Implementing Actor Networks for Discrete Control in TensorFlow 2.0

Method · September 2020

1 author:

Wouter van Heeswijk
University of Twente
34 PUBLICATIONS **288** CITATIONS

Implementing Actor Networks for Discrete Control in TensorFlow 2.0

W.J.A. van Heeswijk

September 4, 2020

Abstract

In this note, we show how to implement and update a discrete actor network in TensorFlow 2.0. First, we show the connection between stochastic gradient ascent as presented in reinforcement learning textbooks and the stochastic gradient descent used to update neural networks. Second, we provide a Python implementation based on the `GradientTape` functionality. Finally, we perform some numerical experiments on a multi-armed bandit problem to demonstrate the working of the algorithm.

Keywords: policy gradient, discrete control, deep reinforcement learning, neural network, cross-entropy loss, TensorFlow 2.0

This method paper explains how to update actor networks for discrete control. This work goes hand in hand with our note on actor networks for continuous control [5], which provides a bit more detail and background information.

Textbooks and lecture slides typically provide update rules based on stochastic gradient descent, but these cannot be directly applied to update neural network weights. This note shows how an actor network can be implemented in Python using the TensorFlow 2.0 library[1]. The algorithm is tested on a multi-armed bandit problem.

[1]Python 3.8 and TensorFlow 2.3 were used for this implementation.

1 Mathematical background

We start by discussing some basic properties and simplifications. A basic knowledge of policy gradients and deep reinforcement learning is expected. The design of our actor-network follows the classical policy gradient algorithm REINFORCE [7]. The actor network is defined by a decision-making policy $\pi_\theta(a \mid s) = \mathbb{P}(a \mid s; \theta)$, where s is the state, a is the action (selected from a discrete and finite action set), and θ represents the network parametrization. The input to the actor network is the state s itself or a derived feature vector $\phi(s)$.

After taking an action, we observe a corresponding reward $\hat{v}$. Defining a learning rate α, the update rule we typically see looks as follows [4]:

$$\Delta\theta = \alpha \nabla_\theta \log\left(\pi_\theta(a \mid s)\right) \hat{v} \ . \tag{1}$$

Essentially, $\pi_\theta(a \mid s)$ represents the probability of selecting a given action. For the weight update, we perform a so-called log transformation on this probability. From calculus, you may recall that $\log(1) = 0$ and $\log(x) < 0, \forall x < 1$. As the probabilities of all actions eminently must sum to 1, this log transformation typically returns a negative number. Note that actions with low probabilities return numbers of larger magnitude. We multiply the log probability with the reward signal $\hat{v}$. Naturally, large rewards spur a larger weight update. One can see that actions with low probability and high rewards result in the strongest policy updates. For linear approximation schemes the update rule can directly be applied to the weight corresponding to each feature [6].

When training neural networks we usually do not update the weights by hand, but instead let automated backpropagation methods take care of the updates. Rather than updating weights into a direction that increases the expected policy reward, in deep reinforcement learning we update actor networks by minimizing a loss function. Usually, we compute this loss by squaring the difference between the predicted value and the observed value; this loss function is known as the mean-squared error. We then backpropagate the loss through the network, computing partial losses and gradients to update the network weights.

Actor network updates are a bit atypical as we do not have predicted and observed values. However, from the update rule we may deduce a 'pseudo-loss' function that enables us to update the actor network using backprop-

agation [1]. Computing the loss function is an intermediate step in defining the update rule, therefore the learning rate α and the gradient notation ∇_θ are not part of the loss function itself. Furthermore, remind that the update rule performs gradient *ascent*, whereas in the neural network we minimize the loss by means of gradient *descent* (either the classical algorithm or a variant). Thus we need to add a minus sign. After these modifications we obtain the following pseudo-loss function, also known as the log loss function or the cross-entropy loss function [2]:

$$\mathcal{L}(a, s, \hat{v}) = -\log(\pi_\theta(a \mid s))\hat{v} \ . \tag{2}$$

2 TensorFlow 2.0 implementation

The loss function in Equation (2) is the basis for our backpropagation, now we address its implementation in TensorFlow 2.0. Using TensorFlow functions where appropriate, we may define this loss function as follows in Python:

```
"""Cross entropy loss function for RL"""
def cross_entropy_loss(probability_action, state, reward):
        log_probability = tf.math.log(probability_action + 1e-5)
        loss_actor = - reward * log_probability

        return loss_actor
```

It is imperative to remember that TensorFlow is able differentiate the loss function to automatically compute the gradients, therefore we must first make a forward pass through the actor network. Furthermore, we cannot use our loss function into functions such as `model.fit` or `model.compile`, as these require exactly two arguments. In contrast, the `GradientTape` functionality allows for loss functions with additional arguments. It memorizes how the loss function was computed and subsequently backpropagates the loss to compute and apply the gradients. The weight update is composed of three steps: (i) computing the custom loss function, (ii) computing the gradients with the computed loss and the trainable network weights, and (iii) applying the gradients with an `optimizer` of your choosing. After invoking the `GradientTape`, we take the following steps:

```
"""Compute and apply gradients to update network weights"""
with tf.GradientTape() as tape:
        # Obtain action probabilities from network
        action_probabilities = actor_network(state)

        # Select random action based on probabilities
        action = np.random.choice(len(bandits),
        ↪  p=np.squeeze(action_probabilities))

        # Obtain reward from bandit
        reward = get_reward(bandits[action])

        # Store probability of selected action
        probability_action = action_probabilities[0, action]

        # Compute cross-entropy loss
        loss_value = cross_entropy_loss(probability_action, state,
        ↪  reward)

        # Compute gradients
        grads = tape.gradient(loss_value[0],
        ↪  actor_network.trainable_variables)

        #Apply gradients to update network weights
        opt.apply_gradients(zip(grads,
        ↪  actor_network.trainable_variables))
```

3 Numerical example

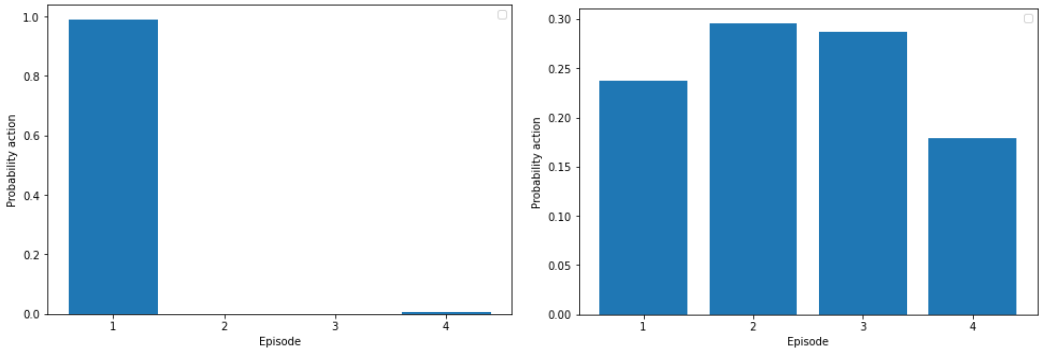
We illustrate the mechanism of the discrete control algorithm with the classical multi-armed bandit problem [3]. In this setting, we have a number of slot machines (four in this example) that have certain payoff properties. Each machine i has an unknown mean μ_i and an unknown standard deviation σ_i; for simplicity we keep the latter constant for each machine. Clearly, the optimal policy is to always play the machine with the highest expected payoff; but we must first explore in order to reliably estimate the true μ_i of each machine. The full Python code may be found on GitHub [2].

For the experiments, we define a fully connected neural network consisting of two hidden layers (containing five neurons each and using ReLU

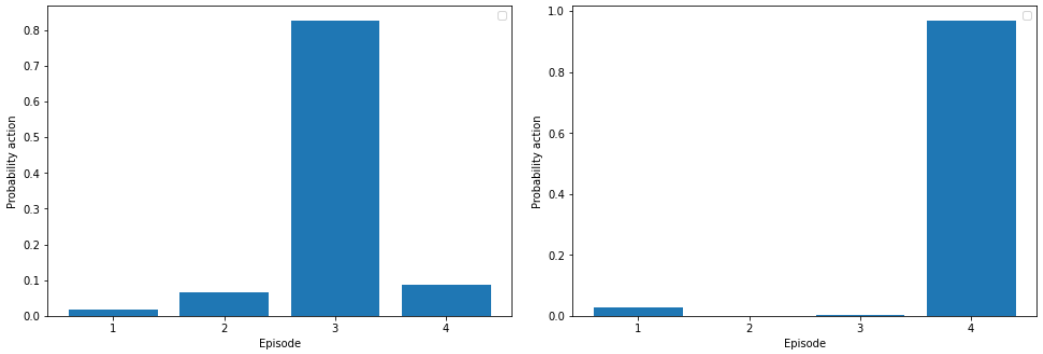
[2] www.github.com/woutervanheeswijk/example_discrete_control

activation functions), the Adam optimizer (with default learning rate 0.001). We perform 10,000 training iterations. The connections to the output layer are initialized such that each machine is played with equal probability at the beginning. For the remainder of the weights we use He initialization.

Figure 3 shows the results of four such experiments. After 10,000 iterations the machines' properties are well-explored and we will mostly play the machine with the highest expected payoff. With equal payoffs we are indifferent about the machine. Remind that policy gradient algorithms return a stochastic policy rather than a deterministic one. Therefore, unlike value-based algorithms such as Q-learning, some degree of exploration remains due to the non-zero probabilities placed on the remainder of the machines. The smaller the differences between pay-offs, the higher the chance that other machines with suboptimal pay-offs are played.



(a) $\mu_1 = 1.1$, $\mu_2 = 0.0$, $\mu_3 = 1.0$, $\mu_4 = 1.0$ (b) $\mu_1 = 1.0$, $\mu_2 = 1.0$, $\mu_3 = 1.0$, $\mu_4 = 1.0$



(c) $\mu_1 = 4.8$, $\mu_2 = 4.9$, $\mu_3 = 5.1$, $\mu_4 = 4.9$ (d) $\mu_1 = 1.0$, $\mu_2 = 0.9$, $\mu_3 = 0.9$, $\mu_4 = 1.0$

Figure 1: Examples of policies for various multi-armed bandits settings.

References

[1] Levine, S. (2019). CS 285: Deep reinforcement learning, decision making, and control – Policy gradients. *University of California, Berkeley*.

[2] McCaffrey, J. D. (2016). Log loss and cross entropy are almost the same. `https://jamesmccaffrey.wordpress.com/2016/09/25/log-loss-and-cross-entropy-are-almost-the-same/`. Accessed: 2020-09-04.

[3] Ryzhov, I. O., Frazier, P. I., and Powell, W. B. (2010). On the robustness of a one-period look-ahead policy in multi-armed bandit problems. *Procedia Computer Science*, 1(1):1635–1644.

[4] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, 2nd edition.

[5] Van Heeswijk, W. J. A. (2020a). Implementing Gaussian actor networks for continuous control in Tensorflow 2.0. *researchgate.net*.

[6] Van Heeswijk, W. J. A. (2020b). Smart containers with bidding capacity: A policy gradient algorithm for semi-cooperative learning. In Lalla-Ruiz, E., Mes, M. R. K., and Voß, S., editors, *Computational Logistics*, pages 1–15.

[7] Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8(3-4):229–256.