # Resulting transition removes T dimension, selecting relevant positions
    return Transition(
        obs=sub_traj_batches.obs[:, 0, :],  # first obs
        action=sub_traj_batches.action[:, 0, :],  # first action
        reward=discounted_reward,  # discounted cumulative reward
        next_obs=sub_traj_batches.obs[:, -1, :],  # last obs
        done=sub_traj_batches.done[:, -1, :],  # last done
    )
```

Listing 1: Training Loop Code

```
def step_qr_sac(m: Models, opts: Optimizers, t: Transition,) -> Losses:

    critic_params = m.af1.trainable_params + m.af2.trainable_params
    target_critic_params = (
        m.af1_target.trainable_params + m.af2_target.trainable_params
    )

    # these values do not need gradients traced through them
    policy_head_next = m.policy.policy_head(t.next_obs)
    actions_next = m.policy.sample_from_head(policy_head_next)
    log_prob_next = m.policy.log_prob(policy_head_next, actions_next)

    # update policy
    with tf.GradientTape() as tape:
        policy_head = m.policy.policy_head(t.obs)
        sampled_actions = m.policy.sample_from_head(policy_head)
        log_prob = m.policy.log_prob(policy_head, sampled_actions)
        policy_loss = policy_loss(
            log_prob_samples=log_prob,
            q1_sampled=m.af1.action_value(t.obs, sampled_actions),
            q2_sampled=m.af2.action_value(t.obs, sampled_actions),
        )
    opts.policy.minimize(policy_loss, m.policy.trainable_params, tape)

    # update critic networks
    with tf.GradientTape() as tape:
        critic_loss = critic_loss(
            q1_quantile_observed=m.af1.quantile(t.obs, t.action),
            q2_quantile_observed=m.af2.quantile(t.obs, t.action),
            q1_quantile_sampled_next=m.af1_target.quantile(t.next_obs, actions_next),
            q2_quantile_sampled_next=m.af2_target.quantile(t.next_obs, actions_next),
            log_prob_next=log_prob_next,
            reward=t.reward,
            done=t.done,
        )
    opts.critic.minimize(critic_loss, critic_params, tape)

    # move target network params toward prediction networks
    for pred_param, target_param in zip(critic_params, target_critic_params):
        target_param.assign(
            SMOOTH_FACTOR * pred_param + (1.0 - SMOOTH_FACTOR) * target_param
        )

    return Losses(policy_loss, critic_loss)

```

```python
def policy_loss(
    log_prob_samples: tf.Tensor,  # (N, 1)
    q1_sampled: tf.Tensor,  # (N, 1)
    q2_sampled: tf.Tensor,  # (N, 1)
) -> tf.Tensor:
    min_q = tf.minimum(q1_sampled, q2_sampled)
    return tf.math.reduce_mean(ALPHA * log_prob_samples - min_q)


def critic_loss(
    q1_quantile_observed: tf.Tensor,  # (N, M)
    q2_quantile_observed: tf.Tensor,  # (N, M)
    q1_quantile_sampled_next: tf.Tensor,  # (N, M)
    q2_quantile_sampled_next: tf.Tensor,  # (N, M)
    log_prob_next: tf.Tensor,  # (N, 1)
    reward: tf.Tensor,  # (N, 1)
    done: tf.Tensor,  # (N, 1)
) -> tf.Tensor:
    target = tf.stop_gradient(qr_sac_bootstrap_target(
        q1_quantile_sampled_next,
        q2_quantile_sampled_next,
        log_prob_next,
        reward,
        done,
        DISCOUNT**N_STEP,  # adjust for N-step transition
        ALPHA,
    ))

    q1_loss = tf.math.reduce_mean(quantile_loss(q1_quantile_observed, target))
    q2_loss = tf.math.reduce_mean(quantile_loss(q2_quantile_observed, target))
    return q1_loss + q2_loss


def quantile_loss(
    quantile: tf.Tensor,  # (N, M)
    quantile_target: tf.Tensor,  # (N, M)
) -> tf.Tensor:
    batch_size = quantile.shape[0]
    num_quantiles = quantile.shape[1]

    y = tf.reshape(quantile_target, [batch_size, -1, 1])  # (N, M, 1)
    x = tf.reshape(quantile, [batch_size, 1, -1])  # (N, 1, M)

    # compute huber loss
    diff = y - x  # (N, M, M)
    huber_loss = tf.where(
        tf.math.abs(diff) < 1.0, 0.5 * diff ** 2, tf.math.abs(diff) - 0.5
    )  # (N, M, M)

    # sample taus
    steps = tf.range(num_quantiles, dtype=tf.float32)  # (M,)
    taus = tf.reshape((steps + 1) / num_quantiles, [1, 1, -1])  # (1, 1, M)
    taus_minus = tf.reshape((steps / num_quantiles), [1, 1, -1])  # (1, 1, M)
    taus_hat = (taus + taus_minus) / 2.0  # (1, 1, M)

    # compute quantile loss
    delta = tf.cast(tf.stop_gradient(diff) < 0.0, dtype=tf.float32)  # (N, M, M)
    element_wise_loss = tf.math.abs(taus_hat - delta) * huber_loss  # (N, M, M)

    return tf.reduce_mean(tf.reduce_sum(element_wise_loss, axis=2), axis=1)  # scalar
```

```python
def qr_sac_bootstrap_target(
    q1_quantile_sample_next: tf.Tensor,  # (..., M)
    q2_quantile_sample_next: tf.Tensor,  # (..., M)
    next_log_prob_samples: tf.Tensor,  # (..., 1)
    reward: tf.Tensor,  # (..., 1)
    done: tf.Tensor,  # (..., 1)
    discount: float,  # scalar
    alpha: float,  # scalar
) -> tf.Tensor:
    quantile_next = minimum_quantile(
        q1_quantile_sample_next, q2_quantile_sample_next
    )
    entropy = alpha * next_log_prob_samples
    return reward + discount * (1.0 - done) * (quantile_next - entropy)


def minimum_quantile(
    quantile1: tf.Tensor,  # (N, M)
    quantile2: tf.Tensor,  # (N, M)
) -> tf.Tensor:
    batch_size = quantile1.shape[0]

    # compute Q value
    q1 = tf.reduce_mean(quantile1, axis=1, keepdims=True)  # (N, 1)
    q2 = tf.reduce_mean(quantile2, axis=1, keepdims=True)  # (N, 1)

    # pick quantiles based on minimum prediction
    stacked_q = tf.concat([q1, q2], axis=1)  # (N, 2)
    min_indices = tf.cast(tf.math.argmin(stacked_q, axis=1), dtype=tf.int32)  # (N,)
    indices = tf.stack([tf.range(batch_size), min_indices], axis=1)  # (N, 2)

    stacked_quantile = tf.stack([quantile1, quantile2], axis=1)  # (N, 2, M)
    minimum_quantile = tf.gather_nd(stacked_quantile, indices)  # (N, M)

    return minimum_quantile
```

Listing 2: QR-SAC Code

```python
def get_task() -> Task:
    # The returned task defines which cars are agent controlled and for each opponent
    # whether to control it with a pre-trained policy, PID, or the built-in AI
    task_server = get_task_server_info()
    if task_server.warm_up:
        # During warm up, run 20 cars across the track at once all selecting
        # uniformly random actions
        return Task(
            cars_launch=uniform_jittered_launch(num_cars=20, mph_range=(0, 60)),
            random_policy=True,
            agent_controlled_car_ids=list(range(20)),
            car_id_to_opp_policy={},
            car_id_to_opp_pid={},
            builtin_ai_car_ids=[],
            table_name="1v0",
            time_out_fn=lambda steps, obs: steps >= MAX_STEPS,
            is_eval=False,
            builtin_ai_params=None,
        )
    elif task_server.needs_eval:
        # Time for periodic evaluation
        task_server.needs_eval = False
        return Task(
            cars_launch=EVAL_LAUNCH,
            random_policy=False,
            agent_controlled_car_ids=[0],  # one car controlled by current policy
            car_id_to_opp_policy=EVAL_OPP_POLICIES,
            car_id_to_opp_pid={},
            builtin_ai_car_ids=EVAL_OPP_BUILTIN_IDS,
            table_name=None,  # Do not collect training data in evals
            time_out_fn=lambda steps, obs: steps >= MAX_STEPS,
            is_eval=True,
            builtin_ai_params=None,
        )
    elif (
        # limited slots for mistake learning to prevent regular tasks from starving
        len(task_server.mistake_learning_tasks) > 0
        and task_server.mistake_slots_available()
    ):
        return task_server.pop_mistake_learning_task()
    else:
        # Standard data collection setting
        # First we sample a task specification (e.g., 1v0, slipstream, etc.)
        # The TaskSpec is almost a Task, but the launch, opponent policies, and
        # builtin-ai params need to be sampled from a distribution
        task_spec = np.random.choice(TASK_SPECS, p=TASK_SPEC_PROBS)
        return Task(
            cars_launch=task_spec.sample_launch(),
            random_policy=False,
            agent_controlled_car_ids=task_spec.agent_ids,
            car_id_to_opp_policy=task_spec.sample_opp_policies(),
            car_id_to_opp_pid=task_spec.pids,
            builtin_ai_car_ids=task_spec.builtin_ids,
            table_name=task_spec.table_name,
            time_out_fn=task_spec.time_out_fn,
            is_eval=False,
            builtin_ai_params=task_spec.sample_builtin_ai_params(),
        )
```

Listing 3: The task factory is run in a separate process and is remotely queried by rollout workers.

```
def rollout_loop():
    policy = make_policy_network()
    param_server = get_parameter_server()
    ps_server = get_playstation_server()
    replay_server = get_experience_replay_server()
    task_server = get_task_server()
    while True:
        task = get_task()
        if task.random_policy:
            agent_policy = UniformRandomPolicy()
        else:
            load_weights(policy, param_server.get_params())  # Get latest policy
            agent_policy = policy

        ps_server.launch_race(  # Start the race on the PlayStation
            TRACK_NAME,  CAR_INFO, task.cars_launch, task.builtin_ai_car_ids
        )
        obs = ps_server.wait_for_race_start()
        p_obs = preprocess(obs)  # Preprocess creates an obs batch for all cars
        trajs = [
            start_trajectory(initial_obs=p_obs[agent_id])
            for agent_id in task.agent_car_ids()
        ]

        for step in range(MAX_STEPS):
            actions = compute_all_actions(
                task, agent_policy, p_obs, is_exploit=task.is_eval
            )
            ps_server.set_and_hold(actions)
            # Get the next observation 6 frames later = 0.1 seconds @60fps
            next_obs = ps_server.wait_for_obs_at_frame(obs.frame_count + 6)
            rewards = compute_reward(obs, next_obs)  # list of rewards for all cars
            dones = [False] * task.cars_launch.num_cars  # Continuing task

            obs = next_obs
            p_obs = preprocess(next_obs)

            for agent_id, traj in zip(task.agent_car_ids(), trajs):
                traj.append(
                    obs=p_obs[agent_id],
                    action=actions[agent_id],
                    reward=rewards[agent_id],
                    done=dones[agent_id],
                )
                if task.is_eval and should_retry(p_obs[agent_id]):
                    task_server.add_mistake_learning_task(traj)

            # create N-step subtrajectory if this task collects data
            if step + 1 >= N_STEP and task.table_name:
                for traj in trajs:
                    # this writes one additional observation time step
                    replay_server.write(task.table_name, traj[-N_STEP:])

            if task.time_out_fn(step, obs):
                break

        if task.is_eval:
            write_metrics(trajs)

```

```
def compute_all_actions(
    task: Task,
    agent_policy: Policy,
    p_obs: tf.Tensor,
    is_exploit: bool,
) -> List[Optional[tf.Tensor]]:
    # Actions for built-in AI cars remain None
    all_actions = [None for _ in range(task.cars_launch.num_cars)]
    # Compute actions for agent-controlled cars
    for car_id in task.agent_car_ids():
        if is_exploit:
            a = agent_policy.exploit_action(p_obs[car_id])
        else:
            a = agent_policy.sample_action(p_obs[car_id])
        all_actions[car_id] = a
    # Compute actions for opponent cars using previously trained policies
    for car_id, opp_policy in task.car_id_to_opp_policy.items():
        if is_exploit:
            a = opp_policy.exploit_action(p_obs[car_id])
        else:
            a = opp_policy.sample_action(p_obs[car_id])
        all_actions[car_id] = a
    # Compute actions for opponent cars using a PID controller
    for car_id, pid in task.car_id_to_opp_pid.items():
        # PID controllers always deterministically select actions
        a = pid.exploit_action(p_obs[car_id])
        all_actions[car_id] = a

    return all_actions
```

Listing 4: Rollout worker code. 21 instances, each with their own PlayStation connection, are run in parallel.

Hyperparameters

Here we provide a set of tables that cover the hyperparameters used for the training experiments. We focus on the parameters used to train GT Sophy for competitive racing.

RL Algorithm	
n-step	7
discount factor	0.9896
entropy regularization coefficient, α	0.01
number of quantiles	32
parameter smoothing factor	0.005
policy learning rate	2.5e-5
critic learning rate	5.0e-5
optimizer	Adam
global clip norm (critic optimizer only)	10.0

Table 1: Table of parameters for QR-SAC RL algorithm.

Neural Network	
number of hidden layers	4
number of units per hidden layer	2048
activation	relu
dropout (policy network only)	0.1

Table 2: Table of parameters for the neural network architecture.

Trainer	
mini-batch size	1024
total replay buffer capacity	10e+6
number of buffer tables	8 to 10
training steps per epoch	6000
minimum transitions to take before training	40000

Table 3: Table of parameters for the trainer.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.