Dopamine
A Research Framework for Deep Reinforcement Learning

Pablo Samuel Castro
@pcastr
señor swesearcher
Where I’m from
Core team

<table>
<thead>
<tr>
<th>Pablo Samuel Castro</th>
<th>Marc G. Bellemare</th>
<th>Carles Gelada</th>
<th>Subhodeep Moitra</th>
<th>Saurabh Kumar</th>
</tr>
</thead>
</table>

And many other contributors.
Dopamine is a research framework for fast prototyping of reinforcement learning algorithms.

It aims to fill the need for a small, easily grokked codebase in which users can freely experiment with wild ideas [speculative research].
Behavioural hypothesis → Run experiment → Conclude

“Double DQN leads to more conservative policies in fast-paced Atari games”
Behavioural hypothesis → Run experiment → Conclude

“Double DQN leads to more conservative policies in fast-paced Atari games”

Algorithmic variation → Implement

- Incorporate
- Discard

“Off-policy actor-critic”
“Double DQN leads to more conservative policies in fast-paced Atari games”

“Off-policy actor-critic”

“Distributional RL with autoregressive models”
Behavourial hypothesis → Run experiment → Conclude

“Double DQN leads to more conservative policies in fast-paced Atari games”

Algorithmic variation → Implement → Incorporate, Discard

“Off-policy actor-critic”

Dopamine

Idea → Develop → New algorithm, Discard

“Distributional RL with autoregressive models”
● Launched on August 27th to a very positive reception

In a new blog post, Google Brain team researchers @pcastr & @marcgbellemare share a new TensorFlow-based reinforcement learning framework that aims to provide flexibility, stability, and reproducibility for new and experienced RL researchers alike.

Introducing a New Framework for Flexible and Reproducible Reinforcement Learning

Posted by Pablo Samuel Castro, Research Software Developer and Marc G. Bellemare, Research Scientist, Google Brain Team

ai.googleblog.com
Launch stats
Launch stats

Google dopamine
12:08 PM - 21 May 2018

2,304 Retweets 11,249 Likes
Code Stats

- 12 python files (excluding tests)
  - A little over 2000 lines in total
- 98% code coverage
- Initial offering:
  - Atari environment (ALE)
  - 4 agents: DQN, C51, Rainbow, IQN
- Tensorboard integration
Code Design

- Colab
- Logger: logger.py (61)
- Checkpointer: checkpoint.py (90)
- ALE
- Runner: train.py (89), preprocessing.py (103), run_experiment.py (262), iteration_statistics.py (16)
- Replay Memory: circular_replay_buffer.py (448), prioritized_replay_buffer.py (156), sum_tree.py (82)
- Agent: dqn_agent.py (261), rainbow_agent.py (224), implicit_quantile_agent.py (221)
Code details

● Code is well-documented
● We ran 5 independent runs for all 4 agents on all 60 games
● We provide:
  ○ TensorFlow checkpoints for each of these runs
  ○ pickle files to easily visualize in colab (or anywhere)
  ○ Tensorboard event files
  ○ JSON files with data for plotting
● Colabs for extra documentation and instruction
DQN gin-config

DQNAgent.gamma = 0.99
DQNAgent.update_horizon = 1
DQNAgent.min_replay_history = 20000  # agent steps
DQNAgent.update_period = 4
DQNAgent.target_update_period = 8000  # agent steps
DQNAgent.epsilon_train = 0.01
DQNAgent.epsilon_eval = 0.001
DQNAgent.epsilon_decay_period = 250000  # agent steps
DQNAgent.tf_device = '/gpu:0'  # use '/cpu:*' for non-GPU version
DQNAgent.optimizer = @tf.train.RMSPropOptimizer()

tf.train.RMSPropOptimizer.learning_rate = 0.00025
tf.train.RMSPropOptimizer.decay = 0.95
tf.train.RMSPropOptimizer.momentum = 0.0
tf.train.RMSPropOptimizer.epsilon = 0.00001
tf.train.RMSPropOptimizer.centered = True

Runner.game_name = 'Pong'
Runner.sticky_actions = True
Runner.num_iterations = 200
Runner.training_steps = 250000  # agent steps
Runner.evaluation_steps = 125000  # agent steps
Runner.max_steps_per_episode = 27000  # agent steps

WrappedReplayBuffer.replay_capacity = 1000000
WrappedReplayBuffer.batch_size = 32
Comparison with published settings

**SPACEINVADERS**
- **Published settings**
- **Default settings**

- **SEAQUEST**
- **Published settings**
- **Default settings**
Episode ends: LifeLoss vs GameOver

Train vs Eval curves

**SEAQUEST**
- DQN (train)
- DQN (eval)
- C51 (train)
- C51 (eval)

**BREAKOUT**

**ASTERIX**
Sticky vs non-sticky actions

Runner.sticky_actions = False
New agent based on DQN

```python
# @title Create an agent based on DQN, but choosing actions randomly.

LOG_PATH = os.path.join(BASE_PATH, 'random_dqn', GAME)

class MyRandomDQNAgent(dqn_agent.DQNAgent):
    def __init__(self, sess, num_actions):
        """This maintains all the DQN default argument values."""
        super(MyRandomDQNAgent, self).__init__(sess, num_actions)

    def step(self, reward, observation):
        """Calls the step function of the parent class, but returns a random action."
        return np.random.randint(self.num_actions)

def create_random_dqn_agent(sess, environment, summary_writer=None):
    """The Runner class will expect a function of this type to create an agent."""
    return MyRandomDQNAgent(sess, num_actions=environment.action_space.n)

# Create the runner class with this agent. We use very small numbers of steps
# to terminate quickly, as this is mostly meant for demonstrating how one can
# use the framework. We also explicitly terminate after 110 iterations (instead
# of the standard 200) to demonstrate the plotting of partial runs.
random_dqn_runner = run_experiment.Runner(LOG_PATH,
                                          create_random_dqn_agent,
                                          game_name=GAME,
                                          num_iterations=200,
                                          training_steps=10,
                                          evaluation_steps=10,
                                          max_steps_per_episode=100)
```
New agent from scratch

class StickyAgent(object):
    """This agent randomly selects an action and sticks to it. It will change
    actions with probability switch_prob."""
    def __init__(self, sess, num_actions, switch_prob=0.1):
        self._sess = sess
        self._num_actions = num_actions
        self._switch_prob = switch_prob
        self._last_action = np.random.randint(num_actions)
        self.eval_mode = False
    def _choose_action(self):
        if np.random.random() <= self._switch_prob:
            self._last_action = np.random.randint(self._num_actions)
        return self._last_action
    def bundle_and_checkpoint(self, unused_checkpoint_dir, unused_iteration):
        pass
    def unbundle(self, unused_checkpoint_dir, unused_checkpoint_version,
                 unused_data):
        pass
    def begin_episode(self, unused_observation):
        return self._choose_action()
    def end_episode(self, unused_reward):
        pass
    def step(self, reward, observation):
        return self._choose_action()
    def create_sticky_agent(self, sess, environment, summary_writer=None):
        """The Runner class will expect a function of this type to create an agent."""
        return StickyAgent(sess, num_actions=environment.action_space.n,
                           switch_prob=0.2)
sticky_runner = run_experiment.Runner(LOG_PATH,
create_sticky_agent,
game_name=GAME,
num_iterations=200,
training_steps=10,
evaluation_steps=10,
max_steps_per_episode=100)
Baselines plots
Common requests

- Non-Atari gym environments (Cartpole, Acrobot, etc.)
  - very soon!
- Policy gradient methods
- Distributed training
- Visualizations
¡Gracias!

github.com/google/dopamine

Pablo Samuel Castro
@pcastr