

# explore

January 17, 2024

## 1 Deep exploration with DQN

### 1.1 List of Contents

#### 1.1.1 Provided classes from exercise sheet 2

- `default_params()` this dictionary defines the default hyper-parameters
- `TransitionBatch` the basic class to summarize transitions and build an experience replay buffer.
- `Runner` interacts with one environment
- `MultiRunner` runs multiple `Runner` in parallel
- `QController` translates the model outputs into greedy actions
- `EpsilonGreedyController` performs epsilon-greedy exploration
- `QLearner` trains the model with Q-learning loss
- `Experiment` encapsulates and executes a single experiment
- `QLearningExperiment` performs online Q-learning

#### 1.1.2 Sample solution of exercise sheet 2

- `DQNExperiment`, `QLearnerHardTarget`, `QLearnerSoftTarget`, `DoubleQLearner`,

#### 1.1.3 New provided classes

- `RescaledEnv` this wrapper class rescales the states/observations of an environment to  $(-1,1)$
- `CountUncertainty` this class estimates the novelty/uncertainty of having seen states

#### 1.1.4 Exercises

- A4.1a) Run the `MountainCar` environment with rescaled states
- A4.1b) Implement intrinsic-reward for deep exploration
- A4.1c) Implement Random Network Distillation as intrinsic reward
- A4.1d) Compare `Acrobot-v1` with and without exploration

```
[3]: # Pytorch and tools
import torch as th
from torch import Tensor, LongTensor
from torch.utils.data import DataLoader
import numpy as np
from copy import deepcopy
import numbers
from datetime import datetime
```

```

# Multi-threading
import threading
# Plotting
from IPython import display
import matplotlib.pyplot as plt
import pylab as pl
# Reinforcement learning
import gym
import cv2

```

This dictionary defines the default hyper-parameters that you will use in your experiments.

```

[4]: def default_params():
    """ These are the default parameters used in these experiments. """
    return {# Debugging outputs and plotting during training
        'plot_frequency': 10,          # plots a debug message every n
    ↪ steps
        'plot_train_samples': True,    # whether the x-axis is env.steps
    ↪ (True) or episodes (False)
        'print_when_plot': True,       # prints debug message if True
        'print_dots': False,           # prints dots for every gradient
    ↪ update
        # Environment parameters
        'env': 'MountainCar-v0',       # the environment the agent is
    ↪ learning in
        'run_steps': 0,                # samples whole episodes if
    ↪ run_steps <= 0
        'max_episode_length': 200,     # maximum number of steps per
    ↪ episode
        # Runner parameters
        'max_episodes': int(1E6),      # experiment stops after this many
    ↪ episodes
        'max_steps': int(2E5),         # experiment stops after this many
    ↪ steps
        'multi_runner': False,         # uses multiple runners if True
        'parallel_environments': 4,    # number of parallel runners
    ↪ (only if multi_runner==True)
        # Exploration parameters
        'epsilon_anneal_time': int(2E4), # exploration anneals epsilon over
    ↪ these many steps
        'epsilon_finish': 0.05,        # annealing stops at (and keeps)
    ↪ this epsilon
        'epsilon_start': 1,            # annealing starts at this epsilon
        # Optimization parameters
        'lr': 1E-4,                    # learning rate of optimizer
        'gamma': 0.99,                 # discount factor gamma
    
```

```

        'batch_size': 2048,                # number of transitions in a
    ↪ mini-batch
        'grad_norm_clip': 1,              # gradient clipping if grad norm is
    ↪ larger than this
        # DQN parameters
        'replay_buffer_size': int(1E5),    # the number of transitions in the
    ↪ replay buffer
        'use_last_episode': False,         # whether the last episode is
    ↪ always sampled from the buffer
        'target_model': True,              # whether a target model is used
    ↪ in DQN
        'target_update': 'soft',           # 'soft' target update or hard
    ↪ update by regular 'copy'
        'target_update_interval': 10,      # interval for the 'copy' target
    ↪ update
        'soft_target_update_param': 0.01,  # update parameter for the 'soft'
    ↪ target update
        'double_q': True,                  # whether DQN uses double
    ↪ Q-learning
        'grad_repeats': 10,                # how many gradient updates /
    ↪ runner call
        # Image input parameters
        'pixel_observations': False,        # use pixel observations (we will
    ↪ not use this feature here)
        'pixel_resolution': (78, 78),      # scale image to this resolution
        'pixel_grayscale': True,            # convert image into grayscale
        'pixel_add_last_obs': True,         # stacks 2 observations
        'pixel_last_obs_delay': 3,          # delay between the two stacked
    ↪ observations
    }

```

TransitionBatches are dictionaries of variables, e.g. states or actions, that are saved in contiguous Tensors.

```

[5]: class TransitionBatch:
    """ Simple implementation of a batch of transitions (or another
    ↪ dictionary-based tensor structure).
        Read and write operations are thread-safe, but the iterator is not (you
    ↪ cannot iterate
        over the same TransitionBatch in two threads at the same time). """
    def __init__(self, max_size, transition_format, batch_size=32):
        self.lock = threading.Lock()
        self.indices = []
        self.size = 0
        self.first = 0
        self.max_size = max_size

```

```

        self.batch_size = batch_size
        self.dict = {}
        for key, spec in transition_format.items():
            self.dict[key] = th.zeros([max_size, *spec[0]], dtype=spec[1])

    def _clone_empty_batch(self, max_size=None, batch_size=None):
        """ Clones this TransitionBatch without cloning the data. """
        max_size = self.max_size if max_size is None else max_size
        batch_size = self.batch_size if batch_size is None else batch_size
        return TransitionBatch(max_size=max_size, transition_format={},
        ↪batch_size=batch_size)

    def __setitem__(self, key, newvalue):
        assert isinstance(key, str), "You can only assign entire Tensors of a
        ↪[key] like this."
        self.dict[key] = newvalue

    def __getitem__(self, key):
        """ Access the TransitionBatch with the [] operator. Use as key either
            - the string name of a variable to get the full tensor of that
        ↪variable,
            - a slice to get a time-slice over all variables in the batch,
            - a LongTensor that selects a subset of indices for all variables in
        ↪the batch. """
        # Return the entry of the transition called "key"
        if isinstance(key, str):
            return self.dict[key]
        # Return a slice of the batch
        if isinstance(key, slice):
            key = slice(0 if key.start is None else key.start, self.size if key.
        ↪stop is None else key.stop,
                        1 if key.step is None else key.step)
            self.lock.acquire()
            try:
                batch = self._clone_empty_batch()
                batch.size = (key.stop - key.start) // key.step
                for k, v in self.dict.items():
                    batch.dict[k] = v[key]
            finally: self.lock.release()
            return batch
        # Collect and return a set of transitions specified by the LongTensor
        ↪"key"
        if isinstance(key, th.Tensor):
            self.lock.acquire()
            try:

```

```

        batch = self._clone_empty_batch(max_size=key.shape[0])
        batch.size = key.shape[0]
        for k, v in self.dict.items():
            key = key.view(batch.size, *[1 for _ in range(len(v.shape[1:
→ ]))])

            batch.dict[k] = v.gather(dim=0, index=key.expand(batch.size,
→ *v.shape[1:]))
        finally: self.lock.release()
        return batch
    return None

def get_first(self):
    """ Returns a batch of the oldest entries of all variables. """
    batch = self._clone_empty_batch(max_size=1)
    self.lock.acquire()
    try:
        batch.size = 1
        for k, v in self.dict.items():
            batch.dict[k] = v[self.first].unsqueeze(dim=0)
    finally: self.lock.release()
    return batch

def get_last(self):
    """ Returns a batch of the newest entries of all variables. """
    batch = self._clone_empty_batch(max_size=1)
    self.lock.acquire()
    try:
        batch.size = 1
        for k, v in self.dict.items():
            batch.dict[k] = v[(self.first + self.size - 1) % self.size].
→ unsqueeze(dim=0)
    finally: self.lock.release()
    return batch

def add(self, trans:dict):
    """ Adding transition dictionaries, which can contain Tensors of
→ arbitrary length. """
    if isinstance(trans, TransitionBatch):
        trans = trans.dict
    # Add all data in the dict
    self.lock.acquire()
    try:
        n = 0
        idx = None
        for k, v in trans.items():
            if idx is None:
                n = v.shape[0]

```

```

        idx = th.LongTensor([(self.first + self.size + i) % self.
↪max_size for i in range(n)])
        else:
            assert n == v.shape[0], 'all tensors in a transition need to
↪have the same batch_size'
            idx = idx.view(idx.shape[0], *[1 for _ in range(len(v.shape) -
↪1)])

            self.dict[k].scatter_(dim=0, index=idx.expand_as(v), src=v)
            # Increase the size (and handle overflow)
            self.size += n
            if self.size > self.max_size:
                self.first = (self.first + n) % self.max_size
                self.size = self.max_size
            finally: self.lock.release()
            return self

    def trim(self):
        """ Reduces the length of the max_size to its actual size (in-place).
↪Returns self. """
        self.lock.acquire()
        try:
            for k, v in self.dict.items():
                self.dict[k] = v[:self.size]
            self.max_size = self.size
        finally: self.lock.release()
        return self

    def replace(self, batch, index=0):
        """ Replaces parts of this batch with another batch (which must be
↪smaller). """
        self.lock.acquire()
        try:
            #assert batch.max_size <= self.max_size - index, "Replacement is
↪larger then target area in batch."
            assert batch.size <= self.max_size - index, "Replacement is larger
↪then target area in batch."
            for k, v in batch.dict.items():
                if batch.size < batch.max_size:
                    v = v[:batch.size]
                self.dict[k][index:(index + batch.max_size)] = v
            finally: self.lock.release()

    def sample(self):
        """ Samples a random mini-batch from the batch. """
        return self[th.randint(high=self.size, size=(self.batch_size,1))]

```

```

def __len__(self):
    """ Returns the length of the batch. """
    return self.size

def __iter__(self):
    """ Initializes an iterator over the batch. """
    self.indices = list(range(self.size))
    np.random.shuffle(self.indices)
    return self

def __next__(self):
    """ Iterates through batch, returns list of contiguous tensors. """
    if len(self.indices) == 0: raise StopIteration
    size = min(self.batch_size, len(self.indices))
    batch = self[th.LongTensor(self.indices[-size:])]
    self.indices = self.indices[:-size]
    return batch

```

Runner implements a simple runner class that uses a controller to interact with the environment by calling `run()` or `run_episode()`.

```

[6]: class Runner:
    """ Implements a simple single-thread runner class. """
    def __init__(self, controller, params={}, exploration_step=1):
        self.env = gym.make(params.get('env', 'CartPole-v0'))
        self.cont_actions = isinstance(self.env.action_space, gym.spaces.Box)
        self.controller = controller
        self.epi_len = params.get('max_episode_length', self.env.
→ max_episode_steps)
        self.gamma = params.get('gamma', 0.99)
        self.use_pixels = params.get('pixel_observations', False)
        if self.use_pixels:
            self.grayscale = params.get('pixel_grayscale', True)
            self.add_last_obs = params.get('pixel_add_last_obs', False)
            self.last_obs_delay = params.get('pixel_last_obs_delay', 4)
            n_colors = 1 if self.grayscale else 3
            n_feats = n_colors * (2 if self.add_last_obs else 1)
            resolution = params.get('pixel_resolution', (25, 25))
            self.state_shape = (n_feats, *resolution)
            self.last_observations = TransitionBatch(max_size=self.
→ last_obs_delay,
                                                    transition_format={'img':␣
→ ((n_colors, *resolution), th.float32)})
        else:
            self.state_shape = self.env.observation_space.shape
            # Set up current state and time step
            self.sum_rewards = 0

```

```

self.state = None
self.time = 0
self._next_step()

def close(self):
    """ Closes the underlying environment. Should always be called when ending an
    → experiment. """
    self.env.close()

def transition_format(self):
    """ Returns the format of transitions: a dictionary of (shape, dtype)
    → entries for each key. """
    return {'actions': ((1,), th.long),
            'states': (self.state_shape, th.float32),
            'next_states': (self.state_shape, th.float32),
            'rewards': ((1,), th.float32),
            'dones': ((1,), th.bool),
            'returns': ((1,), th.float32)}

def _wrap_transition(self, s, a, r, ns, d):
    """ Takes a transition and returns a corresponding dictionary. """
    trans = {}
    form = self.transition_format()
    for key, val in [('states', s), ('actions', a), ('rewards', r),
    → ('next_states', ns), ('dones', d)]:
        if not isinstance(val, th.Tensor):
            if isinstance(val, numbers.Number) or isinstance(val, bool): val
            → = [val]
            val = th.tensor(val, dtype=form[key][1])
            if len(val.shape) < len(form[key][0]) + 1: val = val.unsqueeze(dim=0)
            trans[key] = val
    return trans

def _pixel_observation(self, reset=False):
    """ Returns the pixel-observation for the current state. Opens extra
    → window for rendering. """
    img = self.env.render(mode='rgb_array')
    img = cv2.resize(img, dsize=self.state_shape[1:], interpolation=cv2.
    → INTER_CUBIC)
    img = th.from_numpy(img.astype(np.float32) / 255).transpose(dim0=0,
    → dim1=2).unsqueeze(dim=0)
    if self.grayscale: img = img.mean(dim=1, keepdim=True)
    if self.add_last_obs:
        if reset: self.last_observations.size = 0
        if self.last_observations.size < self.last_observations.max_size:
            obs = img * 0

```



```

        else:
            obs = self.last_observations.get_first()['img'].clone()
            self.last_observations.add({'img': img})
            img = th.cat([obs, img], dim=1)
        return img

def _run_step(self, a):
    """ Make a step in the environment (and update internal bookeeping) """
    ns, r, t, d, _ = self.env.step(a.item())
    self.sum_rewards += r
    if self.use_pixels: ns = self._pixel_observation()
    return r, ns, t, d or t # reward, next state, terminal, done

def _next_step(self, done=True, next_state=None):
    """ Switch to the next time-step (and update internal bookeeping) """
    self.time = 0 if done else self.time + 1
    if done:
        self.sum_rewards = 0
        self.state, _ = self.env.reset()
        if self.use_pixels: self.state = self._pixel_observation(reset=True)
    else:
        self.state = next_state

def run(self, n_steps, transition_buffer=None, trim=True, return_dict=None):
    """ Runs n_steps in the environment and stores them in the_
    ↪ transition_buffer (newly created if None).
        If n_steps <= 0, stops at the end of an episode and optionally trims_
    ↪ the transition_buffer.
        Returns a dictionary containing the transition_buffer and episode_
    ↪ statistics. """
    my_transition_buffer = TransitionBatch(n_steps if n_steps > 0 else self.
    ↪ epi_len, self.transition_format())
    time, episode_start, episode_lengths, episode_rewards = 0, 0, [], []
    max_steps = n_steps if n_steps > 0 else self.epi_len
    for t in range(max_steps):
        # One step in the environment
        a = self.controller.choose(self.state)
        r, ns, terminal, done = self._run_step(a)
        my_transition_buffer.add(self._wrap_transition(self.state, a, r, ns,
    ↪ terminal))
        if self.env._elapsed_steps == self.epi_len: done = True
        # Compute discounted returns if episode has ended or max_steps has_
    ↪ been reached
        if done or t == (max_steps - 1):
            my_transition_buffer['returns'][t] =_
    ↪ my_transition_buffer['rewards'][t]

```

```

        for i in range(t - 1, episode_start - 1, -1):
            my_transition_buffer['returns'][i] =
→my_transition_buffer['rewards'][i] \
                                                    + self.gamma *
→my_transition_buffer['returns'][i + 1]
            episode_start = t + 1
            # Remember statistics and advance (potentially initilaizing a new
→episode)
            if done:
                episode_lengths.append(self.time + 1)
                episode_rewards.append(self.sum_rewards)
                self._next_step(done=done, next_state=ns)
                time += 1
                # If n_steps <= 0, we return after one episode (trimmed if specified)
                if done and n_steps <= 0:
                    my_transition_buffer.trim()
                    break
            # Add the sampled transitions to the given transition buffer
            transition_buffer = my_transition_buffer if transition_buffer is None \
                                else transition_buffer.add(my_transition_buffer)
            if trim: transition_buffer.trim()
            # Return statistics (mean reward, mean length and environment steps)
            if return_dict is None: return_dict = {}
            return_dict.update({'buffer': transition_buffer,
                                'episode_reward': None if len(episode_rewards) == 0
→else np.mean(episode_rewards),
                                'episode_length': None if len(episode_lengths) == 0
→else np.mean(episode_lengths),
                                'env_steps': time})
            return return_dict

    def run_episode(self, transition_buffer=None, trim=True, return_dict=None):
        """ Runs one episode in the environemt.
        Returns a dictionary containing the transition_buffer and episode
→statistics. """
        return self.run(0, transition_buffer, trim, return_dict)

```

MultiRunner runs a number of Runner instances in parallel.

```

[7]: class MultiRunner:
    """ Simple class that runs multiple Runner objects in parallel and merges
→their outputs. """
    def __init__(self, controller, params={}):
        self.workers = []
        self.runners = []
        n = params.get('parallel_environments', 1)
        for _ in range(n):

```

```

        self.runners.append(Runner(controller=controller, params=params))

def transition_format(self):
    """ Same transition-format as underlying Runners. """
    return self.runners[0].transition_format()

def close(self):
    """ Closes the underlying environment. Should always when ending an
    ↪ experiment. """
    # Join all workers
    for w in self.workers:
        w.join()
    # Exit all environments
    for r in self.runners:
        r.close()

def fork(self, target, common_args=None, specific_args=None):
    """ Executes the function "target" on all runners. "common_args" is a
    ↪ dictionary of
        arguments that are passed to all runners, "specific_args" is a list
    ↪ of
        dictionaries that contain individual parameters for each runner. """
    # Fork all runners
    self.workers = []
    for i, r in enumerate(self.runners):
        r_args = [] if specific_args is None else [arg[i] for arg in
    ↪ specific_args]
        self.workers.append(threading.Thread(target=target, args=(r,
    ↪ *common_args, *r_args)))
        self.workers[-1].start()
    # Join all runners
    for w in self.workers:
        w.join()

def run(self, n_steps, transition_buffer=None, trim=True):
    """ Runs n_steps, split amongst runners, and stores them in the
    ↪ transition_buffer (newly created if None).
        If n_steps <= 0, stops at the end of an episode and optionally trims
    ↪ the transition_buffer.
        Returns a dictionary containing the transition_buffer and episode
    ↪ statistics. """
    n_steps = n_steps // len(self.runners)
    if transition_buffer is None:
        buffer_len = len(self.runners) * (n_steps if n_steps > 0 else self.
    ↪ runners[0].epi_len)

```

```

        transition_buffer = TransitionBatch(buffer_len, self.runners[0].
→transition_format())
        return_dicts = [{ } for _ in self.runners]
        self.fork(target=Runner.run, common_args=(n_steps, transition_buffer,
→False), specific_args=(return_dicts,))
        if trim: transition_buffer.trim()
        rewards = [d['episode_reward'] for d in return_dicts if
→d['episode_reward'] is not None]
        lengths = [d['episode_length'] for d in return_dicts if
→d['episode_reward'] is not None]
        return {'buffer': transition_buffer,
                'episode_reward': np.mean(rewards) if len(rewards) > 0 else None,
                'episode_length': np.mean(lengths) if len(lengths) > 0 else None,
                'env_steps': len(transition_buffer)}

    def run_episode(self, transition_buffer=None, trim=True):
        """ Runs one episode in the environemt.
        Returns a dictionary containing the transition_buffer and episode_
→statistics. """
        return self.run(0, transition_buffer, trim)

```

A QController translates model responses into actions. Call `choose()` to select actions or `probabilities()` to get the probabilities with which the controller would choose the actions.

```

[8]: class QController:
    """ Controller for Q-value functions, synchronizes the model calls. """
    def __init__(self, model, num_actions=None, params={}):
        self.lock = threading.Lock()
        self.num_actions = model[-1].out_features if num_actions is None else
→num_actions
        self.model = model

    def copy(self):
        """ Shallow copy of this controller that does not copy the model. """
        return QController(model=self.model, num_actions=self.num_actions)

    def parameters(self):
        """ Returns a generator of the underlying model parameters. """
        return self.model.parameters()

    def sanitize_inputs(self, observation, **kwargs):
        """ Casts numpy arrays as Tensors. """
        if isinstance(observation, np.ndarray):
            observation = th.Tensor(observation).unsqueeze(dim=0)
        return observation

    def choose(self, observation, **kwargs):

```

```

        """ Returns the greedy actions the agent would choose when facing an
        ↪ "observation". """
        self.lock.acquire()
        try:
            mx = self.model(self.sanitize_inputs(observation))
            if mx.shape[-1] > self.num_actions: mx = mx[:, :self.num_actions]
        finally: self.lock.release()
        return th.max(mx, dim=-1)[1]

    def probabilities(self, observation, **kwargs):
        """ Returns the probabilities with which the agent would choose actions
        ↪ (here one-hot because greedy). """
        self.lock.acquire()
        try:
            mx = self.model(self.sanitize_inputs(observation))
            if mx.shape[-1] > self.num_actions: mx = mx[:, :self.num_actions]
        finally: self.lock.release()
        return th.zeros(*mx.shape).scatter_(dim=-1, index=th.max(mx, dim=-1)[1].
        ↪ unsqueeze(dim=-1), src=th.ones(1, 1))

```

An EpsilonGreedyController is a controller that autonomously anneals an epsilon greedy exploration strategy.

```

[9]: class EpsilonGreedyController:
    """ A wrapper that makes any controller into an epsilon-greedy controller.
        Keeps track of training-steps to decay exploration automatically. """
    def __init__(self, controller, params={}, exploration_step=1):
        self.controller = controller
        self.num_actions = controller.num_actions
        self.max_eps = params.get('epsilon_start', 1.0)
        self.min_eps = params.get('epsilon_finish', 0.05)
        self.anneal_time = int(params.get('epsilon_anneal_time', 10000) /
        ↪ exploration_step)
        self.num_decisions = 0

    def epsilon(self):
        """ Returns current epsilon. """
        return max(1 - self.num_decisions / (self.anneal_time - 1), 0) \
            * (self.max_eps - self.min_eps) + self.min_eps

    def choose(self, observation, increase_counter=True, **kwargs):
        """ Returns the (possibly random) actions the agent takes when faced
        ↪ with "observation".
            Decays epsilon only when increase_counter=True. """
        eps = self.epsilon()
        if increase_counter: self.num_decisions += 1
        if np.random.rand() < eps:

```

```

        return th.randint(self.controller.num_actions, (1,)), dtype=th.long)
    else:
        return self.controller.choose(observation, **kwargs)

    def probabilities(self, observation, **kwargs):
        """ Returns the probabilities with which the agent would choose actions.
        ↪ """
        eps = self.epsilon()
        return eps * th.ones(1, 1) / self.num_actions + \
            (1 - eps) * self.controller.probabilities(observation, **kwargs)

```

A QLearner is a learner class that performs Q-learning. At the moment this does not include target models or double Q-learning, which you will add in later exercises.

```

[10]: class QLearner:
        """ A basic learner class that performs Q-learning train() steps. """
        def __init__(self, model, params={}):
            self.model = model
            self.all_parameters = list(model.parameters())
            self.gamma = params.get('gamma', 0.99)
            self.optimizer = th.optim.Adam(self.all_parameters, lr=params.get('lr',
            ↪ 5E-4))
            self.criterion = th.nn.MSELoss()
            self.grad_norm_clip = params.get('grad_norm_clip', 10)
            self.target_model = None # Target models are not yet implemented!

        def target_model_update(self):
            """ This function updates the target network. No target network is
            ↪ implemented yet. """
            pass

        def q_values(self, states, target=False):
            """ Reutrns the Q-values of the given "states".
                I supposed to use the target network if "target=True", but this is
            ↪ not implemented here. """
            return self.model(states)

        def _current_values(self, batch):
            """ Computes the Q-values of the 'states' and 'actions' of the given
            ↪ "batch". """
            qvalues = self.q_values(batch['states'])
            return qvalues.gather(dim=-1, index=batch['actions'])

        def _next_state_values(self, batch):
            """ Computes the Q-values of the 'next_states' of the given "batch".
                Is greedy w.r.t. to current Q-network or target-network, depending
            ↪ on parameters. """

```

```

        with th.no_grad(): # Next state values do not need gradients in DQN
            # Compute the next states values (with target or current network)
            qvalues = self.q_values(batch['next_states'], target=True)
            # Compute the maximum over Q-values
            return qvalues.max(dim=-1, keepdim=True)[0]

    def train(self, batch):
        """ Performs one gradient decent step of DQN. """
        self.model.train(True)
        # Compute TD-loss
        targets = batch['rewards'] + self.gamma * (~batch['dones'] * self.
→_next_state_values(batch))
        loss = self.criterion(self._current_values(batch), targets.detach())
        # Backpropagate loss
        self.optimizer.zero_grad()
        loss.backward()
        grad_norm = th.nn.utils.clip_grad_norm_(self.all_parameters, self.
→grad_norm_clip)
        self.optimizer.step()
        # Update target network (if specified) and return loss
        self.target_model_update()
        return loss.item()

```

An Experiment is an abstract class that starts and maintains a single learning experiment (i.e. random seed). The experiment is started using `run()` and can be interrupted at any time using `close()`. Afterwards the experiment can be restarted at any time calling `run()` again.

```

[11]: class Experiment:
        """ Abstract class of an experiment. Contains logging and plotting_
→functionality. """
        def __init__(self, params, model, **kwargs):
            self.params = params
            self.plot_frequency = params.get('plot_frequency', 100)
            self.plot_train_samples = params.get('plot_train_samples', True)
            self.print_when_plot = params.get('print_when_plot', False)
            self.print_dots = params.get('print_dots', False)
            self.episode_returns = []
            self.episode_lengths = []
            self.episode_losses = []
            self.env_steps = []
            self.total_run_time = 0.0

        def plot_training(self, update=False):
            """ Plots logged training results. Use "update=True" if the plot is_
→continuously updated
            or use "update=False" if this is the final call (otherwise there_
→will be double plotting). """

```

```

    # Smooth curves
    window = max(int(len(self.episode_returns) / 50), 10)
    if len(self.episode_losses) < window + 2: return
    returns = np.convolve(self.episode_returns, np.ones(window)/window,
↪'valid')
    lengths = np.convolve(self.episode_lengths, np.ones(window)/window,
↪'valid')
    losses = np.convolve(self.episode_losses, np.ones(window)/window,
↪'valid')
    env_steps = np.convolve(self.env_steps, np.ones(window)/window, 'valid')
    # Determine x-axis based on samples or episodes
    if self.plot_train_samples:
        x_returns = env_steps
        x_losses = env_steps[(len(env_steps) - len(losses)):]
    else:
        x_returns = [i + window for i in range(len(returns))]
        x_losses = [i + len(returns) - len(losses) + window for i in
↪range(len(losses))]
    # Create plot
    colors = ['b', 'g', 'r']
    fig = plt.gcf()
    fig.set_size_inches(16, 4)
    plt.clf()
    # Plot the losses in the left subplot
    pl.subplot(1, 3, 1)
    pl.plot(env_steps, returns, colors[0])
    pl.xlabel('environment steps' if self.plot_train_samples else 'episodes')
    pl.ylabel('episode return')
    # Plot the episode lengths in the middle subplot
    ax = pl.subplot(1, 3, 2)
    ax.plot(env_steps, lengths, colors[0])
    ax.set_xlabel('environment steps' if self.plot_train_samples else
↪'episodes')
    ax.set_ylabel('episode length')
    # Plot the losses in the right subplot
    ax = pl.subplot(1, 3, 3)
    ax.plot(x_losses, losses, colors[0])
    ax.set_xlabel('environment steps' if self.plot_train_samples else
↪'episodes')
    ax.set_ylabel('loss')
    # dynamic plot update
    display.clear_output(wait=True)
    if update:
        display.display(pl.gcf())

def close(self):

```



```

        """ Frees all allocated runtime resources, but allows to continue the
        ↪ experiment later.
        Calling the run() method after close must be able to pick up the
        ↪ experiment where it was. """
        pass

    def run(self):
        """ Starts (or continues) the experiment. """
        assert False, "You need to extend the Experiment class and override the
        ↪ method run(). "

```

QLearningExperiment performs online Q-learning using QLearner. One can specify another learner, which you will do in later exercises.

```

[12]: class QLearningExperiment (Experiment):
        """ Experiment that perfoms DQN. You can provide your own learner. """
        def __init__(self, params, model, learner=None, **kwargs):
            super().__init__(params, model, **kwargs)
            self.max_episodes = params.get('max_episodes', int(1E6))
            self.max_steps = params.get('max_steps', int(1E9))
            self.run_steps = params.get('run_steps', 0)
            self.grad_repeats = params.get('grad_repeats', 1)
            self.controller = QController(model, num_actions=gym.make(params['env']).
            ↪ action_space.n, params=params)
            self.controller = EpsilonGreedyController(controller=self.controller,
            ↪ params=params)
            self.runner = MultiRunner(self.controller, params=params) if params.
            ↪ get('multi_runner', True) \
                else Runner(self.controller, params=params)
            self.learner = QLearner(model, params=params) if learner is None else
            ↪ learner

        def close(self):
            """ Overrides Experiment.close(). """
            self.runner.close()

        def _learn_from_episode(self, episode):
            """ This function uses the episode to train.
            Although not implemented, one could also add the episode to a replay
            ↪ buffer here.
            Returns the training loss for logging or None if train() was not
            ↪ called. """
            # Call train (params['grad_repeats']) times
            total_loss = 0
            for i in range(self.grad_repeats):
                total_loss += self.learner.train(episode['buffer'])
            return total_loss / self.grad_repeats

```

```

def run(self):
    """ Starts (or continues) the experiment. """
    # Plot previous results if they exist
    if self.plot_frequency is not None and len(self.episode_losses) > 2:
        self.plot_training(update=True)
    # Start (or continue experiment)
    env_steps = 0 if len(self.env_steps) == 0 else self.env_steps[-1]
    for e in range(self.max_episodes):
        begin_time = datetime.now()
        # Run an episode (or parts of it)
        if self.run_steps > 0:
            episode = self.runner.run(n_steps=self.run_steps, trim=False)
        else:
            episode = self.runner.run_episode()
        # Log the results
        env_steps += episode['env_steps']
        if episode['episode_length'] is not None:
            self.episode_lengths.append(episode['episode_length'])
            self.episode_returns.append(episode['episode_reward'])
            self.env_steps.append(env_steps)
        # Make one (or more) learning steps with the episode
        loss = self._learn_from_episode(episode)
        if loss is not None: self.episode_losses.append(loss)
        self.total_run_time += (datetime.now() - begin_time).total_seconds()
        # Quit if maximal number of environment steps is reached
        if env_steps >= self.max_steps:
            break
        # Show intermediate results
        if self.print_dots:
            print('.', end='')
        if self.plot_frequency is not None and (e + 1) % self.plot_frequency == 0 \
            and len(self.episode_losses) > 2:
            self.plot_training(update=True)
            if self.print_when_plot:
                print('Update %u, 100-epi-return %.4g +- %.3g, length %u, \
↳loss %g, run-time %g sec.' %
                    (len(self.episode_returns), np.mean(self.
↳episode_returns[-100:]),
                    np.std(self.episode_returns[-100:]), np.mean(self.
↳episode_lengths[-100:]),
                    np.mean(self.episode_losses[-100:]), self.
↳total_run_time))

```

The following classes are the unmodified sample solution of Q3.2 in exercise sheet 3:

```

[13]: class DQNExperiment (QLearningExperiment):
    """ Experiment that performs DQN. You can provide your own learner. """
    def __init__(self, params, model, learner=None, **kwargs):
        super().__init__(params, model, learner=learner, **kwargs)
        self.use_last_episode = params.get('use_last_episode', True)
        self.replay_buffer = TransitionBatch(params.get('replay_buffer_size',
→int(1E5)),
                                           self.runner.transition_format(),
→1024))
                                           batch_size=params.get('batch_size',

    def _learn_from_episode(self, episode):
        self.replay_buffer.add(episode['buffer'])
        # Make a gradient update step
        if self.replay_buffer.size >= self.replay_buffer.batch_size:
            # Call train() params['grad_repeats'] times
            total_loss = 0
            for i in range(self.grad_repeats):
                sampled_batch = self.replay_buffer.sample()
                if self.use_last_episode:
                    sampled_batch.replace(episode['buffer'] if
→len(sampled_batch) >= len(episode['buffer'])
                                           else
→episode['buffer'][:len(sampled_batch)])
                    total_loss += self.learner.train(sampled_batch)
                # returned the averaged loss
            return total_loss / self.grad_repeats
        else:
            return None

class QLearnerHardTarget (QLearner):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.target_update = params.get('target_update', 'hard')
        self.target_update_interval = params.get('target_update_interval', 200)
        self.target_update_calls = 0
        if params.get('target_model', True):
            self.target_model = deepcopy(model)
            for p in self.target_model.parameters():
                p.requires_grad = False
            assert self.target_model is None or self.target_update == 'soft' or self.
→target_update == 'copy',\
                'If a target model is specified, it needs to be updated using the
→"soft" or "copy" options.'

        def target_model_update(self):

```

```

        """ This function updates the target network. """
        if self.target_model is not None:
            # Target network update by copying it every so often
            if self.target_update == 'copy':
                self.target_update_calls = (self.target_update_calls + 1) % self.
→target_update_interval
                if self.target_update_calls == 0:
                    self.target_model.load_state_dict(self.model.state_dict())

        def q_values(self, states, target=False):
            """ Reutrns the Q-values of the given "states". Uses the target network_
→if "target=True". """
            target = target and self.target_model is not None
            return (self.target_model if target else self.model)(states)

class QLearnerSoftTarget (QLearnerHardTarget):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.target_update = params.get('target_update', 'soft')
        self.soft_target_update_param = params.get('soft_target_update_param', 0.
→1)

    def target_model_update(self):
        """ This function updates the target network. """
        if self.target_model is not None:
            # Target network update by copying it every so often
            if self.target_update == 'copy':
                super().target_model_update()
            elif self.target_update == 'soft':
                for tp, mp in zip(self.target_model.parameters(), self.model.
→parameters()):
                    tp *= (1 - self.soft_target_update_param)
                    tp += self.soft_target_update_param * mp.detach()

class DoubleQLearner (QLearnerSoftTarget):
    def __init__(self, model, params={}):
        super().__init__(model, params)
        self.double_q = params.get('double_q', True)

    def _next_state_values(self, batch):
        """ Computes the Q-values of the 'next_states' of the given "batch".
            Is greedy w.r.t. to current Q-network or target-network, depending_
→on parameters. """
        with th.no_grad(): # Next state values do not need gradients in DQN
            # Compute the next states values (with target or current network)

```

```

        qvalues = self.q_values(batch['next_states'], target=True)
        # Compute the maximum (note the case of double Q-learning)
        if self.target_model is None or not self.double_q:
            # If we do not do double Q-learning or if there is not target
            ↪network
            qvalues = qvalues.max(dim=-1, keepdim=True)[0]
        else:
            # If we do double Q-learning
            next_values = self.q_values(batch['next_states'], target=False)
            actions = next_values.max(dim=-1)[1].unsqueeze(dim=-1)
            qvalues = qvalues.gather(dim=-1, index=actions)
        return qvalues

```

The class `RescaledEnv` wraps around an environment `Env` class and rescales the all state/observation variables to be in the range  $(-1, 1)$ .

```

[14]: class RescaledEnv:
        def __init__(self, env, max_episode_length=None):
            self.env = env
            self.bounds = [(l, h) for l, h in zip(env.observation_space.low, env.
            ↪observation_space.high)]
            self._elapsed_steps = self.env._elapsed_steps
            if max_episode_length is not None: self.env._max_episode_steps =
            ↪max_episode_length

            def rescale(self, state):
                return np.array([2 * (x - l) / (h - l) - 1 for x, (l, h) in zip(state,
                ↪self.bounds)])

            def step(self, action):
                ns, r, t, d, x = self.env.step(action)
                self._elapsed_steps = self.env._elapsed_steps
                return self.rescale(ns), r, t, d, x

            def reset(self):
                state, descriptor = self.env.reset()
                self._elapsed_steps = self.env._elapsed_steps
                return self.rescale(state), descriptor

            def render(self, mode="human"):
                self.env.render(mode)

            def close(self):
                self.env.close()

            def seed(self, seed=None):
                return self.env.seed()

```

The class `CountUncertainty` counts observed states/observations in a specified resolution. The call function `varname(state)` returns the estimated uncertainty scale /  $\sqrt{n}$  for  $n$  observations of the given state/observation.

```
[15]: class CountUncertainty:
    """ Defines an uncertainty estimate based on counts over the state/
    ↪ observation space.
        Uncertainty will be scaled by 'scale'. Define boundaries either by
    ↪ 'state_bounds'
        or automatically by passing the environment 'env'. The counts will use
        'resolution'~m different bins for m-dimensional state vecotrs"""
    def __init__(self, scale=1, env=None, state_bounds=None, resolution=50):
        if state_bounds is None:
            self.bounds = [(l, h) for l,h in zip(env.observation_space.low, env.
            ↪ observation_space.high)]
        else:
            self.bounds = state_bounds
        self.resolution = resolution
        self.count = th.zeros(*([resolution for _ in self.bounds]), dtype=th.
        ↪ long)
        self.scale = scale
        self.eps = 1E-7 # this constant is used to avoid division-by-zero and
        ↪ boundary-effects

    def state_bin(self, state):
        """ Find the correct bin in 'self.count' for one state. """
        return tuple([int((x - l) / (h - l + self.eps) * self.resolution) for x,
        ↪ (l, h) in zip(state, self.bounds)])

    def observe(self, state, **kwargs):
        """ Add counts for observed 'state's.
            'state' can be either a Tuple, List, 1d Tensor or 2d Tensor (1d
        ↪ Tensors stacked in dim=0). """
        if isinstance(state, th.Tensor):
            if len(state.shape) == 1: state.unsqueeze_(dim=0)
        else:
            state = [state]
        for s in state:
            b = self.state_bin(s)
            self.count[b] += 1

    def __call__(self, state, **kwargs):
        """ Returns the estimated uncertainty for observing a (minibatch of)
        ↪ state(s) ans Tensor.
            'state' can be either a Tuple, List, 1d Tensor or 2d Tensor (1d
        ↪ Tensors stacked in dim=0).
            Does not change the counters (i.e. does not 'observe'). """
```

```

    if isinstance(state, th.Tensor):
        if len(state.shape) == 1: state.unsqueeze_(dim=0)
    else:
        state = [state]
    n = th.zeros(len(state))
    for i, s in enumerate(state):
        b = self.state_bin(s)
        n[i] = self.count[b]
    return self.scale / np.sqrt(n + self.eps)

```

## 1.2 A4.1a) Run the MountainCar environment with rescaled states

The Mountaincar environment is challenging because of the exploration behavior, random episodes will rarely see *any* reward before the maximum length is reached, but also because of its extremely unevenly scaled state space: while the car's position is between -1.2 and +0.6, the car's velocity is bounded between -0.07 and +0.07. Run the following experiment, which changes this state space by rescaling each dimension to be between -1 and 1 using the provided `RescaledEnv` wrapper class, for 200k environmental steps. To make sure you propagate rewards fast enough to see a result in that time frame (if there is one), run 10 gradient updates for every sampled episode.

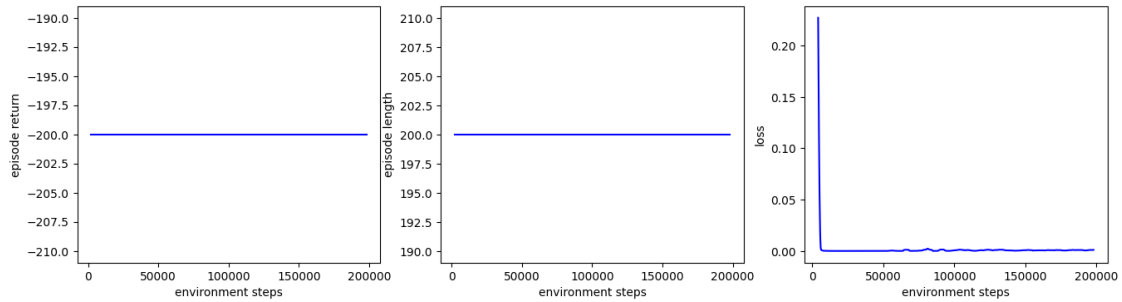
```

[16]: class RescaledDQNEperiment (DQNEperiment):
    def __init__(self, params, model, learner=None, **kwargs):
        super().__init__(params, model, learner, **kwargs)
        assert not params.get('multi_runner', True), "This experiment does not,
        ↪work with the MultiRunner. "
        self.runner.env = RescaledEnv(self.runner.env, max_episode_length=params.
        ↪get('max_episode_length', None))

params = default_params()
env = gym.make(params['env'])
n_actions, state_dim = env.action_space.n, env.observation_space.shape[0]
model = th.nn.Sequential(th.nn.Linear(state_dim, 128), th.nn.ReLU(),
                        th.nn.Linear(128, 512), th.nn.ReLU(),
                        th.nn.Linear(512, 128), th.nn.ReLU(),
                        th.nn.Linear(128, n_actions))
experiment = RescaledDQNEperiment(params, model, learner=DoubleQLearner(model,
        ↪params))

# Re-executing this code-block picks up the experiment where you left off
try:
    experiment.run()
except KeyboardInterrupt:
    experiment.close()
experiment.plot_training()

```



### 1.3 A4.1b) Implement intrinsic-reward for deep exploration

To solve the `Mountaincar-v0` environment, you will need to use some deep exploration technique. To test this we will “cheat” a bit at first: use the provided `CountUncertainty` class to produce intrinsic reward during training. The class must `observe()` states from newly sampled episodes (make sure each transition in your replay buffer has only been observed once) and provides a scaled uncertainty estimate with a `()` method call, e.g. `u(state)` for `u = CountUncertainty(...)`. This particular uncertainty class divides the state space into  $\text{uncertainty\_resolution}^m$  bins, where  $m$  is the number of state-dimensions, counts how often observed states fall into these bins and estimates the resulting uncertainty with  $\text{uncertainty\_scale} / \sqrt{n}$  for  $n$  observations of the same states. Complete the implementation of `ExplorationDQNEExperiment` and run the `Mountaincar-v0` environment with intrinsic rewards for 200k steps. Make sure that the experiment only uses intrinsic reward if the `intrinsic_reward` parameter is `True`.

*Hint:* there are multiple ways to implement intrinsic reward, but the most intuitive is to use the uncertainty of having seen the *next state* of a transition as additional reward. Make sure that you have observed those states before you compute their uncertainty, though, as states with observation-counts of 0 produce extremely large uncertainties that can destabilize learning.

```
[33]: class ExplorationDQNEExperiment (RescaledDQNEExperiment):
    def __init__(self, params, model, learner=None, uncertainty=None, **kwargs):
        super().__init__(params, model, learner, **kwargs)
        self.uncertainty = uncertainty
        self.intrinsic_reward = params.get('intrinsic_reward', True) and
        ↪uncertainty is not None

    # YOUR CODE HERE!!!
    def _learn_from_episode(self, episode):
        #-----
        if self.intrinsic_reward:
            self.uncertainty.observe(episode['buffer']['next_states'])
            estimated_uncertainty = self.
            ↪uncertainty(episode['buffer']['next_states']).unsqueeze(dim=1)
            episode['buffer']['rewards'] += estimated_uncertainty
        #-----
        self.replay_buffer.add(episode['buffer'])
```



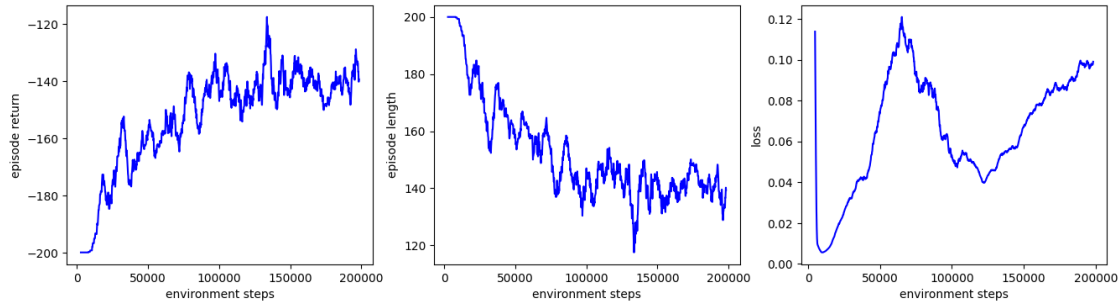
```

        # Make a gradient update step
        if self.replay_buffer.size >= self.replay_buffer.batch_size:
            # Call train() params['grad_repeats'] times
            total_loss = 0
            for i in range(self.grad_repeats):
                sampled_batch = self.replay_buffer.sample()
                if self.use_last_episode:
                    sampled_batch.replace(episode['buffer'] if
→len(sampled_batch) >= len(episode['buffer'])
                                                    else
→episode['buffer'][:len(sampled_batch)])
                    total_loss += self.learner.train(sampled_batch)
                # returned the averaged loss
            return total_loss / self.grad_repeats
        else:
            return None

params = default_params()
params['intrinsic_reward'] = True
params['uncertainty_scale'] = 0.5
params['uncertainty_resolution'] = 50
env = gym.make(params['env'])
n_actions, state_dim = env.action_space.n, env.observation_space.shape[0]
model = th.nn.Sequential(th.nn.Linear(state_dim, 128), th.nn.ReLU(),
                        th.nn.Linear(128, 512), th.nn.ReLU(),
                        th.nn.Linear(512, 128), th.nn.ReLU(),
                        th.nn.Linear(128, n_actions))
uncertainty = CountUncertainty(scale=params.get('uncertainty_scale', 1.0),
→state_bounds=[(-1,1)] * env.observation_space.
→shape[0],
                                resolution=params.get('uncertainty_resolution',
→50))
experiment = ExplorationDQNExperiment(params, model,
→learner=DoubleQLearner(model, params), uncertainty=uncertainty)

# Re-executing this code-block picks up the experiment where you left off
try:
    experiment.run()
except KeyboardInterrupt:
    experiment.close()
experiment.plot_training()

```



## 1.4 A4.1c) Implement Random Network Distillation as intrinsic reward

Your above implementation of `ExplorationDQNExperiment` should be able to solve the `Mountaincar-v0` now. However, the `CountUncertainty` class does not scale to other environments. For example, the `Acrobot-v1` environment has 6-dimensional states and would induce over 15 billion bins if we would count with the same resolution as above. To use intrinsic rewards in high dimensional state spaces, you will complete the implementation of Random Network Distillation (slide 19 of Lecture 7) in the `RNDUncertainty` class. The RND uncertainty estimate shall use 3 linear layers with hidden dimension 1024, ReLU's between them and an output dimension of 256 (e.g. 2-1024-1024-256 for 2-dimensional `Mountaincar-v0` states). Test your implementation on the `Mountaincar-v0` environment with intrinsic rewards for 200k steps.

*Hint:* useful `uncertainty_scale` parameters depend a lot on your exact implementation of RND. It is recommended to print the average intrinsic reward in `Mountaincar-v0` for `CountUncertainty` (e.g. in the previous question) and then change `uncertainty_scale` for this question until `RNDUncertainty` yields similar intrinsic rewards at the beginning of training. A good value for average intrinsic rewards of an initial episode is around 0.1.

```
[55]: class RNDUncertainty:
    """ This class uses Random Network Distillation to estimate the uncertainty/
    ↪novelty of states. """
    def __init__(self, scale, env, hidden_dim=1024, embed_dim=256, **kwargs):
        self.scale = scale
        self.criterion = th.nn.MSELoss(reduction='none')
        # YOUR CODE HERE
        self.psi_net = th.nn.Sequential(th.nn.Linear(2, hidden_dim), th.nn.
        ↪ReLU(),
                                     th.nn.Linear(hidden_dim, hidden_dim), th.
        ↪nn.ReLU(),
                                     th.nn.Linear(hidden_dim, embed_dim), th.
        ↪nn.ReLU())
        self.si_net = th.nn.Sequential(th.nn.Linear(2, hidden_dim), th.nn.
        ↪ReLU(),
                                     th.nn.Linear(hidden_dim, hidden_dim), th.
        ↪nn.ReLU(),
```

```

th.nn.Linear(hidden_dim, embed_dim), th.
→nn.ReLU())

self.optimizer = th.optim.Adam(list(self.si_net.parameters()), lr=0.0001)

def error(self, state):
    """ Computes the error between the prediction and target network. """
    if not isinstance(state, th.Tensor): state = th.tensor(state)
    if len(state.shape) == 1: state.unsqueeze(dim=0)
    # YOUR CODE HERE: return the RND error
    target = self.psi_net(state)
    prediction = self.si_net(state)
    return self.criterion(prediction, target)

def observe(self, state, **kwargs):
    """ Observes state(s) and 'remembers' them using Random Network
→Distillation"""
    # YOUR CODE HERE
    error = self.error(state)
    self.optimizer.zero_grad()
    error.mean().backward()
    self.optimizer.step()

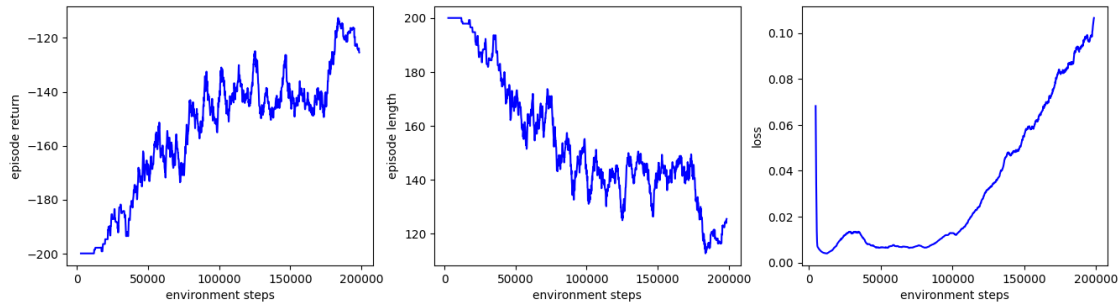
def __call__(self, state, **kwargs):
    """ Returns the estimated uncertainty for observing a (minibatch of)
→state(s) as Tensor. """
    # YOUR CODE HERE
    error = self.error(state)
    return error.mean(dim=1) * self.scale

params = default_params()
params['intrinsic_reward'] = True
params['uncertainty_scale'] = 200
env = gym.make(params['env'])
n_actions, state_dim = env.action_space.n, env.observation_space.shape[0]
model = th.nn.Sequential(th.nn.Linear(state_dim, 128), th.nn.ReLU(),
                        th.nn.Linear(128, 512), th.nn.ReLU(),
                        th.nn.Linear(512, 128), th.nn.ReLU(),
                        th.nn.Linear(128, n_actions))
experiment = ExplorationDQNExperiment(params, model,
→learner=DoubleQLearner(model, params),
                                )
→uncertainty=RNDUncertainty(scale=params['uncertainty_scale'], env=env))

# Re-executing this code-block picks up the experiment where you left off
try:
    experiment.run()

```

```
except KeyboardInterrupt:
    experiment.close()
experiment.plot_training()
```

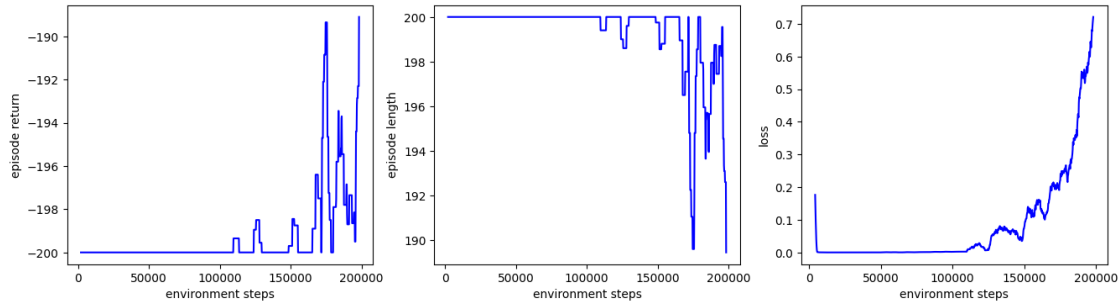


## 1.5 A4.1d) Compare Acrobot-v1 with and without exploration

The `Acrobot-v1` environment requires the agent to learn how to swing up a chain of two connected links. The joint between the two links is (under-) actuated, and swinging the Acrobot can exhibit chaotic behavior. Normally this environment allows episodes of up to 500 steps to ensure the agent sees at least some rewards using random exploration. We make this here a bit harder by restricting episodes to 200 steps (like in `Mountaincar-v0`). First run double DQN without intrinsic rewards for 200k steps to evaluate whether random exploration is enough to learn in this environment.

```
[56]: params = default_params()
params['env'] = 'Acrobot-v1'
env = gym.make(params['env'])
n_actions, state_dim = env.action_space.n, env.observation_space.shape[0]
model = th.nn.Sequential(th.nn.Linear(state_dim, 128), th.nn.ReLU(),
                        th.nn.Linear(128, 512), th.nn.ReLU(),
                        th.nn.Linear(512, 128), th.nn.ReLU(),
                        th.nn.Linear(128, n_actions))
experiment = RescaledDQNExperiment(params, model, learner=DoubleQLearner(model,
    ↪params))

# Re-executing this code-block picks up the experiment where you left off
try:
    experiment.run()
except KeyboardInterrupt:
    experiment.close()
experiment.plot_training()
```



Next run `Acrobot-v1` with `RNDUncertainty` intrinsic reward. If you do not see similar learning as in `Mountcar-v0`, try to adjust the `uncertainty_scale` with the same techniques as in A4.1c.

```
[57]: params = default_params()
params['env'] = 'Acrobot-v1'
params['intrinsic_reward'] = True
params['uncertainty_scale'] = 2
env = gym.make(params['env'])
n_actions, state_dim = env.action_space.n, env.observation_space.shape[0]
model = th.nn.Sequential(th.nn.Linear(state_dim, 128), th.nn.ReLU(),
                        th.nn.Linear(128, 512), th.nn.ReLU(),
                        th.nn.Linear(512, 128), th.nn.ReLU(),
                        th.nn.Linear(128, n_actions))
experiment = ExplorationDQNExperiment(params, model,
    ↪ learner=DoubleQLearner(model, params),
    ↪ ↪
    ↪ uncertainty=RNDUncertainty(scale=params['uncertainty_scale'], env=env))

# Re-executing this code-block picks up the experiment where you left off
try:
    experiment.run()
except KeyboardInterrupt:
    experiment.close()
experiment.plot_training()
```

```

RuntimeError                                Traceback (most recent call last)
Cell In [57], line 16
     14 # Re-executing this code-block picks up the experiment where you left off
     15 try:
--> 16     experiment.run()
     17 except KeyboardInterrupt:
     18     experiment.close()

Cell In [12], line 50, in QLearningExperiment.run(self)

```

```

48     self.env_steps.append(env_steps)
49 # Make one (or more) learning steps with the episode
---> 50 loss = self._learn_from_episode(episode)
51 if loss is not None: self.episode_losses.append(loss)
52 self.total_run_time += (datetime.now() - begin_time).total_seconds()

```

Cell In [33], line 11, in ExplorationDQNEExperiment.\_learn\_from\_episode(self, episode)

```

8 def _learn_from_episode(self, episode):
9     #-----
10     if self.intrinsic_reward:
---> 11         self.uncertainty.observe(episode['buffer']['next_states'])
12         estimated_uncertainty = self.
-> uncertainty(episode['buffer']['next_states']).unsqueeze(dim=1)
13         episode['buffer']['rewards'] += estimated_uncertainty

```

Cell In [55], line 28, in RNDUncertainty.observe(self, state, \*\*kwargs)

```

26 """ Observes state(s) and 'remembers' them using Random Network
-> Distillation"""
27 # YOUR CODE HERE
---> 28 error = self.error(state)
29 self.optimizer.zero_grad()
30 error.mean().backward()

```

Cell In [55], line 21, in RNDUncertainty.error(self, state)

```

19 if len(state.shape) == 1: state.unsqueeze(dim=0)
20 # YOUR CODE HERE: return the RND error
---> 21 target = self.psi_net(state)
22 prediction = self.si_net(state)
23 return self.criterion(prediction, target)

```

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/module.py:1518, in

```

-> Module._wrapped_call_impl(self, *args, **kwargs)
1516     return self._compiled_call_impl(*args, **kwargs) # type: ignore[misc]
1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

```

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/module.py:1527, in

```

-> Module._call_impl(self, *args, **kwargs)
1522 # If we don't have any hooks, we want to skip the rest of the logic in
1523 # this function, and just call forward.
1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
-> _forward_hooks or self._forward_pre_hooks
1525         or _global_backward_pre_hooks or _global_backward_hooks
1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
1529 try:
1530     result = None

```

```

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/container.py:215, in
↳ Sequential.forward(self, input)
    213 def forward(self, input):
    214     for module in self:
--> 215         input = module(input)
    216     return input

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/module.py:1518, in
↳ Module._wrapped_call_impl(self, *args, **kwargs)
    1516     return self._compiled_call_impl(*args, **kwargs) # type: ignore[misc]
    1517 else:
-> 1518     return self._call_impl(*args, **kwargs)

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/module.py:1527, in
↳ Module._call_impl(self, *args, **kwargs)
    1522 # If we don't have any hooks, we want to skip the rest of the logic in
    1523 # this function, and just call forward.
    1524 if not (self._backward_hooks or self._backward_pre_hooks or self.
↳ _forward_hooks or self._forward_pre_hooks
    1525         or _global_backward_pre_hooks or _global_backward_hooks
    1526         or _global_forward_hooks or _global_forward_pre_hooks):
-> 1527     return forward_call(*args, **kwargs)
    1529 try:
    1530     result = None

File ~/.local/lib/python3.10/site-packages/torch/nn/modules/linear.py:114, in
↳ Linear.forward(self, input)
    113 def forward(self, input: Tensor) -> Tensor:
--> 114     return F.linear(input, self.weight, self.bias)

RuntimeError: mat1 and mat2 shapes cannot be multiplied (200x6 and 2x1024)

```

## 1.6 A4.2: Nash equilibria

### A4.2: Nash equilibria

(a) Which inequalities of  $r^i$  hold in a state-less two-player zero-sum game?

In a state-less two-player zero-sum game, the Nash Equilibrium (NE) inequalities for each player  $i$  are:

$$r^i(a_i^*, a_{-i}^*) \geq r^i(a_i, a_{-i}^*) \quad \forall a_i \in A^i$$

Because the game is zero-sum, we have:

$$r^1(a_1, a_2) = -r^2(a_1, a_2) \quad \forall (a_1, a_2) \in A$$

This relationship simplifies the NE inequalities to:

$$\begin{aligned} r^1(a_1^*, a_2^*) &\geq r^1(a_1, a_2^*) \quad \forall a_1 \in A^1 \\ r^1(a_1^*, a_2^*) &\leq r^1(a_1^*, a_2) \quad \forall a_2 \in A^2 \end{aligned}$$

Since the actions  $a_1^*$  and  $a_2^*$  are the best responses to each other, no player can unilaterally change their action to increase their payoff.

**(b) Prove analytically that all NE in such a game yield exactly the same reward.**

To prove this, consider any two Nash equilibria  $(a_1^*, a_2^*)$  and  $(a_1^{**}, a_2^{**})$ . By definition, for both Nash equilibria, no player can improve their reward by unilaterally changing their action. Thus, we have:

$$\begin{aligned} r^1(a_1^*, a_2^*) &\geq r^1(a_1, a_2^*) \quad \forall a_1 \in A^1 \\ r^1(a_1^{**}, a_2^{**}) &\geq r^1(a_1^{**}, a_2) \quad \forall a_2 \in A^2 \end{aligned}$$

By substituting  $a_1 = a_1^{**}$  in the first inequality and  $a_2 = a_2^*$  in the second inequality, we get:

$$\begin{aligned} r^1(a_1^*, a_2^*) &\geq r^1(a_1^{**}, a_2^*) \\ r^1(a_1^{**}, a_2^{**}) &\geq r^1(a_1^{**}, a_2^*) \end{aligned}$$

By the transitivity of the inequality, we have:

$$r^1(a_1^*, a_2^*) \geq r^1(a_1^{**}, a_2^{**})$$

Now, let's consider the perspective of the second player. Since  $(a_1^*, a_2^*)$  is a Nash equilibrium, the second player cannot do better by deviating from  $a_2^*$  when player 1 plays  $a_1^*$ . Similarly, since  $(a_1^{**}, a_2^{**})$  is also a Nash equilibrium, the first player cannot do better by deviating from  $a_1^{**}$  when player 2 plays  $a_2^{**}$ . We can write:

$$-r^2(a_1^*, a_2^*) \geq -r^2(a_1^*, a_2^{**}) \quad \text{and} \quad -r^2(a_1^{**}, a_2^{**}) \geq -r^2(a_1^{**}, a_2^*)$$

By the zero-sum nature of the game, this implies:

$$r^1(a_1^*, a_2^*) \leq r^1(a_1^*, a_2^{**}) \quad \text{and} \quad r^1(a_1^{**}, a_2^{**}) \leq r^1(a_1^{**}, a_2^*)$$

Combining these with the previous inequalities, we have:

$$r^1(a_1^*, a_2^*) \geq r^1(a_1^{**}, a_2^{**}) \quad \text{and} \quad r^1(a_1^*, a_2^*) \leq r^1(a_1^{**}, a_2^{**})$$

Hence, it must be that:

$$r^1(a_1^*, a_2^*) = r^1(a_1^{**}, a_2^{**})$$



Therefore, we conclude that all Nash equilibria in a state-less two-player zero-sum game yield exactly the same reward for both players. This result is a fundamental characteristic of zero-sum games and reflects the nature of the equilibrium where no player has an incentive to deviate unilaterally from their strategy.

#### A4.3: Solving cyclic games

**(a) Simulate a cycle of max-min decisions to show the game does not have a Nash equilibrium.**

In the given payoff matrix, the best responses for each player lead to a cycle and do not converge to a stable strategy profile, indicating that no pure Nash equilibrium exists. The cycle is as follows:

- Player 1 chooses  $a_1$ , expecting player 2 to choose  $b_2$ .
- Player 2 then chooses  $b_1$  to maximize their payoff.
- Anticipating this, player 1 would want to switch to  $a_2$ .
- Player 2 would respond by choosing  $b_2$ .
- Player 1, seeing this, would switch back to  $a_1$ , and the cycle would repeat.

This cyclical pattern shows that the game does not possess a Nash equilibrium in pure strategies.

**(b) Compute the analytical expected reward  $E[r^1]$  for a pair of stochastic policies.**

The expected reward for player 1, given the stochastic policies  $\pi^*(i)$ , is computed as follows:

$$E[r^1] = \theta_1\theta_2 \cdot (+1) + \theta_1(1 - \theta_2) \cdot (-1) + (1 - \theta_1)\theta_2 \cdot (-1) + (1 - \theta_1)(1 - \theta_2) \cdot (+1)$$

Simplifying this expression yields:

$$E[r^1] = 1 - 2\theta_2(1 - \theta_1)$$

**(c) Compute the parameters  $\vartheta = [\vartheta_1, \vartheta_2]$  of a mixed Nash equilibrium for  $r^1$  and  $r^2$ .**

To find the mixed Nash equilibrium, we set the derivatives of  $E[r^1]$  with respect to  $\vartheta_1$  and  $\vartheta_2$  equal to zero:

For player 1:

$$\frac{\partial E[r^1]}{\partial \theta_1} = 0 \implies -2\theta_2 = 0 \implies \theta_2 = 0.5$$

For player 2:

$$\frac{\partial E[r^1]}{\partial \theta_2} = 0 \implies -2(1 - \theta_1) = 0 \implies \theta_1 = 0.5$$

Therefore, the mixed Nash equilibrium strategies for both players involve choosing each action with a probability of 0.5, leading to  $\vartheta = [0.5, 0.5]$ .

#### A4.4: Relative overgeneralization

In this problem, we analyze relative overgeneralization within a predator-prey game with two agents utilizing independent Q-learning. Here are the analytical solutions to the posed questions:

**(a) Determine analytically the value of the optimal policy when the first agent is 3 steps, and the second agent 5 steps away from the prey.**

The optimal policy would be for both agents to move towards the prey. If the reward for catching the prey is  $R_c$ , and the discount factor is  $\gamma$  within  $[0,1]$ , then the value of the optimal policy when the first agent is 3 steps away, and the second agent is 5 steps away, is given by:

$$V^* = \gamma^6 \cdot R_c$$

Since agent 1 has to wait to agent 2 to for him to arrive to catch the prey.

**(b) Determine the independent Q-value of agent 1 (Blue) in state  $s$  assuming both agents uniformly explore.**

If the agent gets stuck after leaving state  $s$ , the Q-value of any state-action pair will only depend on the immediate reward  $r$ . Therefore, for action  $a'_1$  from state  $s$ , the Q-value is simply:

$$q(s, a'_1; r) = r$$

since there are no future rewards due to the agent getting stuck.

**(c) Determine the range of punishments  $p$  that induce expectation relative overgeneralization.**

Relative overgeneralization occurs when the Q-value for the ‘catch’ action is less than that for ‘move left/right’, despite ‘catch’ having a higher immediate reward. For overgeneralization to occur, the Q-value for ‘catching’, considering the probability of both agents choosing ‘catch’, is:

$$q(s, C; r) = \frac{1}{3} \cdot r + \frac{2}{3} \cdot p$$

The Q-value for moving ‘left/right’, considering the probability of moving towards the prey together, is:

$$q(s, L/R; r; T) = \frac{1}{3} \cdot T + \frac{2}{3} \cdot p$$

For relative overgeneralization to occur, we need:

$$q(s, C; r) < q(s, L/R; r; T)$$

Substituting the Q-values, we get:

$$\frac{1}{3} \cdot r + \frac{2}{3} \cdot p < \frac{1}{3} \cdot T + \frac{2}{3} \cdot p$$

Simplifying, we find that relative overgeneralization occurs when:

$$r < T$$

regardless of the punishment  $p$ . Thus, relative overgeneralization is dictated by the immediate rewards  $r$  and  $T$ , and not by the punishment  $p$ .