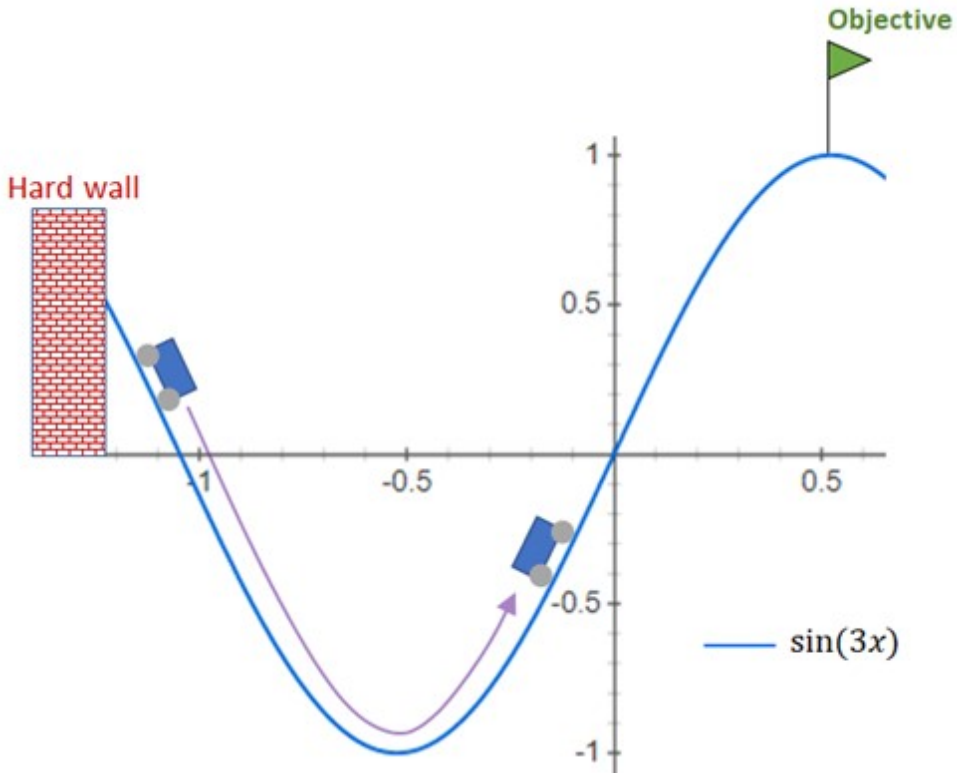
# Actor-critic using deep-RL: continuous mountain car in TensorFlow

Andy Steinbach Follow
Dec 18, 2018 · 10 min read

## Part 2:

In Part 1, we introduced pieces of deep reinforcement learning theory. Now we'll implement the **TD Advantage Actor-Critic algorithm** that we constructed. (Hint: this is the fun part! - Get the ***code*** now if you want)

We choose a classic introductory problem called "Mountain Car", seen in Figure 1 below. In this problem, a car is released near the bottom of a steep hill and its goal is to actively drive to the top (green flag). Since it is under-powered, it will need to apply the gas in forward and reverse directions to roll back and forth several times until reaching the objective. It moves under gravity, its own applied acceleration, friction, and being constrained to the curve of the hill. There is a hard inelastic wall on the left side.




In the usual variant of this problem, the car has only two **discrete** actions: forward or reverse at a fixed acceleration, and the goal is to reach the top as soon as possible. In the variant we consider, the applied acceleration can be any **continuous** value between a positive and negative limit, and the goal is to reach the top using the least cumulative applied energy. The reward upon reaching the objective is +100, and otherwise it is the negative amount of energy applied in each time step due to the applied power. This problem is harder than it appears, as it is highly non-linear, the initial release point varies to add stochasticity, and there is never any positive reward until the first time reaching the goal, posing an exploration challenge.

The **state space** of observations has two continuous variables: x-position and velocity of the car, with limits shown below. The **action** is a single continuous variable, representing an applied acceleration in the range

[-1.0, 1.0]. The initial state is a random x-position in the range [-0.6, -0.4], and zero velocity.

| State variable | Quantity | Min | Max |
|---|---|---|---|
| 0 | Car position | -1.2 | 0.6 |
| 1 | Car velocity | -0.07 | 0.07 |

Training an RL agent happens either with a real physical system (in the real world) or with a simulated system, requiring a computer simulator for the environment. If available, using a simulator is advantageous: it does not risk doing damage in the real world as the agent learns and it can be sped up faster than real time. Fortunately, we have an environmental simulator available through *OpenAI Gym,* a toolkit providing a number of simulated environments (Atari games, board games, physical systems, etc.), including continuous mountain car. Installing *Gym* is generally as easy as a `pip install gym` command in Linux with Python 3.5+ but varies depending on your system, and there is a lot of online support for installing *Gym*.

Let's get to know our Mountain Car *OpenAI* environment in Python:

We import the Gym package, and call the envs.make() method, creating an object for use as the Mountain Car environment

```
In [13]: import gym
         env = gym.envs.make("MountainCarContinuous-v0")

         # Calling observation_space.sample() returns a randomly sample
         d state variable
         env.observation_space.sample()
```

```
Out[13]: array([ 0.22510507,  0.00404529])
```

The reset() method is used to start a new episode.

```
In [14]: # reset() returns a rondom initial state

         initial_state = env.reset()
         initial_state
```

```
Out[14]: array([-0.46356136,  0.        ])
```

The step() method is called to execute the next action

```
In [16]: import numpy as np
```

**Gist RL blog number_1.ipynb** hosted with ❤ by **GitHub** **view raw**

Let's recall the algorithm we introduced in Part 1 and begin its implementation :

**Algorithm** TD Advantage Actor-Critic

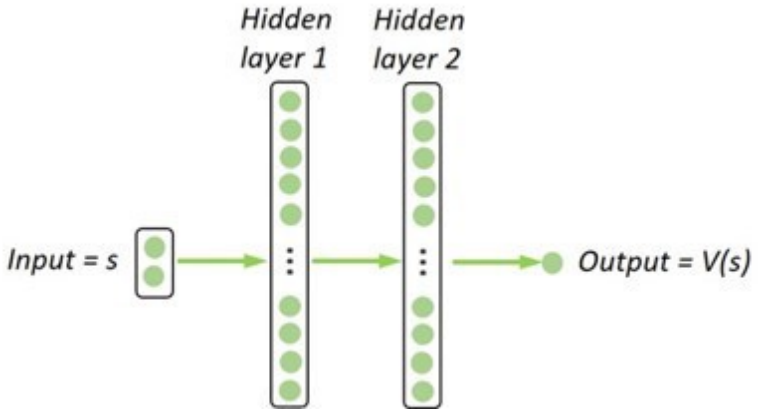
*Randomly initialize critic network* $V_\pi^U(s)$ *and actor network* $\pi^\theta(s)$ *with weights* $U$ *and* $\theta$
*Initialize environment E*
***for*** *episode = 1, M* ***do***
  *Receive initial observation state* $s_0$ *from E*
  ***for*** *t=0, T* ***do***
    *Sample action* $a_t \sim \pi(a|\mu,\sigma) = \mathcal{N}(a|\mu,\sigma)$ *according to current policy*
    *Execute action* $a_t$ *and observe reward* $r$ *and next state* $s_{t+1}$ *from E*
    *Set TD target* $y_t = r + \gamma \cdot V_\pi^U(s_{t+1})$
    *Update critic by minimizing loss:* $\delta_t = \left(y_t - V_\pi^U(s_t)\right)^2$
    *Update actor policy by minimizing loss:*

$$Loss = -log\left(\mathcal{N}(a \mid \mu(s_t), \sigma(s_t))\right) \cdot \delta_t$$

    *Update* $s_t \leftarrow s_{t+1}$
  ***end for***
***end for***

Figure 1

First, we'll use TensorFlow to build our neural networks for the actor (policy) and critic (value) functions. We'll start with the state-value function, building the simple fully-connected network below:




The value function network (critic model)

Here is the construction in TensorFlow:

Import TensorFlow and reset the graph

In [ ]:
```
import tensorflow as tf
tf.reset_default_graph()
```

Define and instantiate the state-value function network

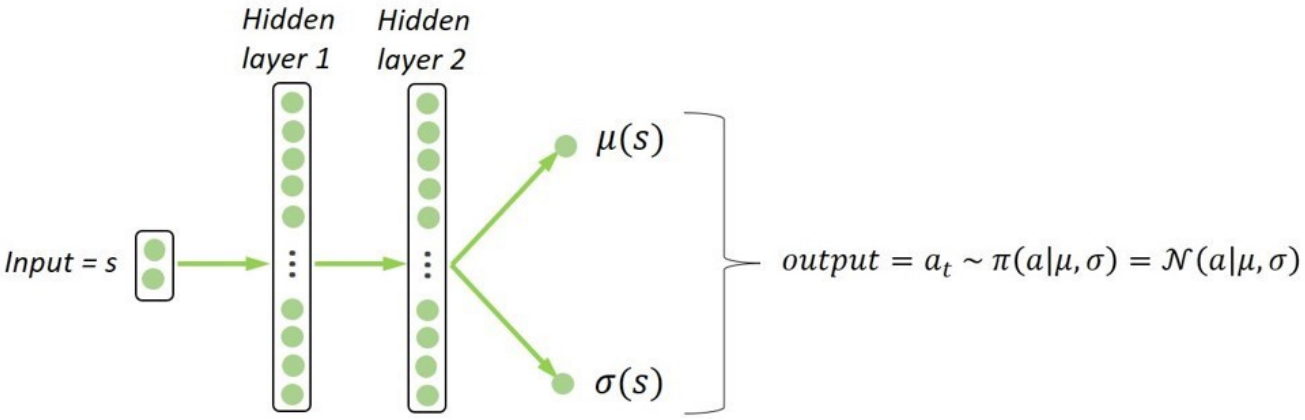
In [ ]:
```
input_dims = 2
state_placeholder = tf.placeholder(tf.float32, [None, input_di
ms])

def value_function(state):
    n_hidden1 = 400
    n_hidden2 = 400
    n_outputs = 1

    with tf.variable_scope("value_network"):
        init_xavier = tf.contrib.layers.xavier_initializer()

        hidden1 = tf.layers.dense(state, n_hidden1, tf.nn.elu,
                                  init_xavier)
        hidden2 = tf.layers.dense(hidden1, n_hidden2, tf.nn.el
u,
                                  init_xavier)
        V = tf.layers.dense(hidden2, n_outputs, None, init_xav
```

actor_critic gist2.ipynb hosted with ❤ by GitHub view raw

The code is pretty simple. We encapsulate the neural network construction in a function, and for simplicity hard code the layer sizes, the activation functions (elu), and initialization (Xavier). The number of inputs is the dimension of the state space (for Mountain Car this is two) and the value function output is a scalar (1-dimensional) with no activation.

Next, we'll build our stochastic policy function, estimated by the fully-connected network below. The network input is the state and output are two scalar functions, $\mu(s)$ and $\sigma(s)$, which are used as the mean and standard deviation of a Gaussian (normal) distribution. We will choose our actions by sampling from this distribution. The stochastic policy provides some degree of built-in exploration mechanism, since the network

initialization will cause a non-zero sigma value.




The stochastic policy network (actor model)

Here is the TensorFlow construction:

```
In [ ]: def policy_network(state):
            n_hidden1 = 40
            n_hidden2 = 40
            n_outputs = 1

            with tf.variable_scope("policy_network"):
                init_xavier = tf.contrib.layers.xavier_initializer()

                hidden1 = tf.layers.dense(state, n_hidden1,
                                          tf.nn.elu, init_xavier)
                hidden2 = tf.layers.dense(hidden1, n_hidden2,
                                          tf.nn.elu, init_xavier)
                mu = tf.layers.dense(hidden2, n_outputs,
                                     None, init_xavier)
                sigma = tf.layers.dense(hidden2, n_outputs,
                                        None, init_xavier)
                sigma = tf.nn.softplus(sigma) + 1e-5
                norm_dist = tf.contrib.distributions.Normal(mu, sigma)
                action_tf_var = tf.squeeze(norm_dist.sample(1), axis=
        0)
                action_tf_var = tf.clip_by_value(action_tf_var,
                                                 env.action_space.low
        [0],
                                                 env.action_space.high
        [0])
            return action_tf_var, norm_dist
```

actor_critic gist3.ipynb hosted with ❤ by GitHub view raw

Again, we hard code the parameters for simplicity of the example. The network has two hidden layers and outputs TensorFlow variables μ and σ, which we use to create a normal distribution using the TensorFlow function `tf.contrib.distributions.Normal()`. Finally, the output variable is a single sample from this probability distribution, whose value we clip to the allowed max and min of the action space. We remove the batch dimension using the `tf.squeeze()` function, because this is compatible with what's needed later from the return value. Remarkably, note that the returned sampled action is not just a Python variable - *it is a **TensorFlow variable** which can be **backpropagated!***

This section of code above and using this TD method was initially inspired

by the implementation of TD Advantage Actor-Critic in Denny Britz's GitHub RL repo (see here also for a wealth of great additional references).

Next, we instantiate the value and policy functions, and create the loss functions and training operations we will need. Recall from Part 1 that our actor and critic loss functions are:

$$Critic\ loss = \left(y_t - V_\pi^U(s_t)\right)^2$$

$$Actor\ loss = -log\left(\mathcal{N}(a \mid \mu(s_t), \sigma(s_t))\right) \cdot \delta_t$$

We implement these, including the placeholders `action_placeholder`, `target_placeholder` and `delta_placeholder` used in the code below to represent $a$, $y_t$ and $\delta_t$ respectively in the above equations - to be fed with values at training time. We create Adam Optimizer training ops for both the actor and critic loss functions, each with their own learning rate. We squeeze `v` again to remove the last tensor dimension, as we will work with 0-D scalars (not 1-D tensor arrays) in the training loop.

**actor_critic gist4.ipynb** hosted with ❤ by **GitHub** **view raw**

It is ***crucial*** to scale/normalize the input data. In the code above we import *scikit learn*, and using the call `env.observation_space.sample()` to sample input data, we create an array of state space samples to fit a *scikit learn* `StandardScaler()` object that subtracts the mean and normalizes states to unit variance. Using the `transform()` method, we create the `scale_state()` function to perform this operation below in the training loop.

Below is the main training loop mirroring exactly the algorithm in Figure 1. The discount factor is set to to 0.99, we open a TensorFlow session and initialize some variables. Each pass through the outer loop executes one full episode, and each pass through the inner loop will execute one environmental time step, until the `env.step()` call returns `done=True` ,

terminating the current episode. This happens only when the goal is reached, or the episode times out after 1000 steps of not reaching the goal. The `reward_total` variable accumulates the sum of all rewards in a full episode (i.e. the start state return, if gamma were set to 1). We use the same descriptive language taken from the algorithm in Figure 1 to comment the corresponding code below, for ease of connecting the two. ***Full code here.***

**actor_critic gist6.ipynb** hosted with ❤ by **GitHub** **view raw**

This is a **online algorithm** - which means it can collect labels and train itself while it is also executing as an working agent (in the context of RL). This means it operates with a batch size of one: it executes one action and

gets one reward and next state, and takes a training step/update based just on that one example. So, it is constantly learning with every step. The downsides of this approach are that it discards every new data instance after just one use, it can't benefit from larger batch sizes (which is computationally efficient and has a stabilizing effect), and the sequential training data examples in one episode are highly correlated, which turns out to be very problematic for training higher dimensional RL problems. More advanced actor-critic algorithms use an ***experience replay buffer*** to mitigate these shortcomings, by storing data for re-use. We will cover such an algorithm (DDPG) in a future part of this series, but you will notice that - at its heart - it nonetheless shares a very similar structure to our simpler algorithm here. (Note added 03-11-19: Here is an unpolished version of ***DDPG for Continuous Mountain Car in TensorFlow***. Please note this version is somewhat hacky, but it does run. If/when I have time I will polish the code and provide some documentation).

Note that all of the numerical data fed into placeholders during training comes from running TensorFlow variables earlier in the same pass through the inner loop, and those data instances are never used again. All state data fed to actor and critic models are scaled first using the `scale_state()` function. Since the loss function training placeholders were defined as 0-D tensors (i.e. scalars), we need to squeeze out any remaining NumPy dimensions before feeding them in training. There is a bit of shape juggling to accommodate the required shapes of different functions, so we mention these shapes in comments to more easily keep track of this.

Now onto the results!

When we run our main training loop, we generally get two types of

behavior as seen in the green and red curves in Figure 2. In this plot each point is the summed up rewards of every step in one individual episode (so, equivalent to the start state return with γ set to 1 for that episode).

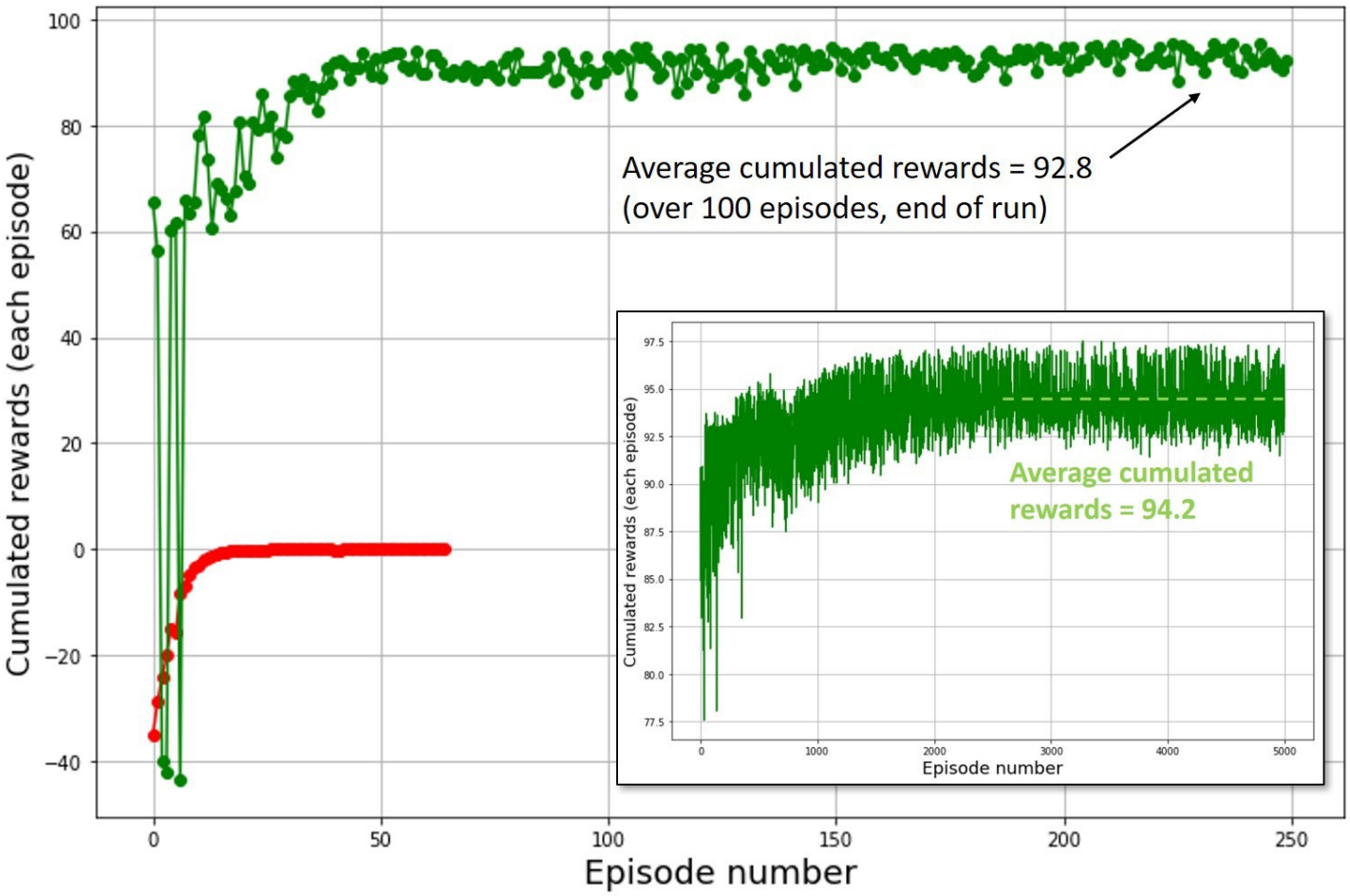



Figure 2. Training runs of our actor-critic algorithm

Either the policy's exploration achieves the goal randomly (by luck, getting the +100 point reward), ***in the initial few episodes,*** or it does not. If it does, it generally learns to solve the problem repeatably and reaches the goal every episode with increasing optimality, like in the green curves, where it averages a 92.8 cumulated rewards by the end of the 250 episodes shown, and in the inset, averaging 94.2 after a long training run of 5000 episodes. If it does not find the +100 reward in the first few episodes, then

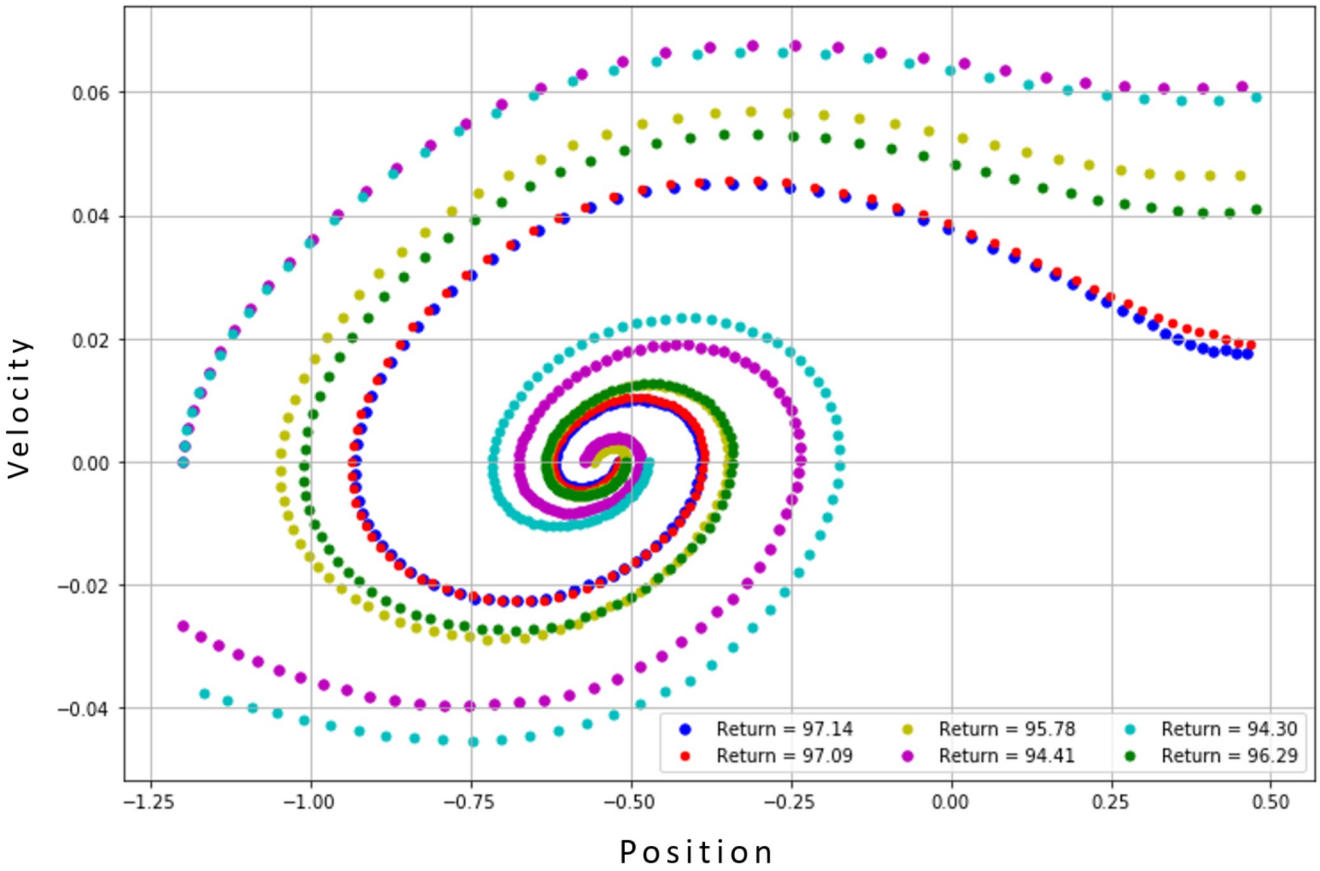
it falls into a sub-optimal local minimum with an end training state of 0 cumulated rewards (red curve). Since the car does not reach the top of the hill it quickly learns the best way to minimize energy is just to apply zero power and come to rest at the ***bottom of the hill.*** Even a large amount of external exploration noise added to the policy will not easily fix this, because the bad local minimum will become "burned-in" to the network parameters by the policy training and hard to undo.

Because of this burn-in effect, the easiest way to solve the problem is to re-start the training (with newly randomized network parameters) every few episodes until it finds the top of the hill. The re-start can be manually performed or automatically done by the program itself. The initial exploration built into the stochastic policy sampling is enough to find the +100 goal within a reasonably small number of runs — when the bad optimum is not yet burned in.

This is a common problem in RL: training can collapse to a local optimum (spoiling prior learning) since most RL problems are not convex, meaning there is not a single global minima/maxima. A powerful algorithm designed to treat this issue is **Trust Region Policy Optimization** (TRPO), which at every training step defines a safe local region for allowed change in the policy parameter space. This region is safe because it is *locally convex* and so under this algorithm every training step improves the policy. See the original paper, or this excellent blog for more details on TRPO.

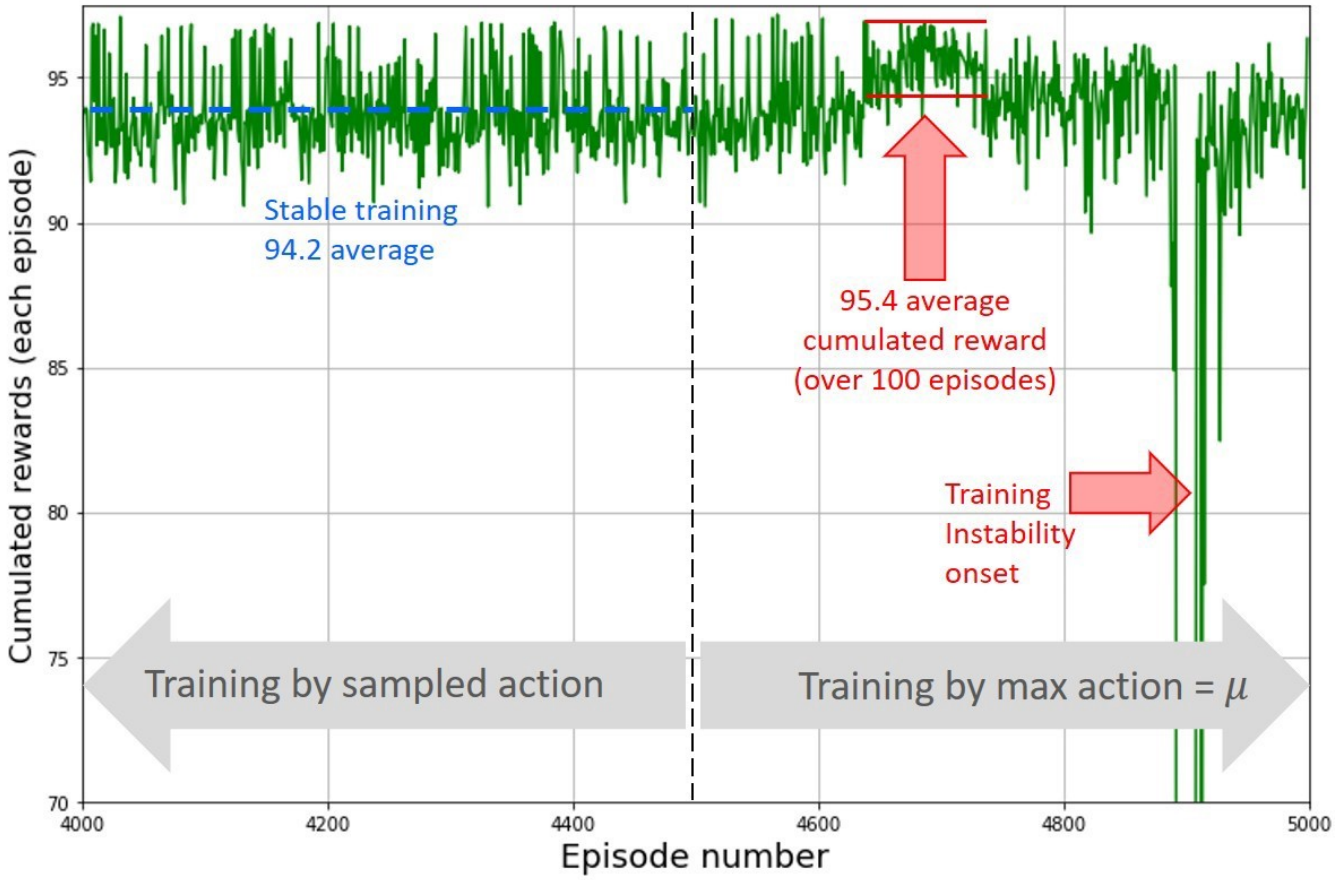
Now let's look at trajectories in state space (position, velocity). In the figure below, each color shows a single start-to-finish episode. The six episodes shown where taken from the end of a long training run like the one in the Figure 2 inset. We see that each episode starts from zero velocity and in the

range [-0.6, -0.4], as noted earlier, and ends at the RHS of the plot, having reached the top of the hill. The key shows the cumulated rewards of each episode, labeled Return (technically it is the Return with gamma = 1). Note that the Return is higher when the goal is reached with lower velocity, because any non-zero final velocity was wasted expended energy. Note also the two trajectories hitting the leftmost stop "hard wall" are reset to zero velocity. These have the lowest Returns in the group because they wasted the energy lost in the wall collision.




Finally, as is customary in training stochastic policies, we have been choosing our actions by sampling from the policy function probability distribution. But what if, after this sampled training, we just chose the argmax action μ (action maximizing probability)? The figure below shows

a training run where we thoroughly train the policy by sampled action, and then switch (at episode number 4500) to training by argmax action (=μ). We wee that in general it destabilizes the training and leads eventually to collapse of the policy. However, there is a short window (red lines) where we eke out an additional reward before the instability sets it. Running the policy (but not training it) using argmax action did not result in a noticeably higher Return.




So that's it! We have walked though the full introduction and implementation of a simple variant of actor-critic reinforcement learning!

Please feel free to comment and let me know if you spot mistakes or have suggested improvements. As always, if you liked it please give a clap!

We will be adding an implementation of the **Deep Deterministic Policy Gradient (DDPG)** algorithm when time permits to show how a more advanced algorithm tackles some of the issues we've encountered, and give another example of a continuous action space actor-critic solution. (Note added 03-11-19: Here is an unpolished version of ***DDPG for Continuous Mountain Car in TensorFlow***. Please note this version is somewhat hacky, but it does run. If/when I have time I will polish the code and provide some documentation).

My code in this blog was initially inspired by the RBF function approximator implementation of TD Advantage Actor-Critic in Denny Britz's GitHub RL repo (see here also for a wealth of great additional references).

Machine Learning Reinforcement Learning TensorFlow