q2 Documentation

Release 0.1.2

Tom Alcorn

Jul 02, 2018

Contents

q2 is a reinforcement learning framework and command line tool. It consists of a set of interfaces that help organize research and development and a set of command line tools that unify your workflow. It also comes "batteries-included" with a set of helpful components and utilities that are frequently used in reinforcement learning work.

q2 aims to improve your reinforcement learning work in the following ways:

- Reduce boilerplate, no more copy-pasted code, avoid reinventing the wheel
- Easier collaboration if your colleagues are familiar with q2
- Faster iteration time: try out a new reinforcement learning idea in only three steps

q2 is completely open-source and under active development, so if you find something missing, please reach out on Github!

CHAPTER 1

Get started

1.1 Installation

You will need to be using Python 3.5 or greater (this is because q2 makes extensive use of Python's new type annotations).

Then simply run:

```
pip install q2
```

1.2 First time usage

To support your work, q2 needs a certain project structure. For a new project, you can generate this structure by running the following command from within the project directory:

```
q2 init
```

This will create four folders and one file:

- `agents/`
- `environments/`
- `objectives/`
- `regimens/`
- `objects.yaml`

You can then run:

```
q2 generate --help
```

to see how to go about generating new agents, environments, objectives, regimens and more.

To see q2 in action right away, start a training session with q2's built-in `random` agent in OpenAI Gym's `Cartpole-v1` environment by running:

```
q2 train random --env gym.CartPole-v1 --episodes 10 --render
```

You should see a window open up rendering the `CartPole-v1` environment with `random` agent playing. The basic syntax of the `train` command is:

```
q2 train <agent> --env <environment> --episodes <num_episodes> --render
```

Run `q2 train --help` to see all the options.

CHAPTER 2

Contents

2.1 Tutorial

In this tutorial you will use q2 to implement a Deep-Q Network that can solve the cart-pole problem. The goal is to familiarize you with q2, show you how it can speed up and simplify development and maybe even learn a bit about reinforcement learning to boot.

In the cart-pole problem, the agent must balance an unstable vertical pole on top of a cart that can roll horizontally. It's a classic problem in dynamics and fortunately, OpenAI gym offers an implementation of the problem in the CartPole-v1 environment, which you can access through q2. You can think of the cart-pole problem as a video game with two buttons, left and right. Your goal is to keep the pole from falling over for as long as possible using only those two buttons.

At each time step, the environment yields an observation that corresponds to the position and velocity of the cart, and the angular position and angular velocity of the pole. The agent then selects either left or right, which adds a constant increment to the cart's velocity in the left and right direction respectively. Every time step that it survives, the agent gets +1 reward.

Now you understand the problem: let's build an agent that solves it!

2.1.1 Setup

First up, there's some basic setup to do. You'll need Python 3.5 or greater. You can check what version you have at the command line with:

```
python3 --version
```

Next, make a new folder called `q2_tutorial` for this project:

```
mkdir q2_tutorial
cd q2_tutorial
```

Then make a virtual environment to keep this project's dependencies isolated from the rest of your system:

```
python3 -m venv env
. ./env/bin/activate
```

And finally, install q2:

```
pip install q2
```

q2 offers a command line interface that automates mundane tasks so you can spend more time on the problem. First, you can have q2 automatically setup your project structure:

```
q2 init
```

Then, test that q2 is working properly by starting a training session on the cart-pole problem with the built-in random agent:

```
q2 train random --env gym.CartPole-v1 --episodes 10 --render
```

You should see the cart-pole environment rendered on the screen, with the random agent controlling the cart. As you might guess from the name, the random agent just chooses a random action every time. In this case, that would be a random string of lefts (`0`) and rights (`1`). This turns out not to be a very good strategy. It's time to build our own agent, to see if we can do better.

Fig. 1: The `random` agent playing `gym.CartPole-v1`.

2.1.2 Generate

q2 can generate a new agent from a template for you. This helps you avoid rewriting boilerplate and gets you started with a working agent based on the random agent. We're going to build a Deep-Q Network, or DQN for short, so use q2 to generate an agent called `dqn`:

```
q2 generate agent dqn
```

Now you should have a file at `q2_tutorial/agents/dqn.py`. Open it up in an editor. You should see a Python class definition that begins:

```
class Dqn(Agent):
    ...
```

There are six methods defined: `__init__`, `load`, `save`, `act`, `step`, and `learn`. Here's a run-down of what each method is for:

`__init__` This is where you will define the neural network and setup other objects that the agent needs.

`act(state, training_flag) -> action` This method is called whenever an action is needed. It takes an environment state (e.g. cart-pole position and velocity) and a boolean flag that tells the agent whether this is training or testing. It should return a valid action.

`step(state, action, reward, next_state, episode_end)` This method is the agent's chance to do some work between steps of the environment. It is called once every time step and can be used to do things like run the learning step or append to internal buffers.

`learn(states, actions, rewards, next_states, episode_ends) -> loss` Runs the learning step using the training data provided. This is where you will run a gradient descent step.

`load`, `save` These methods are responsible for saving and loading the Tensorflow checkpoint file, so that your agent remembers what it has learned between training sessions. You will not need to edit them during this tutorial.

2.1.3 Deep-Q Networks

Before we start implementing, here's a primer on Deep-Q Networks. A Deep-Q Network is fundamentally just a neural network that learns to predict how much reward it will get if it takes action `A` when the environment is in state `S`. Given a good approximation to this reward function (which is usually denoted `Q(S, A)`, hence "Q" in DQN), it's easy to implement a good agent: just choose the action that is predicted to produce the most reward!

So at each time step the agent does something and gets a certain amount of reward. The DQN looks at the state that the environment was in, the action that was taken and makes a prediction for what the reward will be. It then compares its prediction with the actual reward that was given and updates itself based on its error. When it comes time to choose the next action, our agent simply runs the DQN and chooses the action that is predicted to produce the most reward. That's it.

Well, there is one small complication: the reward that we care about maximizing is actually the *total* future reward, not just the reward on the next time step. For example, we want our agent to learn how to wait in order to take 10 marshmallows tomorrow rather than 1 today. To make that happen, the DQN is actually going to be learning to predict the total future reward of an action.

After each time step, the agent will use the DQN twice: once to make a prediction about the state that it just acted on, and once to predict the total future reward based on the *next* environment state that was just entered. Then, we add the actual reward from the current time step to the total future reward predicted for the upcoming state to get our target "ground-truth" total future reward. This is then fed to the DQN to compare with its prediction and update.[1]

2.1.4 Build

You're now ready to define a model. First you'll create a two-layer neural network in `__init__` using Tensorflow:

```
def __init__(self, action_space, observation_space):
    if not isinstance(action_space, Discrete):
        raise TypeError("Invalid environment, this agent only works" +
            "with Discrete action spaces.")

    self.action_space = action_space
    self.name = type(self).__name__ + "Agent"
    self.checkpoint_name = 'checkpoints/' + self.name + '.ckpt'

    # The training regimen pulls messages from the agent to be displayed
    # during training.
    self.message = ""

    # It's a good idea to keep track of training loss
    self.losses = list()

    # Model parameters
    hidden_nodes = 128
    learning_rate = 1e-4

    # Model definition
    with tf.variable_scope(self.name):
        # Input placeholders
        self.state = tf.placeholder(tf.float32,
            [None, *observation_space.shape], name='state')
        self.target = tf.placeholder(tf.float32, [None], name='target')
        self.action = tf.placeholder(tf.int32,
            [None, *action_space.shape], name='action')
```

(continues on next page)

[1] In reinforcement learning this idea is known as the Bellman equation.

(continued from previous page)

```
        # Transformed inputs
        self.action_vector = tf.one_hot(self.action, action_space.n)
        self.state_flat = tf.layers.flatten(self.state)

        # Hidden layers
        self.hidden0 = tf.contrib.layers.fully_connected(self.state_flat,
            hidden_nodes)
        self.hidden1 = tf.contrib.layers.fully_connected(self.hidden0,
            hidden_nodes)

        # Outputs
        self.value = tf.contrib.layers.fully_connected(self.hidden1,
            action_space.n, activation_fn=None)
        self.predicted_reward = tf.reduce_sum(tf.multiply(self.value,
            self.action_vector), axis=1)

        # Learning
        self.loss = tf.reduce_mean(tf.square(self.target -
            self.predicted_reward))
        self.opt = tf.train.AdamOptimizer(
            learning_rate=learning_rate).minimize(self.loss)
```

So far this is a fairly standard model definition in Tensorflow. You've defined a computational graph that will be run later during `act` and `learn` to produce a prediction of the total future reward to be had for each possible action. Next, implementing `act` is straightforward. You just compute the value for each action and then choose the best one:

```
def act(self,
    sess:tf.Session,
    state:np.array,
    train:bool,
    ) -> np.array:
    # self.value holds the predicted rewards for each action
    value = sess.run(self.value, feed_dict={
        self.state: state.reshape((1, *state.shape))
    })
    best_action = np.argmax(value)
    return best_action
```

`learn` is where you compute the total future reward based on the new state of the environment, and then feed that to the DQN as the target towards which to optimise:

```
def learn(self,
    sess:tf.Session,
    states:np.array,
    actions:np.array,
    rewards:np.array,
    next_states:np.array,
    episode_ends:np.array
    ) -> float:
    # Discount factor
    gamma = 0.99
    # Compute ground-truth total future expected value based on actual
    # rewards using the Bellman equation
    future_values = sess.run(self.value, feed_dict={
        self.state: next_states,
```

(continues on next page)

(continued from previous page)

```
    })
    # Expected future value is 0 if episode has ended
    future_values[episode_ends] = np.zeros(future_values.shape[1:])
    # The Bellman equation
    targets = rewards + gamma * np.max(future_values, axis=1)

    loss, _ = sess.run([self.loss, self.opt], feed_dict={
        self.state: states,
        self.target: targets,
        self.action: actions,
    })
    return loss
```

Finally, `step` is where you run the learning step. For now there is nothing else that needs to be done here:

```
def step(self,
    sess:tf.Session,
    state:np.array,
    action:np.array,
    reward:float,
    next_state:np.array,
    done:bool
    ):
    # For this simple agent, all we need to do here is run the
    # learning step.
    loss = self.learn(sess, [state], [action], [reward], [next_state],
        [done])
    self.losses.append(loss)

    self.message = "Loss: {:.2f}".format(loss)
```

You now have a fully functioning DQN agent! Try it out against the cart-pole environment in a training session:

```
q2 train dqn --env gym.CartPole-v1 --episodes 10 --render
```

Once again you should see the cart-pole environment rendered on the screen, only this time your `Dqn` agent is playing.

2.1.5 Extend

With the basic implementation from above, you probably observed that the agent always goes to one-side as quickly as it can. This is a very common failure mode for RL agents. In our case, the initial weights of the DQN came out slightly favouring either left or right. Consequently, the agent chose that action, then receiving a reward of +1 for surviving that time step. This causes the DQN to increase its confidence in that action, leading to a runaway self-reinforcing process in which it will only ever output the same action.

2.1.6 Exploration

One way to remedy this is to break the loop by injecting some randomness into the agent's actions. q2 comes with some useful tools for this out of the box. At the top of the file, import a decaying noise generator like so:

```
from q2.agents.noise import DecayProcess
```

`DecayProcess` generates a stream of `1`s and `0`s, with `1`s showing up less and less frequently as the process goes on. We can use this to add some randomness to our agents behaviour that starts out big and slowly disappears, letting the agent have more control. Go back down to `__init__` and add a line to instantiate the `DecayProcess`

```
def __init__(...):
    ...
    # Agents need to trade off between exploring and exploiting. This decay
    # process starts the agent off with a high initial exploration tendency
    # and gradually reduces it over time.
    self.noise = DecayProcess(
        explore_start=1.0, explore_stop=0.1, final_frame=1e4)
    ...
```

We'll make use of this when choosing the next action. Add these lines to the start of the definition of `act`:

```
def act(...):
    # Decide whether to "explore" i.e. take a completely random action
    if self.noise.sample() == 1 and train:
        return self.action_space.sample()
    ...
```

Finally, in order for the process to decay it needs to be stepped every time that the agent is stepped. Modify the end of `step` like so:

```
def step(...):
    ...
    self.noise.step()

    self.message = "Loss: {:.2f}\tExplore: {:.2f}".format(
        loss, self.noise.epsilon)
```

Now run a training session with your agent again! You should observe it mixing up its actions much more often.

2.1.7 Replay buffer

At this point, if you just left the agent running for a few thousand episodes it would solve this environment. However, at the moment the agent is learning very inefficiently. At each time step it looks at what just happened and tries to learn from it. This means that the variance in the gradient will be high, and the network will take a winding, inefficient path down the objective landscape. Additionally, the fact that the network is learning from events in the order that they happened means that it is vulnerable to loops in the learning process that might prevent it from converging.

We can fix this by adding one last component to the agent: a replay buffer. The agent will record each step of the environment to a buffer, and at each step it will sample from this buffer to get training data for the learning step. This breaks potential feedback loops because learning can happen out of order. It also reduces variance in the gradient step by averaging over multiple data points. Once again, q2 comes with a helper to make implementing this easy. At the top of the file, add:

```
from q2.agents.history import History
```

Then in `__init__`, add this line:

```
def __init__(...):
    ...
    # In the learning step, we will sample from a history of
    # the last 1000 training steps seen.
```

(continues on next page)

(continued from previous page)

```
    self.history = History(1000)
    ...
```

And add these lines to the start of `learn`:

```
def learn(...):
    # Add the current step to the history buffer
    self.history.step(state, action, reward, next_state, done)

    # Sample history for learning
    batch_size = 10
    states, actions, rewards, next_states, dones = self.history.sample(
        batch_size)
```

Finally, modify the learning step to use the batch of data:

```
loss = self.learn(sess, states, actions, rewards, next_states, dones)
```

That's all! Run the agent again and observe how much faster the loss drops. Finally, try running the training for 500 episodes like so:

```
q2 train dqn --env gym.CartPole-v1 --epochs 5 --episodes 100
```

Once it's done, you can run a test session in which the agent doesn't explore at all:

```
q2 train dqn --env gym.CartPole-v1 --episodes --test --render
```

If all went well, the agent should be noticeably better at cart-pole than when it started. Try running the random agent again to compare.

That's the end of this tutorial. Hopefully you see how q2 makes developing RL agents easier and faster. For some next steps, try modifying this agent to learn other environments. Or try messing with the model parameters and architecture to see if you can get it to solve cart-pole faster: OpenAI Gym defines solving cart-pole as consistently achieving an episode score above 195.

2.1.8 Source code

The complete source code for the agent you developed is available below for reference:

```
import tensorflow as tf
import numpy as np
from gym import Space
from gym.spaces import Discrete, Box, MultiBinary
from q2.agents import Agent
from q2.agents.noise import DecayProcess
from q2.agents.history import History

class Dqn(Agent):
    def __init__(self, observation_space:Space, action_space:Space):
        if not isinstance(action_space, Discrete):
            raise TypeError("Invalid environment, this agent only works with Discrete␣
↪action spaces.")

        self.action_space = action_space
        self.name = type(self).__name__ + "Agent"
```

(continues on next page)

(continued from previous page)

```
        self.checkpoint_name = 'checkpoints/' + self.name + '.ckpt'

        # The training regimen pulls messages from the agent to be displayed during␣
↪training
        self.message = ""

        # It's a good idea to keep track of training loss
        self.losses = list()

        # Model parameters
        hidden_nodes = 128
        learning_rate = 1e-4

        # In the learning step, we will sample from a history of training steps seen.
        self.history = History(1000)

        # Agents need to trade off between exploring and exploiting. This decay␣
↪process starts
        # the agent off with a high initial exploration tendency and gradually␣
↪reduces it over
        # time.
        self.noise = DecayProcess(explore_start=1.0, explore_stop=0.1, final_
↪frame=1e4)

        # Model definition
        with tf.variable_scope(self.name):
            # Input placeholders
            self.state = tf.placeholder(tf.float32, [None, *observation_space.shape],␣
↪name='state')
            self.target = tf.placeholder(tf.float32, [None], name='target')
            self.action = tf.placeholder(tf.int32, [None, *action_space.shape], name=
↪'action')

            # Transformed inputs
            self.action_vector = tf.one_hot(self.action, action_space.n)
            self.state_flat = tf.layers.flatten(self.state)

            # Hidden layers
            self.hidden0 = tf.contrib.layers.fully_connected(self.state_flat, hidden_
↪nodes)
            self.hidden1 = tf.contrib.layers.fully_connected(self.hidden0, hidden_
↪nodes)

            # Outputs
            self.value = tf.contrib.layers.fully_connected(self.hidden1, action_space.
↪n,
                                                           activation_fn=None)
            self.predicted_reward = tf.reduce_sum(tf.multiply(self.value, self.action_
↪vector), axis=1)

            # Learning
            self.loss = tf.reduce_mean(tf.square(self.target - self.predicted_reward))
            self.opt = tf.train.AdamOptimizer(learning_rate=learning_rate).
↪minimize(self.loss)

    def load(self, sess:tf.Session):
        train_vars = tf.trainable_variables(scope=self.name)
```

(continues on next page)

(continued from previous page)

```
        saver = tf.train.Saver(train_vars)
        try:
            saver.restore(sess, self.checkpoint_name)
            print("Checkpoint loaded")
        except (tf.errors.InvalidArgumentError, tf.errors.NotFoundError):
            print("Checkpoint file not found, skipping load")

    def save(self, sess:tf.Session):
        train_vars = tf.trainable_variables(scope=self.name)
        saver = tf.train.Saver(train_vars)
        saver.save(sess, self.checkpoint_name)

    def act(self,
        sess:tf.Session,
        state:np.array,
        train:bool,
        ) -> np.array:
        # Decide whether to "explore" i.e. take a completely random action
        if self.noise.sample() == 1 and train:
            return self.action_space.sample()
        value = sess.run(self.value, feed_dict={
            self.state: state.reshape((1, *state.shape))
        })
        best_action = np.argmax(value)
        return best_action


    def step(self,
        sess:tf.Session,
        state:np.array,
        action:np.array,
        reward:float,
        next_state:np.array,
        done:bool
        ):
        # Add the current step to the history buffer
        self.history.step(state, action, reward, next_state, done)

        # Sample history for learning
        batch_size = 10
        states, actions, rewards, next_states, dones = self.history.sample(batch_size)

        # For this simple agent, all we need to do here is run the learning step
        # loss = self.learn(sess, [state], [action], [reward], [next_state], [done])
        loss = self.learn(sess, states, actions, rewards, next_states, dones)
        self.losses.append(loss)

        self.noise.step()

        self.message = "Loss: {:.2f}\tExplore: {:.2f}".format(loss, self.noise.
↪epsilon)

    def learn(self,
        sess:tf.Session,
        states:np.array,
        actions:np.array,
        rewards:np.array,
```

(continues on next page)

(continued from previous page)

```
        next_states:np.array,
        episode_ends:np.array
        ) -> float:
        # Discount factor
        gamma = 0.99
        # Compute ground-truth total future expected value based on actual rewards␣
↪using the Bellman equation
        future_values = sess.run(self.value, feed_dict={
            self.state: next_states,
        })
        # Expected future value is 0 if episode has ended
        future_values[episode_ends] = np.zeros(future_values.shape[1:])
        # The Bellman equation
        targets = rewards + gamma * np.max(future_values, axis=1)

        loss, _ = sess.run([self.loss, self.opt], feed_dict={
            self.state: states,
            self.target: targets,
            self.action: actions,
        })
        return loss
```

2.2 Command line tool

q2 offers three basic commands:

- `init`
- `generate`
- `train`

You can run:

```
q2 --help
```

to see usage on the command line.

2.2.1 Commands

`init` Initialize a new project

A one-time command that you should run at the start of a new project. It will set up the necessary directory structure for you (if some or all of this structure already exists, it won't overwrite it).

`generate` Create new objects easily

The basic syntax is:

```
q2 generate <type> <name>
```

where `<type>` can be `agent`, `environment`, `objective` or `regimen`. When you run the command, an appropriate template is pulled up and rendered with the name you specified, then written to the appropriate folder within your q2 project. You can then edit the newly generated object by opening up the generated file.

train Run a training session

Begins a training session in which your agent will interact with an environment and learn interesting new behaviours. The basic syntax is:

```
q2 train <agent> --env <environment> --regimen <regimen> \
--episodes <num_episodes> [--render]
```

First some setup happens, then the specified regimen is instantiated and control is handed over to it. All regimens perform at least the basic process of successively stepping the agent and the environment and logging basic information, but a lot more can also be happening. See the `Regimen` section for more details (TODO link). When the training is done, certain outputs may have been generated including training loss data and Tensorflow checkpoint files.

2.3 Project structure

To support your work, q2 expects a certain project structure, consisting of four folders and one file:

- `agents/`
- `environments/`
- `objectives/`
- `regimens/`
- `objects.yaml`

q2 will setup this structure for you when you run the command:

```
q2 init
```

from within your project directory.

Each of the directories contains your agents, environments, objectives and regimens respectively. q2 uses `objects.yaml` to keep track of your stuff so that the command line tool knows where to look for it. It is a YAML file that contains a reference to each user-defined object with some supporting information and metadata. You shouldn't need to modify it directly, but it does need to be checked into source control.

2.4 Agents

Agents generate actions in response to states of the environment. Usually they also learn certain behaviours in response to a reward signal. Agents are the primary object of interest to reinforcement learning researchers, and you'll probably spend most of your time working on them. In q2, an agent is any Python class that inherits the q2.Agent abstract base class[1], which simply means that it has to implement a certain set of methods:

class `q2.agents.`**Agent**

An abstract base class (interface) that specifies what an agent must implement. You will fill in your own definitions for each of these methods.

__init__(*observation_space*, *action_space*)

This is where you define the model and setup any other objects that the agent needs.

[1] See q2/agents/agent.py for the full abstract base class.

act (*state*, *training_flag*) → action
: This method is called whenever an action is needed. It takes an environment state (e.g. cart-pole position and velocity) and a boolean flag that tells the agent whether this is training or testing. It should return a valid action.

  Note: this method should be as close to a pure function as possible, always returning the agents best guess for what to do next given the state. Side-effects like updating internal state should be handled by `step`.

step (*state*, *action*, *reward*, *next_state*, *episode_end*)
: This method is the agent's chance to do some work between steps of the environment. It is called once every time step and should be used to do things like run the learning step, or update an internal buffer of environment observations.

learn (*states*, *actions*, *rewards*, *next_states*, *episode_ends*) → loss
: This should run a learning step to update the agent's model using the batch of training data passed as argumnets. It will typically be called from within `step`, but q2 requires you to implement it separately so that training regimens can control when the agent learns.

load (*)*, *save(*)
: These two methods are responsible for loading and saving anything the agent needs to keep around between training sessions e.g. TensorFlow checkpoint files.

q2 will call these methods for you during training sessions. For example, in the default training regimen (`online`), `act` is called when the environment is ready for the action to supply it's next action, `step` is called after the environment has updated to its next state, and `load` and `save` are called at the beginning and end of the training session respectively[2].

When you generate a new agent with `q2 generate agent <agent_name>`, these methods are stubbed out for you with an implementation drawn from the `random` agent (which simply chooses a random action at each time step). This is so that you can see a basic implementation and hack on it instead of starting from scratch.

2.4.1 Implementation checklist

Here is the basic implementation checklist for creating an agent that uses deep learning:

1. Run `q2 generate agent my_agent` to generate a new agent from template.
2. In `__init__`, define your network. From here on I will assume you named the tensor that represents the agents output actions as `self.action`, and the tensor that represents the agents learning step as `self.optimize`.
3. In `act`, call `action = sess.run(self.action, feed_dict={...})` with the appropriate inputs and return the action at the end.
4. In `learn`, call `loss = sess.run([self.loss, self.optimize], feed_dict={...})` with the appropriate inputs and return the loss at the end.
5. In `step`, handle the new environment state and call `self.learn(...)` somewhere.
6. The default implementations of `save` and `load` are good enough for most purposes, I recommend that you leave them as they are until there is a reason to change them.

2.5 Environments

Environments evolve over time as the agent interacts with it, typically according to predetermined rules so that the environment presents consistent behaviour. They may also define end conditions that cause the episode to end.

q2 comes with a set of environments from OpenAI Gym and OpenAI Retro. On the command line, run:

[2] `learn` is only explicitly called by the regimen during `offline` training.

```
q2 train random --env gym.CartPole-v1 --episodes 10 --render
```

to see OpenAI Gym's cart-pole environment in action.

Any OpenAI gym environment can be referenced from the command line tool as `gym.<environment>`, and similarly any OpenAI Retro can be specified as `retro.<game>`. Note that some of the Retro environments require special ROMs[1].

However, eventually you may want to design your own training environment. To do so, you need to make a Python class that inherits from the abstract base class `q2.environments.Environment`, which means that you will implement the following interface:

class `q2.environments.`**Environment**

An abstract base class (interface) that specifies what the environment must implement. You will fill in your own definitions for each of these methods.

reset() → state

Reset the environment to its initial state and return it.

step(*action*) → next_state, reward, done, info

Given the agent's choice of action, move the environment forward one time step and return a 4-tuple that includes the new state, the reward as a `float`, whether the environment has entered an end state as a `bool`, and an arbitrary info `dict` which may be used to output debugging information.

The agents action must match the spec in `action_space` and the new state returned must match the spec in `observation_space`.

render()

Optional to implement, ignore it if you don't care about seeing the environment on the screen. If you do want to implement it, a good practice is to create a new window the first time `render` is called, drawing to it on every subsequent call and then closing the window from within `close()`.

close()

Shutdown the environment, freeing any resources (e.g. rendering windows).

observation_space, action_space

These attributes must be set to instances of `gym.Space`. They provide a specification of the allowable actions and environment states, which agents may use to define their internal models.

All states returned by the environment must match the spec in `observation_space`.

2.6 Objectives

Objectives are pipes that take in the reward signal generated by the environment and transform it into an objective for the agent to optimize toward. This enables more complex training behaviour, but is not always necessary. q2 comes with a `Passthrough` objective that simply passes the reward that the environment yielded directly to the agent. `Passthrough` is used by default unless you override it with your own.

Run:

```
q2 generate objective my_objective
```

to start working on your own objective. You need to implement the following interface:

class `q2.objectives.`**Objective**

An abstract base class (interface) that specifies what the environment must implement. You will fill in your own definitions for each of these methods.

[1] See the https://github.com/openai/retro#roms for details.

reset()
This is called by the training regimen before each episode. It gives you the opportunity to wipe any accumulated state.

step(*state*, *action*, *reward*) → float
Accepts a `state` and `reward` from the environment, an `action` from the agent and returns the new objective for the agent to learn.

2.7 Regimens

Regimens are training algorithms. At its core, a training algorithm is responsible for setting up the environment and the agent and successively calling `step` on each one, while passing actions and states back and forth as appropriate. q2's `Regimen` class implements this basic funtionality and provides hooks for you to customize any other behaviour as desired.

Run:

```
q2 generate regimen my_regimen
```

to generate a new regimen from template, then fill in the implementations of whichever event hooks you need.

class q2.regimens.**Regimen**

before_training()

after_training()
Called once at the start and end of training respectively.

before_epoch(*epoch:int*)

after_epoch(*epoch:int*)
Called before and after each epoch.

before_episode(*episode:int*)

after_episode(*episode:int*)
Called before and after each episode.

before_step(*step*)

after_step(*step*)
Called before and after each step of the environment and agent.

on_error(*step*, *exception*)
Called when an exception occurs. If this method returns `True` then propagation of the exception is stopped, which can be useful when certain exceptions are expected to occur.

plugins() → List[Plugin]
A list of plugins to be used by your regimen.

log(*msg:str*)
Add a message to the logging output for the current timestep. This method is implemented by q2 and provided as a convenience, you should not override it with your own implementation.

agent: Agent

env: Environment

sess: tf.Session
The tensorflow session.

objective: Objective

action_space
The action_space of the environment.

observation_space
The observation_space of the environment.

agent_constructor: Type[Agent]
Callable that constructs a new agent.

env_maker: Maker
Callable that creates a new environment.

2.7.1 Plugins

When implementing your own regimens, you might find that you want to re-use the same morsels of useful behaviour in multiple different regimens. You can achieve this by implementing a `Plugin`. For example, q2 comes with a *DisplayFramerate* plugin that lets any regimen display a nice framerate message in the logs without polluting the core logic of the regimen.

The interface of a plugin is identical to that of a `Regimen` except that each method takes as first argument a `Regimen` which it can inspect, interact with or modify. Note that the plugin event hooks are always called *before* the regimen's hooks, so the regimen always has final say over any state before the next step is run. You should write your plugins to account for the possibility that the regimen itself might change some state before the next event happens.

You can use a plugin by calling adding it to the list returned by `regimen.plugins()`.

- genindex
- modindex
- search

CHAPTER 3

Project goals

Reinforcement learning research is surprisingly hard to reproduce, especially if you want to use a new technique within a larger system. It is also difficult to iterate on, because every researcher has their own bespoke setup, which often means a handful of messy scripts containing a handful of tightly-coupled methods and a ton of unhandled corner cases. I dreamed of cloning another researcher's project and immediately feeling confident enough to start hacking on their algorithms, so I created q2 to make that possible.

In the recent past, the field of web development faced a similar problem, with every developer having their own tools and techniques that they copy from project to project, making it hard for people to collaborate and learn each other's ideas. Their problem has largely been solved thanks to tools and frameworks like Ruby on Rails, AngularJS and React/create-react-app, many of which drew inspiration from manifestos like the Twelve Factor App. These tools encouraged greater standardization and common patterns across projects, solidifying good practices and reducing cognitive load.

My hypothesis is that machine learning and especially reinforcement learning face a similar problem. Fortunately, there are common use cases and patterns that we can optimize. The basic workflow that q2 targets is as follows:

1. Clone a project or spin up a new one with `q2 init` and `q2 generate`.
2. Build, tinker, iterate!
3. Run a training session with `q2 train ...`.

The goal of q2 is to make steps 1 and 3 completely seamless so that you can spend as much time as possible in step 2, confident that with q2 supporting you, all of your effort can be focused on the learning task at hand.

CHAPTER 4

Planned and upcoming features

- Better integration with Keras, PyTorch and other Tensorflow API wrappers
- Web interface to your agent zoo
- Improved deployment story: q2 should support common deployment patterns
- Show better training metrics, explore integrating with Tensorboard

CHAPTER 5

Bugs and feature requests

If you find something wrong, please create an issue on the Github page.

CHAPTER 6

About the author

I'm a software engineer and I hack on RL projects in my spare time. My personal website and blog can be found at tdb-alcorn.github.io.

Index