Capstone_Report_V2

January 2, 2017

1 Machine Learning Engineer Nanodegree

1.1 Capstone Project

Matthias Wettstein
December 17th, 2016

2 Reinforcement Learning: Implementing and Comparing three Algorithms

2.1 I. Definition

2.1.1 Project Overview

Imagine living a complex world, where a myriad of different situations can occur, where the interaction of events are numberless and unpredictable. In this world, we human beings need to adapt, to learn, and to find a way to persue our development and secure self-preservation. In other words, we need to make use of the overwhelming amount of information around us to forge a strategy on how to survive or, at best, to thrive.

This is a non-trivial task, since we are forced to include ever new situations when we act. We might know similar situations from previous experience, but hardly exactly the same situation twice (just think of the film "Groundhog's Day").

The strategy is then to deduce, induce, take the next best fitting experience, and, most importantly, after having acted, to observe the consequences and *update our experience according to the consequences*. The consequences might not occur right away after an action taken, it may take a long while, and yet we must be able to track a consequence back to a certain action / decision we took beforehand.

What we humans do is applicable for machines too. Back in the days (and still), machines have been programmed determinically, i.e., for a finite set of situations, an appropriate action was hard coded. But machine can be brought to learn too, without a pre-defined plethora of "if - then" statements, but with the ability to react to changing environments.

This is the domain of *Reinforcement Learning (RL)*, which again is a domain of Machine Learning. In Reinforcement Learning, the agent interacts with its environment and is confronted with a chain of decision problems. This describes a *Markov Decision Process (MDP)*. After a row of actions, the agent obtains a reward (which can be negative), and adapts its strategy to maximize the reward. Reinforcement Learning embraces a range of methods on how to train an agent to act according to inputs available and a long-term objective.

This project is about to train an agent how to learn in such a chain of events. The agent for this project is a toy robot called the *Lunar Lander*, acting in an environment close to a toy moon surface.

Rocket trajectory optimization is a classic topic in Optimal Control. The Lunar Lander appeared as early as 1969 in the hype around Neil Armstrong and the race for the mooon, and starred several games since then. The object of the game is to land a lunar lander savely on the moon surface. In order to obtain this, the player needs to manoeuvre with propulsion properly, or else the lander crashes on the moon. Due to the weak gravitational forces the module is only slowly accelerated, and despite of the vacuum on the moon, the game is including a slowing air resistance.

In this project, there will be no player, but only the lander acting as a autonomous agent, taking in inputs, acting, and learning from experience.

This project is about training the Lunar Lander to learn landing on the toy moon. This serves as a toy example, ready to be extended to real-life tasks, and as a starter for my research on this area.

The toy environment is provided by Elon Musk's Open AI Gym. It made available the lunar lander module for training Reinforcement Algorithms. It uses a 2D engine in the background (Box2D) in order to simulate the physics in the vicinity of the moon surface. This will be the agent's environment throughout the project.

2.1.2 Problem Statement

The project is designed as a storyline, developing and comparing four algorithms:

- Algorithm 1 is the standard *Q Learning* algorithm. I hereby wrap up the lessons learned from the *Smartcab* section of the Machine Learning Engineer Nanodegree by applying them onto the Lunar Lander.
- Algorithm 2 is *Deep Q Learning (DQN)*, which I will - for reasons reflecting my own learning on the subject - is developed step-by-step away from standard Q Learning up to a fully fledged DQN.
- Finally, I will present two Policy Gradient algorithms
  - Monte-Carlo Policy Gradients
  - Action-Value Actor-Critic Policy Gradients

Before taking the step from algorithm 1 to algorithm 2 a basic Artifical Neural Network will be developed. This again is on account to reflect my learning on the subject.

The algorithms need to solve two paramount problems:

- Upon what experience / model the agent should take an action? Or: How should the agent determine which action might be beneficial given the circumstances?
- How should the agent update the experience / model after an action, in order to make it more accurate when assessing future circumstances?

The algorithms are presented in a standardized, comparable way, in order to make detection of differences and similarities among them easier.

They will be trained and benchmarked against one another by clearly defined metrics, with help of non-exhaustive parameter space exploration.

Since this is computationally extremely expensive, this project is not only about the algorithms, but also about on *where* to implement them. *High Performance Computing (HPC) on the Cloud* will therefore be discussed too in this project.

A significant share of detailed information is included as comments directly in the code.

In this report, I intend to provide a basis for further exploration by establishing Reinforcement Learning algorithms and Artificial Neural Networks (ANN). I expect to show first successes when training the agent, enabling to to continue improving the infrastructure in the right direction.

2.1.3 Metrics

The metrics for assessing the lander's performance are derived directly from the reward which the lander obtains during the landing process after performing actions. Open AI gym environment returns rewards for each time step, as described by the Open AI gym documentation:

> Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points. If lander moves away from landing pad it loses reward back. Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame. Solved is 200 points.

I thus focus on measuring the step rewards during the landing process. The main question thereby is: *Is the problem solved?* A solved problem will result in a time step reward of 200 points. Every algorithm's performance therefore will be measured by counting the number of *solved steps*. As a secondary measure, the number of *successful steps* (with a reward of >= 100) are measured. This is necessary since performing a solved step is non-trivial. In cases when a landing procedure does not produce a solution, the secondary measure serves as an indication of how well the algorithm performs, albeit without solving the environment.

2.2 II. Analysis

2.2.1 Data Exploration

A Markov Decision Process over a sequence of steps `t` is characterized by

- a set of states `s` an agent can be faced with
- a set of actions `a` an agent can take
- a set of new states `s'` an agent can end up after having taken an action
- a reward `r` an agent faces after having taken an action
- a transition probability `T/` policy, which indicates the agent which action to take, and which it has to learn

A full MDP is called an *Episode*. An episode is a period of steps `t` during which the agent is in play, i.e. has not yet crashed or succeeded to land properly.

An arbitrary number of subsequent Episodes is called an *Epoch*. It is the main block of training for an AI agent.

Open AI Gym is taking care of state space (called *observations*), the action space, and the rewards. What is left is to learn a well suited transition policy.

The step interaction between environment and agent boils down to the following single code line:

```
# Observe after action a_t
x_t, r_t, done, info = ENV.step(a_t)
```

where `ENV` is an instance of the Open AI Gym Lunar Lander environment (see below for details [Preparing an Environment]), `x_t` is a state `s'` obtained by action `a_t`, and `r_t` is the associated reward for the transition. `done` is an indication if the Episode has ended, and `info` will be neglected, since not relevant for the environment.

The result of above-mentioned step is e.g.:

```
array([-0.0051157 ,  0.93505154, -0.26596503, -0.19845303,
       0.00819082, 0.10553619,  0. ,  0. ])
       , -2.7344571275052219, False, {}
```

- The state space is a 8D vector of 8-digit floating numbers, whereas the coordinates w.r.t. the moon are the first two numbers in the state vector.
- The reward is described above-standing in the Section 2.5.1:
- done results in a boolean, `True` or `False`

The agent has four different actions at its disposition:

- Fire the main engine, which principally counteracts gravity
- Fire the left engine in order to correct insufficient angles
- Fire the right engine, likewise
- Do not fire at all

All four firing actions are discrete, i.e. fire engine at full throttle or do not fire at all. Again, after the documentation:

> According to Pontryagin's maximum principle it's optimal to fire engine full throttle or turn it off. That's the reason this environment is OK to have discreet actions (engine on or off).

The docs are indicating a few other points worth considering:

> Landing pad is always at coordinates (0,0). (...) Landing outside landing pad is possible. Fuel is infinite, so an agent can learn to fly and then land on its first attempt. Four discrete actions available: do nothing, fire left orientation engine, fire main engine, fire right orientation engine.

2.2.2 Exploratory Visualization

In order to provide a notion of the lunar lander movement and its reflection in state space and rewards, I tracked an example episode where the lander performs a random flight. The generating code is seen in **code snippet 02**.

Every 4th step, I took a snapshot of the rendered frame as well as the output generated by `ENV.step(ENV.action_space.sample())`, which embraces state, reward, and the termination flag.

Please find the visuals in the folder *Capstone_Figures*, subfolder *Visual_Assessment*: The screenshots are labeled *Random_Step_n*, where *n* embraces an interval `[0, 32]`.

The figure *Random_Stepwise_Assessment* displays the development over the steps:

At step 0, the lander is close to the upmost position on the vertical axis, reflected in a *vertical coordinate* close to 1. During the episode, it decreases the vertical position until its crash. The lander dashes against the moon surface and is thrown up a little. During the episode, the *vertical speed* is increasing monotonously. This is due to the inadequate use of the lower engine, which only is engaged at around steps 8, 12 and 16. Of course, the vertical speed is brought to around zero when the lander crashes.

Looking at the *horizontal coordinate*, one observes a steady and monotonously increasing movement from the horizontal center to the right. This is happening already in space, and is intensified as soon as the lander hits the surface, and slides down the hillside to the right. Responsible for this is the slightly increasing horizontal speed, caused by an almost constant engagement of the left engine. This engine is fired even during crash, intensifying the rightward movement of the lander on the ground.

Lastly, the firing pattern is reflected in the angle as well as the angular speed. The angle is increasing until the crash, when the surface - although also sloping to the right - is causing a correction. The angular speed increases up to step 16, where the right engine is fired and so causes a correction of the angle, rewarded with one of the highest rewards (or, least negative punishments). Of course, it is the correctional force of the moon surface which causes the only positive angular speed.

The lander crashes at about 32 steps, reflected in a reward of -100 and the *Done*-flag set from FALSE to TRUE. The rightward angle is responsible for the right leg getting ground contact. Due to the number of steps defined in the loop, the lander continues its movements after step 32, but since the episode is terminated, these movements are omitted here.

In contrast,

```
git clone https://github.com/openai/gym.git
python gym/envs/box2d/lunar_lander.py
```

shows the agent in heuristic action (see *Heuristic_Snapshot* in the above-mentioned folder). The successful landing is characterized by a few key elements:

- The lander lets itself drop until below the middle of the vertical axis, i.e. does not fire the lower engine at all. By doing so, it spares the negative rewards of -.3 for each time firing the lower engine. In contrast, it heavily engages the lower engine to avoid crashing on the surface afterwards.

- The lander does not allow an angle above a certain absolute level by cautiously firing either the left or right engine. It never needs to counterfire heavily and thus never causes the balancing movement. In plus, the lander stays in the middle of the horizontal frame and lands between the flags.

2.2.3 Algorithms and Techniques

Understanding the Gist of Artificial Neural Networks Before starting to implement a Q estimator, I want to understand how an ANN works. According to Andrew Trask's manual, I create a network with one hidden layer, and the output layer, both sigmoid activated, which returns probablities for each of the outputs.

I use a sigmoid activation function due to the easy calculation of the derivative and the widespread application in neural nets.

In the procedure, the magic happens with `layer_2_weighted_loss`: Each output loss is multiplied by the slope / gradient of the predicted value on the sigmoid curve.

In order to get a feeling for changes invoked by changing parametrization, I will alter the learning rate `ALPHA` as well as the number of neurons in the hidden layer `NUM_HIDDEN_NEURON` within an arbitrary range.

Code snippet 03 shows the implementation of such a toy network.

When looking at the output, it is clearly visible that the parametrization of the network has an impaction upon the network performance. Some configurations work well, like e.g. a hidden size of 8 with an `ALPHA` of 100, whereas as of a certain hidden size, the results stay at a bad level.

Preparing the Environment The environment for the whole project is created once. The initialization procedure is created in a way

- I create an instance (ENV) of the current Lunar Lander environment:

  ```
  ENV = gym.make("LunarLander-v2")
  ```

- The input dimension NUM_INPUT is obtained by resetting the environment:

  ```
  NUM_INPUT = ENV.reset().shape[0]
  ```

- Thee actions are obtained by the environment method action_space:

  ```
  NUM_ACTION = ENV.action_space.n
  ```

- The action space then is simply an enumeration:

  ```
  ACTIONS = np.arange(0, NUM_ACTION)
  ```

Code snippet 01 shows the implementation of the environment initialization.

Environment Interaction The main building block describing the interaction of the agent with the environment (during one epoch)

- contains more than zero episodes.
- records statistics (stats) along the way:
  - The reward sum per episode

- The number of times the lander did a successful step during the episode (with a reward greater or equal 100)
- The number of times a step was solved (with a reward greater or equal 200)
- The number of times the agent has revisited states, compared to the size of the Q table.

It looks as follows:

```
# Define a training epoch
def training_epoch():
    # Play through a pre-defined number of episodes
    for episode in range(NUM_EPISODE):
        /* Initialize observations randomly
        and set the episode state to not done */
        x_t = ENV.reset()
        done = False
        /* Start an episode,
        let it run as long as it is not done
        each step is denoted with `t` */
        for step in count()
            # Take an action based upon experience / a suitable model
            ## [X]
            /* Observe the state space, the reward
            and the episode state after action a_t */
            x_t, r_t, done, info = ENV.step(a_t)
            # Train the model
            ## [Y]
            # Exit episode after crash or deadlock
            if done or step > MAX_NUM_STEPS: break
```

[X] and [Y] are denoting the crucial moments of choosing an action and updating the experience. `NUM_EPISODE` denotes the number of episodes to play during an epoch. `MAX_NUM_STEPS` denotes the maximum allowed steps per episode to avoid deadlock.

(Sidenote: this code and some of the following are slightly altered w.r.t. the code in the code section. This is due to formatting / page size reasons in markdown.)

2.2.4 Benchmark

The benchmark metrics for comparing the different algorithms can be directly derived from the reward obtained during the course of an epoch and is described in Section 2.5.1.

Throughout the project, sucesses and solutions are tracked for each episode, for each epoch.

An epoch will entail 150 episodes (`NUM_EPISODES`) each. The lower benchmark is set by the Q learning algorithm, followed by epochs which implement step-by-step to a fully fledged Deep Q learning algorithm. The latter then sets the higher benchmark for the following Policy Gradient algorithms.

2.3 III. Methodology

2.3.1 Data Preprocessing

The input data / state space provided by Open AI Gym is already preprocessed. It is a 8D vector, which means that a convolution is not needed.

However, I argue that it might make sense to let the agent take into account more than just the current state, when taking an action. As such, the "memory" of the agent will extend to `n` past state spaces, which requires stacking of state spaces. It becomes a parameter (`STEP_MEM`) which can be tuned during experimentation.

The random initialization of observations becomes

```
/* Pile up STEP_MEM times the same init observation,
in order to be consistent with the model input */
s_t = np.tile(x_t, STEP_MEM)
```

After having observed a new state space `x_t`, I stack the new state space on top of the state space queue:

```
# Create state at t1: Append x observations, throw away the earliest
s_t1 = np.concatenate((x_t, s_t[:(STEP_MEM-1) * NUM_INPUT,]), axis=0)
```

2.3.2 Implementation

The Starting Point: Evaluating Bellman Equations From Data (Q-Learning) Q-Learning is a variant of *Temporal Difference-Learning (TD Learning)*, where the agent assesses the benefit of an action, instead of the state it is in. An agent performing according to a TD-Learning algorithm does wait until it gets the final reward, but adjusts after every action on basis of an estimated expected reward.

Q learning is about revisiting states. The agent is given a *memory* in form of a lookup table, where it stores states `s`, and updates the punishment or reward for each action it has taken.

The number of possible combinations of the relevant inputs which constitute a state is called the state space. Whenever the state space is sufficiently small, the agent - at step `t` - might discover that it has already been in a specific state `s`. It therefore has made an experience by taking an action.

If the agent should take advantage of its "memory" (possibly a dictionary of dictionaries, main keys being the states, values being the actions / keys and their values), it should use previous experiences and update them with new experiences:

- It looks up the expected lifetime rewards per each possible action in `s` (a.k.a. action-value function, `q` values), selects the maximum `q` value, and executes the chosen action.
- It obtains fresh evidence about the consequence of a specific action in a specific state: It knows the initial state `s_t`, the selected action `a_t`, the reward `r_t`, and the new state this all lead to, `s_t1`.
- This knowledge it now uses to calculate a new `q` value for `s_t` by taking the observed reward `r_t`, and adding to it the discounted maximum `q` value for `s_t1`. The difference to the old `q` value is the new `q` value for action `a_t` in state `s_t`.

It then chooses the action with the highest associated `q` value (<=> the maximum of the `q` function, or: the highest expected lifetime reward at step `t`) and moves onward another step.

Why Q-Learning Often Does Not Work: Exploding State Spaces The crux about state spaces is, that they can get very large very quickly. Just think of a small number of inputs, each input being a floating number with 4 digits. Even this small setting is creating a large amount of combinations: the state space explodes. Revisiting states will become very unlikely, even with a large amount of trials. As a consequence, learning is slow, or even not happening.

Code snippet 04 illustrates the point by setting up a Q Learning algorithm.

One clearly can see that a simple table lookup is not adequate anymore. There is most often simply nothing to look up, since the state space is so huge.

What to Do? Do Not Update the Q Function, But the Q Function Estimator: Deep Q Learning With the fast growth of the state space for the Lunar Lander game in mind, I intend to replace the table lookup by a function approximator. An Artifical Neural Net (ANN) will estimate the `q` function for the state `s` at step `t`.

Once the agent has performed an action, it will know the reward `r_t` and the subsequent state `s_t1`. Based on this, it is able to update its memory. But this time, it does not update the `q` function directly. Instead, it is updating the ANN, which means that is updating the weights used in the ANN:

- At time `t`, the agent already knows the *estimation* of the `q` function for state `s_t`: It used it to pick an action `a_t` accordingly.
- After action `a_t`, the agent knows `s_t`, `a_t`, `r_t` and `s_t1`. This allows us to update the `q` function, *but only for the action taken*: The discounted expected lifetime reward is added to `r_t`. In other words: The ANN estimates the `q` function for state `s_t1`.
- For the action taken, the agent now updates the `q` value. All the other actions are not performed, the agent does not know about the reward, or a subsequent state `s_t1`. So, the agent cannot learn for those. This updated `q` function is the *target*.

The error, which is the difference between the estimation and the target, is backpropagated through the network, such that the weights are updated.

Next time the agent lets estimate the `q` function for another state `s`, it will have updated weights at hand.

Deep Q Learning from Single Current Observations In order to implement the Q estimator, the training epoch function described in **code snippet 04** needs to be changed. The lookup table `Q_TABLE` and the table's `q` value initialization `Q_INIT` are both not needed anymore. In plus, I do not need the learning rate `ALPHA` anymore.

The core routine implemented for the function estimate is an ANN block. For this task I chose Keras and its `Sequential` model architecture. For the Deep Q Learning part of the project, the sequential model consists of

- one fully-connected hidden layer (Keras terminology is `Dense`) plus an non-linear `relu` Activation,
- the output layer with softmax activation, which produces probabilities for each of the possible actions given a certain state input. Here, I am not entirely sure if using probabilities in stead of a regression output is against some rules in Deep Q Learning. I tried also a linear activation and thereby found out that the probabilities are working very nicely. (Comments on this are very welcome.)

This set-up is found after some preliminary experimentation in the parameter space, including visual checks of the landing procedure.

Code snippet 05 shows the exact implementation of a DQN including the ANN model block. Some highlights to special portions of the basic DQN:

- For action selection, Q is estimated by the ANN based upon the current state space `s_t`, using the Keras function `model.predict`:

```
# Estimate q for each action at s_t
q = model.predict(s_t[np.newaxis])[0]
```

- For the model update, Q is estimated by the ANN based upon the subsequent state space `s_t+1`, using the Keras function `model.predict`:

```
# Estimate q for each action at s_t1 (again a forward pass)
Q_sa = model.predict(s_t1[np.newaxis])[0]
```

- For the model update, standard Q learning acts as the target, upon which (supervised learning) the ANN is trained using the Keras function `model.fit` (Note that the target is set to the reward in case the episode is terminated):

```
targets = q
targets[a_t] = r_t + GAMMA * np.max(Q_sa) if not done else r_t
model.fit(s_t[np.newaxis], targets[np.newaxis], verbose=0)
```

  where `GAMMA` is the discount factor for future rewards.

Policy Gradients

A Methododical Overview So far, I have performed Value-based Reinforcement Learning by trying to find the maximum value over all possible actions in a given state. The policy thereby was generated from the value function. Another approach would be to directly parametrize the policy.

Both approaches have pros and cons. Policy-based RL has better convergence properties. It does not need a maximum value, which is of great interest in continuous action spaces. On the other hand, policy-based RL often converges towards local in stead of global optima, and they suffer from high variance.

Taking An Action The agent faces the decision of taking one of the available actions at every step `t`. When applying a policy gradient model, the agent estimates which move is most likely to lead to a good end / a high reward: It assigns probabilities to every action. In order to accomplish this, a first ANN block is needed, which outputs probabilities / uses a softmax at the output layer. It is called the policy estimator.

After the estimation, it samples uniformly from the probability distribution in order to get an action, and observes / learns from the consequences of the action.

Evaluating An Action An example for Policy-based RL is the Monte-Carlo Policy Gradient algorithm, which is also known as REINFORCE. It is first presented here.

The basic principle of the REINFORCE algorithm is to sample from the action-value function q by using the collected returns of a full episode: After each episode, the algorithm runs through each time step again and calculates the target (a value function), which is the total discounted reward after a specific step (the expected return is sampled directly from the episode), representing a sample of the action-value function. It multiplies the target with the score function at each time step to get the gradient. The score function is the negative log probability of the chosen action.

During the trajectory (the time steps of an episode), the transitions are collected for the later adaption of the policy after the episode.

This algorithm suffers from high variance, because the sampled rewards can be very different from one episode to another. Therefore this algorithm is usually used with a baseline function B(s) substracted from the target. Over all states s in the state space S, the expectation of the policy gradient is still 0, so expectations do not change.

Instead of the target, the score is multiplied with the advantage function, which is simply the target minus the baseline. A suitable baseline is the state value function, which is a value function estimate. For this, I need a second ANN block, the so called value estimator, outputting a regression estimate by applying a mean squared error loss function.

Intuitively, the advantage captures how much better than expected the action at time step t was doing.

The last presented algorithm, the Action-Value (or Q) Actor-Critic algorithm, is again a Temporal-Difference-based algorithm.

Actor-critic are a mix between value based and policy based frameworks. The actor is taking the action, the critic is evaluating the action. In contrast to REINFORCE, the latter is done during the episode. This bootstrapping from the value function leads to lower variance, but also to higher bias, compared with REINFORCE.

The target / value function for Q Actor-Critic algorithms is the estimated cumulated reward, which consists of the actual reward at s_t plus a discounted estimated reward for the action at `s_t1`: `target = r_t + GAMMA * Q_sa`. Again, the value function estimate is subtracted from the target. The resulting advantage is then multiplied with the score function.

Implementation Policy gradient methods need two kinds of different estimators:

- The value estimator
- The policy estimator

For this purpose, I recreate the existing `_create_network` function. I will use a different set of initializations as well as hidden layer activations for the two models. The policy estimator has a softmax activation for the output layer, the value estimator a linear activation.

In order to the model the non-standard gradients in Keras, a helper function is needed. Getting the gradients subsequently is relying directly on TensorFlow, applying `tf.gradients()`.

Code snippet 06 is showing the new `create_network` function as well as the gradient helper function.

The Monte Carlo Policy Gradient method is implemented in **code snippet 07**. Note that the agent learns only after having finished an episode, which results in low bias and high variance.

The Action-Value (Q) Actor-Critic is implemented in **code snippet 08**. In contrary to the Monte Carlo Policy Gradient implementation, the algorithm bootstraps from the value function during the episode, which results in higher bias, but lower variance.

2.3.3 Refinement

***Epsilon Greedy* Deep Q Learning from Single Observations** At this point, it is time to refine the Deep Q Network that has been developed so far (Deep Q Learning from Single Current Observations). This is about tackling the issue of getting stuck in local minima. As a remedy, Deep Q Learning makes use the so called *epsilon greedy* action selection policy. It allows for a random move with probability `epsilon`, and by that introduces the notion of exploration (random moves) vs. exploitation (act on estimation of the q function).

Exploration will reduce the probability of getting stuck in local minima, which are *not* reflecting the best action given a certain experience level of the agen. It introduces random ideas.

On the other hand, the agent needs to train and get experience with his selected moves. It needs evidence that one decision was (not) the right one, and to update its decicion finding process (the weights of the ANN. It only gets it by acting according to its own decisions undisturbed by random inputs. This is where *exploitation* comes in.

In RL, usually `epsilon` decreases over a certain exploration period. This reflects the idea that the agent will start with many random moves to fathom the environment by just observing. With time and growing experience, it will decrease the share of random moves, since it feels more confident in its own decisions.

Following the custom of allowing random moves at a linearly decreasing exploration rate during the exploration period. Following Deep Mind: Playing Atari with Deep Reinforcement Learning (p.6), I define a interval between the maximum and minimum `epsilon` allowed (`EPSILON_RANGE`). The exploration period (`NUM_EXPLORATION_STEP`) is set to roughly 1/10 of the total number of steps. Assuming an average 100 steps per episode lets `epsilon` decrease until Episode 6. If I increase the average to 150, I end up with the desired 10 episodes until minimum `epsilon`.

During training, the agent will run on the minimum `epsilon` constantly.

Code snippet 09 shows the exact implementation of a DQN including an epsilon greedy policy. Some highlights on the changes in the code:

- Epsilon is annealed linearly over exploration period:

```
epsilon = max(epsilon - (EPSILON_RANGE[0] - EPSILON_RANGE[1]) / \
    NUM_EXPLORATION_STEP, EPSILON_RANGE[1])
```

- Exploitation is done with probability `1-epsilon` ("epsilon greedy" policy):

```
# Take action with highest estimated reward
a_t = np.argmax(q) if np.random.random() > epsilon \
    else np.random.choice(ACTIONS, 1)[0]
```

Deep Q Learning *from Stored Experiences* Deep Mind: Playing Atari with Deep Reinforcement Learning claims that

> learning directly from consecutive samples is inefficient, due to the strong correlations between the samples

(p.5). Instead, the trick is to learn from a memory storage - which will be called Experience Replay Memory (`ERM`) here - in batches. The `ERM` is set up once per epoch. It is fed at each step with fresh transition evidence, and it is cropped again when its maximum size is reached.

The `ERM` is the main database of the agent: It is the place where it

- stores states and its experiences with the states (transitions `s_t`, `a_t`, `r_t`, and `s_t1`)
- recalls on the memory, collects a memory sample, trains on the sample, and updates the `q` function estimator.

`ERM_SIZE` is setting the size of the experience replay memory. Following the recommendation of Deep Mind: Playing Atari with Deep Reinforcement Learning (p.6), I set it equal to the number of exploration steps. `BATCH_SIZE` denotes the size of the transition sample which is drawn uniformly without replacement from the `ERM` at each step. Again, its size is following the recommendations of Deep Mind: Playing Atari with Deep Reinforcement Learning (p.6).

Each memory update operates on a batch size of 32 randomly samples experiences. From what can be read, larger batch sizes are increasing the probability of the algorithm getting caught in local minima, so I stick with the DeepMind indication here, and do not test out this parameter.

Code snippet 10 shows the exact implementation of a DQN including an epsilon greedy policy and learning from stored experiences.

The new features of the code:

- Start exploration only when the `ERM` is filled up with transitions completely:

```
# Linarily anneal random exploration rate epsilon over exploration period
if len(ERM) < ERM_SIZE: epsilon = EPSILON_RANGE[0]
else: epsilon = max(epsilon - (EPSILON_RANGE[0] - EPSILON_RANGE[1]) / \
    NUM_EXPLORATION_STEP, EPSILON_RANGE[1])
```

- Store transitions in `ERM`

```
ERM.append((s_t, a_t, r_t, s_t1))
```

- Randomly draw a minibatch out of the transitions available in the `ERM` of size `BATCH_SIZE`:

```
minibatch = np.array([ ERM[i] \
    for i in np.random.choice(np.arange(0, len(ERM)), \
    min(len(ERM), BATCH_SIZE)) ])
```

- Compute targets/references for each transition in minibatch

```
inputs = deque(); targets = deque()
for m in minibatch:
    inputs.append(m[0]) # Append s_t of batch transition m to inputs
    # Estimate rewards for each action (targets), at s_t
    m_q = model.predict(m[0][np.newaxis])[0]
    # Estimate rewards for each action (targets), at s_t1
    m_Q_sa = model.predict(m[3][np.newaxis])[0]
    m_targets = m_q
    m_targets[m[1]] = m[2] + GAMMA * np.max(m_Q_sa)
    # Append target of batch transition m to targets
    targets.append(m_targets)
```

- Train the model by backpropagating the errors and update weights

```
model.train_on_batch(np.array(inputs), np.array(targets))
```

2.4 IV. Results

2.4.1 Model Evaluation and Validation

Evaluation Infrastructure First evaluation steps are done on a MacBook Air from early 2014. For tasks not relying directly on TensorFlow, I used an on-site Windows Server 2008 R2 Enterprise with a quad core Intel Xeon CPU E5-2680 @ 2.7 GHz and 32 GB RAM. When performing HPC on AWS EC2, I started with Ubuntu 16.04 t2.micro free tier instances, then switched to Ubuntu 16.04 c4.8xlarge instances. The insufficient speed forced me to move on to bare metal GPU instances on AWS. I finally ended up using pre-configured Bitfusion Ubuntu 14 TensorFlow instances available on Amazon Marketplace, which spared me hours of setting up software and drivers, but still needed some tweaking. The g-version instances as well as the p2.xlarge were not performing as needed, so I went for the p2.8xlarge, which for ca. 5'000 policy gradient episodes needs around 24 hours.

Processing the model evaluations took more than a month, included still too slow computation, set-ups, and trial and error with the test design.

Code snippet 11 contains the shell script to set up the remote infrastructure on AWS EC2, as well as the helper function to produce and obtain the epoch statistics.

Evaluation Design Model evaluation is done along the following generic steps:

- Generate a model id for identification. Every id consists of the exact parametrization of the model

```
EPOCH = '_'.join([repr(STEP_MEM), str(NUM_HIDDEN_NEURON) \
    , INITIALIZATION, ACTIVATION])
```

- Train the model, i.e. call the function containing the model epoch, and save the output to variable

```
stats = train_ql(render=True)
```

- Calculate the summary statistics for each epoch. `highest_reward` is of an informative nature, `successful_steps` and `solved_steps` are the metrics for model benchmarking.

```
highest_reward = max([ v[0] for v in stats.values() ])
successful_steps = sum([ v[1] for v in stats.values() ])
solved_steps = sum([ v[2] for v in stats.values() ])
```

- Visualize the model performance, episode by episode

```
stdout.write("\rEpoch {}, Maximum Reward {}, \
    Successful Episodes {}, Solved Episodes {}".format( \
    EPOCH, highest_reward, successful_steps, solved_steps))
```

The quality of the algorithms is assessed with a handy 150 episodes each during the development phase away from Q Learning towards Deep Q Learning.

Nothing is changed regarding the algorithm parametrization:

- `STEP_MEM = 1`
- `NUM_HIDDEN_NEURON = 200`
- `INITIALIZATION ='glorot_uniform'`
- `ACTIVATION = 'relu'`
- `GAMMA = 0.99`

Evaluating the Phase from Q Learning (lower benchmark) to Deep Q Learning **Code snippet 12** with model `train_ql` implements Q Learning:

```
Q Table size 14005, # Revisited States 0
Epoch Q-Learning, Maximum Reward -0.780001726748,
    Successful Episodes 3, Solved Episodes 0
```

The state space has swollen up to 15k states during 150 episodes. The lower benchmark is set. Not a single state is revisited by the procedure. Q Learning is not able to solve the environment, but produces three successes out of 150 in the trial.

Code snippet 12 with model `dqn_1` implements Deep Q Learning from Single Current Observations:

```
Epoch 10_1_200_glorot_uniform_relu_0.99, Maximum Reward 177.884551583,
    Successful Episodes 5, Solved Episodes 0
```

No solution so far, but a much, much nicer landing behaviour compared to Q Learning. The frantic, purpose-less behaviour is gone most of the time, or is vanishing quickly within a few episodes only.

- Most of the time, I observe an extensive swinging movement. The agent tries to counter-act skewed positions by engaging the lateral engines, and overdoes it. Then, it corrects again, and again, and from all these corrections forgets to fire against the moon's gravity, and then crashes into the surface.
- There are plenty of times I can see the agent trapped into a locally optimal policy. For example, it stays on the ground, engaging left and right engine forever, perfectly stable, but not reaching the ultimate goal. Or a setting where left and right engines are engaged, but the lower engine does not fire at all, over long episodes.

What is to be learned from this basic implementation?

- Does the weight initializations make sense? What are the alternatives?
- Does the nonlinear activation function for the hidden layer make sense? What are the alternatives?
- What about the number of neurons in the hidden layer?
- What is the optimal step memory `STEP_MEM`: Does it make sense to let the agent know only its current state, or shall it take into consideration also some of the states before? If yes, how far back should it remember?

Code snippet 12 with model `dqn_2` implements *Epsilon Greedy* Deep Q Learning from Single Observations:

```
Epoch 10_1_200_glorot_uniform_relu_0.99, Maximum Reward 189.44243222,
    Successful Episodes 7, Solved Episodes 0
```

Code snippet 12 with model `dqn_3` implements Deep Q Learning *from Stored Experiences* :

```
Epoch 10_1_200_glorot_uniform_relu_0.99, Maximum Reward 153.51536308,
    Successful Episodes 10, Solved Episodes 0
```

A notion crosses my mind, during experimenting around with the minibatch: Why not perform a prediction for each sample transition of the `ERM`, during iterating through the batch?

Probably, because it has the disadvantage that I predict with the knowledge available at step `t`. This might be faulty, and the faulty prediction stays as target reference in the batch, and is used to compare the loss for the taken action between the prediction at timestep `t` and the potenitially long ago target estimation. This will bias the learning process significantly. Thus, `model.train_on_batch` is done only after the batch has been created and computed.

2.4.2 Justification

Brute Force Parameter Space Exploration This section covers the experimental parameter space exploration for the three algorithms Deep Q Learning, Monte Carlo Policy Gradients, and Action-Value (Q) Actor-Critic. The Deep Q Learning thereby is algorithm *dqn_3*, which finalized the development phase of Deep Q Learning.

The goal is to identify well-working parameter settings.

The parameter space will entail ranges of `NUM_HIDDEN_NEURON`, and `STEP_MEM`. `EPSILON_RANGE` for Deep Q Learning starts at 20% instead of 10%, in order to allow more exploration than before.

The original idea was to include more parameters into the routine:

- Steps per Action (`SPA`) determines how many steps pass before a fresh action is taken. Deep Mind - Playing Atari with Deep Reinforcement Learning suggests 4 skipping 4 time steps before conducting an action (p.6): `if step % SPA == 0: a_t = np.argmax(q) if np.random.random() > epsilon else np.random.choice(ACTIONS, 1)[0]`.
- Reward Clipping (`R_CLIP`) is also an idea taken from Deep Mind - Playing Atari with Deep Reinforcement Learning (p.6), where it was implemented in order to make rewards comparable across the different games: `if R_CLIP and r_t != 0: r_t = abs(r_t) / r_t`.
- Looping over a range of `GAMMA` candidates.
- Looping over a multitude of `EPSILON_RANGE`.
- Looping over all possible Keras activations and initializations.

In order to stay within a feasible time frame w.r.t. computation and a exploding parameter space, I decided to exclude a test of the above-standing parameters.

Each parameter setting is tested within an epoch consisting of 150 episodes. Parameter settings with at least one successful episode are written down to `.json` files.

Code snippet 13 implements the brute force parameter exploration for Deep Q Learning. This exploration entails 42 epochs over 150 episodes each. Reading the cumulative statistics from the `DQN_Stats.json`:

```
Solved!: DQN_Stats.json | 6_400_glorot_uniform_relu_0.99 |
    [124.32605681580637, 11, 1]

Solved!: DQN_Stats.json | 6_200_glorot_uniform_relu_0.99 |
    [210.01669881507274, 20, 1]

Best trial without solved: DQN_Stats.json |
    (u'2_200_glorot_uniform_relu_0.99', 200.3079725916203, 25)
```

Obviously, an operational memory of 6 frames leads to two solved steps. The hidden layer size of 200 paramounts with 210 cumulated reward. For Deep Q Learning, this is the way to go forward.

Code snippet 14 implements step 1 of the brute force parameter exploration for Policy Gradient methods. The 205'800 epochs - with each 150 episodes for each model resulting in a total of 30'870'000 episodes - is far too ambitious in terms of the computational capacities at hand.

In fact, a first trial with this full set up had to be aborted after very long runtimes on one single GPU / CPU. The following non-exhaustive results are an indication in which direction to tune at least some of the parameters (`MCPG_Stats.json`, `QAC_Stats.json`):

```
Best trial without solved: MCPG_Stats.json |
    (u'5_200_normal_normal_softplus_softsign', -7.553130756059716, 5)

Best trial without solved: QAC_Stats.json |
    (u'1_200_he_uniform_he_uniform_softplus_softplus', -8.638336724713128, 7)
```

For the consecutive exploration, I decided to hold the following parameter subspace:

- Value estimator initialization `normal`
- Policy estimator intialization `normal`
- Value estimator activation function `softplus`
- Policy estimator activation function `softsign`

for Monte Carlo Policy Gradients, and

- Value estimator initialization `he_uniform`
- Policy estimator intialization `he_uniform`
- Value estimator activation function `softplus`
- Policy estimator activation function `softplus`

for Action-Value (Q) Actor-Critic, based on the available *non-exhaustive* activation and initialization exploration resulting in 5 resp. 7 successful episodes out of 150.

Code snippet 15 implements step 2 of the brute force parameter exploration for the Action-Value Actor-Critic algorithm. Given above-mentioned initializations and activations, combinations of hidden neuron size `NUM_HIDDEN_NEURON` and operational memory `STEP_MEM` are played through. The parameter space shinks 48 combinations / epochs or ambitious 7'200 epsiodes.

Given the computational constraints at hand, I will not follow up with the Monte-Carlo Policy Gradient, since it produced sightly weaker results, and leave this excercise to a subsequent study.

For the Action-Value Actor-Critic algorithm, the results of this step show that it has not a single solved step and delivers an altogether moderate performance:

```
Best trial without solved: QAC_Stats_V2.json |
    (u'1_600_he_uniform_he_uniform_softplus_softplus',
    -21.99401980452734, 3)
```

The results indicates that a `STEP_MEM` of 1 and a `NUM_HIDDEN_NEURON` of 600 perform best.

2.5 V. Conclusion

2.5.1 Free-Form Visualization

In this section, I would like to highlight the performance of the Action-Value (Q) Action-Critic algorithm as well as Deep Q Learning algorithm.

Please find the visuals in the folder *Capstone_Figures*, subfolder *Free_Form_Visualization*: The screenshots are labeled according to the originating model (*DQN* or *QAC*). **Code snippet 16** sets up long episodes for the algoritm.

Metrics I will conduct one epoch each algorithm and record

- the summed up reward per episode
- the number of solved and successful steps per episode
- the episode length
- the average reward per episode

over the range of 1'000 episodes each, in order to check on the learning development of the algorithms. For this reason, both algorithms are staffed with a forth statistic, the episode length. Deep Learning is starting from epsilon 1, reflecting the longer period of random moves now allowed by the extended epoch length.

Parametrization The parametrizations are the ones found during brute force exploration:

- Value estimator initialization `he_uniform`
- Policy estimator intialization `he_uniform`
- Value estimator activation function `softplus`
- Policy estimator activation function `softplus`
- Operational memory of 1 time step
- Hidden neuron size of 600

for Q Actor-Critic, and

- Initialization `glorot_uniform`
- Activation function `relu`
- Operational memory of 6 time steps
- Hidden neuron size of 200

for Deep Q Learning.

Discussion I could not reproduce the solved steps for Deep Q Learning, nor the Actor-Critic algorithm produce a solved step. In terms of successful steps, the former outperforms the latter with 131 vs. 85 steps respectively. Looking at the figures, it becomes obvious that both algorithms stationary w.r.t. rewards, be it summed up or in average.

- Implemented Deep Q Learning:
  - Average episode rewards stay at around -5 , and a relatively high variation. It seems as if the algorithm produces average rewards which oscillate up and down over the epochs.
  - Summed up episode rewards also seem to oscillate over epochs, showing extreme peaks (up to almost 200, down to -1'400), and averaging at -600 overall.
  - Successful steps are performed steadily over the epoch. There are hardly any periods during learning where the algorithm produces significantly less or more successes.
- Implemented Q Actor-Critic
  - Average episode rewards stay at around -3.5 very stably, having a smaller variation. The peaks tend to go to the positive side.
  - Summed up episode rewards are averaging at -250 overall, showing again a smaller variation (0 to -500).
  - Successful steps are less steady than with Deep Q Learning. There seem to be periods where the algorithm learns significantly faster or slower.

Overall, the Deep Q Learning algorithm performs on a lower level, producing more variation and even oscillation. In contrary to the Q Actor-Critic algorithm, which in average tends to perform better, but never leaves mediocrity, the Deep Q Learning algorithm is able to peak in performance from time to time, and producing successful, or - as seen before - even solved steps.

2.5.2 Reflection

This project reflects the development and the learning curve which I had to undergo:

I have presented a basic implementation of an Artificial Neural Network, and I have gone the way from *Smartcab* Q Learning to Deep Q Learning by applying step-by-step extensions to the original algorithm. I afterwards presented two more Reinforcement Learning algorithms and let them undergo a testing procedure for comparison reasons.

Having a more hands-on approach in mind, it was challenging to understand the concepts and to implement them in a generic, as-simple-as-possible way. Without the many resources on the net this would not have been possible. I belief that toy code is the best way to learn, and luckily there were others sharing their experience (see below-standing credits for details).

It was extremely insightful to bring different concepts under one generic methological hood. Looking at the differences in different portions of the code made many concept clearer.

I now have a clear indication about the nature of the algorithms and Artifical Neural Networks in general. I know some of the tools to use (Keras and TensorFlow). I have a clear indication that hardware constraints are a paramount topic with Reinforcement Learning, and that a well developed test design is worth a lot.

Testing evolvement and design was not a straight line, but trial-and-error, resulting in a sub-optimal testing set-up. It was clearly a matter of underrating the vast parameter space associated especially with Artificial Neural Networks, which resulted in a considerable amount of time spent for testing as well.

On the minus side, a Box2D dependency deserves a short side note, being a major productivity slowdown with its erratic installation behaviour. On the plus side, a thanks to the helpful *Bitfusion* guys.

The final solutions do not quite match my initial expectations. Of course, I had an agent with a steeper learning curve in mind. But since this project was intended only to be a starting point to fix the basics, and Deep Q Learning partially went into the right direction.

There are many questionmarks left:

- Are the ANN models accurate?
- Are the RL algorithms accurate?
- Is a deadlock prevention really part of the algorithm, or should the agent be allowed to try infinitely many times?
- Not all questions raised when testing the basic Deep Q Learning algorithm have been answered: Which are the parameters to put emphasis upon when testing?

Regarding these questions, I am of course grateful for inputs.

There is no doubt that Reinforcement Learning algorithms are an extremely valuable way to go when it comes to self-learning agents / robots in games or in reality.

2.5.3 Improvement

My final benchmark is easily surpassed by other existing solutions.

As mentioned in the *Reflection* section, Improvements start with scrutinizing architectures, and continue with adapt the test design and hardware allocation.

There are other RL algorithms around, which also could be taken into consideration. Having not presented them in this project was a matter of choice. Intellegible, hands-on instructions exist for a plethora of other algorithms.

After having established true learning curves for the agent using stabilized models and algorithms, one can think of saving the weights and make available those pre-trained models (keywords `model.save_weights()` and `model.load_weights()` with Keras), or going to the Open AI Gym competition.

Finally, it is a task to generalize the algorithms to a multitude of different MDPs and see how they perform. Narrowing them down to one environment - Lunar Lander - is only the first step.

2.5.4 Credits and Thanks

- Neural Networks
  - Andrew Trask
  - Sebastian Raschka
- Deep Q Learning
  - Deep Mind - Playing Atari with Deep Reinforcement Learning
  - Tambet Matiisen (here and here)
  - Eder Santana
  - Ben Lau
- Policy Gradients
  - Andrej Karpathy
  - Denny Britz
  - Open AI Gym Documentation
  - David Silver
  - Non-standard gradient optimization
- Keras
  - Francois Chollet (here and here)
- Box2D & Swig
  - Forums
- Cloud Computing on Amazon Web Services EC2 / GPU activation
  - Amazon Tutorials
  - Jie Yang
  - David Sanwald
  - TensorFlow Instructions