EV3 robot controller with Q-learning and Neural Network

Amra Omanović, Nejka Bolčič, Magda Nowak-Trzos,
University of Ljubljana

Index Terms—Deep Learning, Q-learning, machine learning, neural networks, line follower

1 Introduction

The main purpose of the project was to create the EV3 robot controller, capable of steering the robot in the way that it follows the previously prepared black line.

Lego Mindstorms EV3 is the third generation robotics kit in Lego's Mindstorms line. The EV3 set, besides elements necessary to assemble the robot, includes following sensors:

- Touch Sensor
- Color Sensor
- Infrared Sensor

Among all available items, to complete our task we needed only the Color Sensor. The sensors values are treated as our algorithm input, and based on them the decision of further movement is taken.

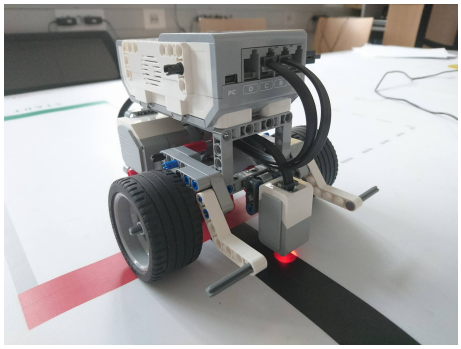

Fig. 1. EV3 robot controller

Instructing the robot to move and turn is accomplished by the Large Motors which rotate in a predetermined direction where positive amount of power (e.g.75) will cause a clockwise rotation and negative power (e.g. -45) will cause a counter-clockwise rotation.

To accomplish our goal we used two different approaches.

1) Reinforcement learning
The reinforcement learning algorithms offer one of the most general frameworks in learning subjects. We will present an approach using the Q-Learning algorithm on a Lego robot in order for it to learn "by itself" how to follow a black line drawn down on a white surface, using Python as a programming environment.
2) Neural Network approach
The neural network approach is slightly more complicated. First of all the training data has to be collected, based on which the neural network is trained. In our case the data was collected from previously implemented Q-learning algorithm. After the training process the parameters of neural network are established and NN can be applied on the robot.

- *M. Shell was with the Department of Electrical and Computer Engineering, Georgia Institute of Technology, Atlanta, GA, 30332. E-mail: see http://www.michaelshell.org/contact.html*
- *J. Doe and J. Doe are with Anonymous University.*

Manuscript received April 19, 2005; revised August 26, 2015.

2 Q-learning

The reinforcement learning assumes that the world can be described by a set of states S, finite number of actions A and for each step the robot observes the state of the world S_t and chooses an action a_t. After taking the action, the reward function gives a reward r_t to the robot.

The model of reinforcement learning looks as follows:

1) check the state S_t
2) pick an action a_t
3) do that action
4) check the new state S_{t1}
5) get the reward r_{t1}
6) learn from experience(relate state-action reward value and store it)
7) repeat

The Q-learning algorithm is the particular reinforcement learning algorithm which approximates the value of state-action function Q iterating the process. We keep an estimate of the $Q(s, a)$ function in a form of matrix. This values are updated as the agent achieves more experience.
We use the following notation:

- $Q(s, a)$ is the learning function
- $R(s, a)$ is the matrix of "rewards" and "punishments"
- $s \in S$ is the current state
- $a \in A$ is the executed action

- $s' \in S$ is the state driven by the action a
- $0 < \alpha$ is the learning rate
- $0 \leq \gamma$ is the discount rate

The pseudocode of the Q-learning algorithm is:

1) initialize $Q(s, a)$ on a small random values, $\forall s, \forall a$
2) check the state s
3) pick an action a and execute it out
4) check the new state s' and reward the last action
5) $Q(s, a) \leftarrow Q(s, a) + \alpha * (R(s, a) + \gamma * max(Q(s', :)) - Q(s, a))$
6) go to step 2.

We defined three states:

1) sensor detects black- returns the value in range (0,15)
2) sensor detects grey - returns the value in range (16,85)
3) sensor detects white - returns the value in range (86,100)

and three actions:

1) turn left - is achieved by applying the speed to the right motor
2) turn right - is achieved by applying the speed to the left motor
3) go forward - is achieved by applying the speed to both motors

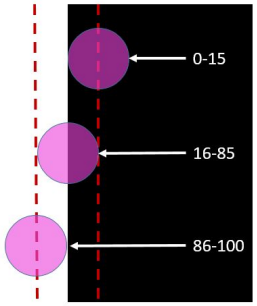

Fig. 2. Color discretization

We chose the next values in our implementation:

- $\alpha = 1.0$
- $\gamma = 0.8$
-

$$R(s, a) = \begin{bmatrix} 1 & -10 & 1 \\ -100 & 10 & -1 \\ -100 & -10 & 100 \end{bmatrix}$$

The matrix R is the reward-punishment matrix that defines values with which the Q matrix will be updated.

Since the following line problem is relatively easy and the R matrix has strong punishments and rewards our solution converges very fast and robot learns quickly how to follow the line. It happens after approximately 4-5 iterations.

3 Neural Networks

We created a simple neural network with 1 hidden layer that learned to steer the EV3 robot based on the data, collected during Q-learning process. The data was collected in the form (colorBeforeMove, colorAfterMove, move), where the colors and moves were coded as follows:
Colors:

- 0 black
- 1 white
- 2 gray

Moves:

- 0 left
- 1 right
- 2 forward

3.1 Preprocessing of data

The tuples were collected in a .csv file that we processed in order to get the data format needed to enter into the neural network. Each input consists of the previous movement and states reading along with the last color reading, and the corresponding output is the movement following that reading. E.g input is (black, black, left, gray) and the output is forward.

In order to achieve better accuracy in the neural network all values were converted to one-hot format in the following way:

$$0 = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$
$$1 = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}$$
$$2 = \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}$$

This resulted in input vectors of length 12 and output vectors of length 3. In order to avoid bias in neural network the pairs of input and output vectors were shuffled before training. We used 90% of the data for training and 10% to test the neural network.

3.2 Neural network model

We feed the preprocessed data into the neural network. First we randomly initialize the weights for the input and the hidden layer with mean 0. We then train the neural network. In the feed forward we use the sigmoid nonlin function to predict the output based on the weights:

```
def nonlin(x, deriv=False):
    if (deriv == True):
        return x * (1 - x)
    return 1 / (1 + np.exp(-x))
```

We calculate the error based on the target value. To adjust the weights we calculate the deltas for each layer:

$$l2\_delta = l2\_error * nonlin(l2, deriv = True) * descent\_rate$$

We adjust the error according to the level of confidence of the result (where it lies on the sigmoid function) and multiply it by the descent rate, which controls the speed of convergence of the results.
Once the deltas are calculated we adjust the weights and

repeat the entire process until the error decreases to an acceptable level.
In our network we used a dataset with 700 samples. The hidden layer has 2 neurons and in order to prevent too steep convergence we use a descent rate of 0,01. The trained network achieves $98,8\%$ accuracy.
After the network is trained we connect the EV3 and send the collected data to the network as the robot moves. The model produces the instructions for the robot to move in real time according to the color read from the sensor, effectively enabling the robot to follow the line.

4 Summary

Neural network achieves good accuracy of 98.8% but it needs to collect and preprocess the data before training process, while Q-learning is inaccurate in the beginning and with time it gets better and even has smoother movements than the neural network.

References

[1] Michael Nielsen, ,,Neural Networks and Deep Learning", May 2017
[2] Andrew Trask, ,,A Neural Network in 11 lines of Python", July 2015, *http://iamtrask.github.io/2015/07/12/basic-python-network/*
[3] V. Cruz-Alvarez, E. Hidalgo-Pena, H. Acosta-Mesa, ,,A line follower robot implementation using Lego's Mindstorms Kit and Q-learning", February 2012