Summary of recent thesis work

Coogan, Clement

May 2018

1 Joint update

1.1 Movement of Pioneer

The pioneer should have a more complex movement pattern than going straight indefinitely. This is accomplished by having the car turn every x steps for a given amount of y steps with a random (but limited) turning radius, in alternate directions (i.e. left, right, left, etc.). This is accomplished with the following code in the pioneer's Lua script:

```
function turn()
    if(turnRight == true) then
        simSetJointTargetVelocity(motorRight,v0+speedIncrease)
        simSetJointTargetVelocity(motorLeft,0)
    else
        simSetJointTargetVelocity(motorLeft,v0+speedIncrease)
        simSetJointTargetVelocity(motorRight,0)
    end
end

function endTurn()
    simSetJointTargetVelocity(motorRight,v0)
    simSetJointTargetVelocity(motorLeft,v0)
end

-- In the actuation function (i.e. every step)
  -- Turn every turnFrequency steps
    if(math.fmod(step,turnFrequency)==0) then
        turning = true
        math.randomseed(os.time())
        randomNumber = math.random()
        speedIncrease = math.fmod(randomNumber * 100, v0/2)
        --speedIncrease = v0/4
    end

    -- As long as turning is true
    if(turning == true) then
```

```
    -- If stepsLeft equals turnLength,
    -- call turn function
    if(stepsLeft == turnLength) then
        turn()
    end
    -- Decrease stepsLeft
    stepsLeft = stepsLeft - 1
end

...

-- If stepsLeft equals 0
if(stepsLeft == 0) then
    -- turning is set to false
    turning = false
    -- turnRight is set to the opposite,
    -- so the next turn goes in opposite direction
    turnRight = not turnRight
    -- stepsLeft is resetted to turnLength
    stepsLeft = turnLength
    -- endTurn function is called
    endTurn()
end
```

1.2 Keeping distance to pioneer

The HiWi also implemented a way for the snake not to come too close to the pioneer. However, this resulted in weird movement. Thus, a new distance keeping algorithm was needed and the subsequent code was used:

```
-- Read res (-1, 0 or 1) and dist from proxSensor
    res, dist = simReadProximitySensor(proxSensor)

    -- Proximity functionality
    -- If proxSensor senses an object
    if (res == 1) then
        -- If distance below 1.5, slow down
        if(dist < minDistance) then
            w = w - speedChange
        -- If distance above 3, speed up
        elseif(dist > maxDistance) then
            w = w + speedChange
        end
    end
```

To ensure smoother acceleration speedChange was set to 0.003.

1.3 Saving learning parameters

In order to not only have access to the learned weights for each session but also have the capability to reproduce the results, the corresponding learning parameters need to be saved. We decided to do so in the JSON format for simplicity. For this we saved the necessary parameters from parameters.py as a dictionary and converted this to JSON like this:

```
params['w_min'] = p.w_min
params['w_max'] = p.w_max
. . .
params['r_min'] = p.r_min
params['reward_slope'] = p.reward_slope

# Save to json file
json_data = json.dumps(params)
with open(p.path+'/training_parameters.json','w')
as file:
    file.write(json_data)
```

However, as learning is also influenced by the motion parameters of the snake robot and Pioneer, these have to be saved as well. For this the parameters were published at the beginning of a training session out of V-REP and subscribed to from the running python script as follows:

```
# environment.py
self.params_sub =
rospy.Subscriber('parameters', String,
self.params_callback)

...

def params_callback(self, msg):
        if(self.first_cb):
            self.snake_params = msg.data
            self.first_cb = False
        else:
            self.pioneer_params = msg.data
        return
```

The snake and pioneer parameters are saved separately but use the same subscription topic and thus the same callback function. V-REP sends the snake parameters first and the Boolean first_cb variable is used for separation. In the corresponding Lua scripts the parameters are published as such:

```
function publishParameters()
    data = {}
    parameters = '{"Snake parameters":{' .. '"speedChange": '
    .. speedChange .. ', "minDistance": '.. minDistance ..',
    "maxDistance": '.. maxDistance ..'}}'
    data['data'] = parameters
```

```
    simExtRosInterface_publish(paramsPub,data)
end
```

They are then saved in training.py similarly to the python parameters.

```
with open(p.path+'/snake_parameters.json','w') as file:
    file.write(snake_params.__str__())

with open(p.path+'/pioneer_parameters.json','w') as file:
    file.write(pioneer_params.__str__())
```

1.4 Blind steps

Although the snake's head movement is compensated for and is hence more static it still moves quite a bit. This results in the snake losing sight of the pioneer (especially when fairly close to the car) although it is following it well. However, this temporary (even just for one time step) loss of sight results in a complete reset of the episode. To solve this problem, the snake was allowed to make a set amount of consecutive steps without seeing the pioneer without being reset. The code to accomplish this:

```
def image_callback(self, msg):
    ...
        if M['m00'] == 0:
            self.blind_steps_counter += 1
            if self.blind_steps_counter == blind_steps:
                self.terminate = True
                self.blind_steps_counter = 0
            self.cx = 0.0
        else:
            self.blind_steps_counter = 0
            self.terminate = False
            self.cx =
            2*M['m10']/(M['m00']*img_resolution[1]) - 1.0
```

1.5 Smaller code changes

There were also some smaller code changes that needed to be done to correct errors.

1.5.1 Reward signal

```
# Old reward
r = 1 - abs(self.cx)

# New reward
r = self.cx
```

1.5.2 getState() function

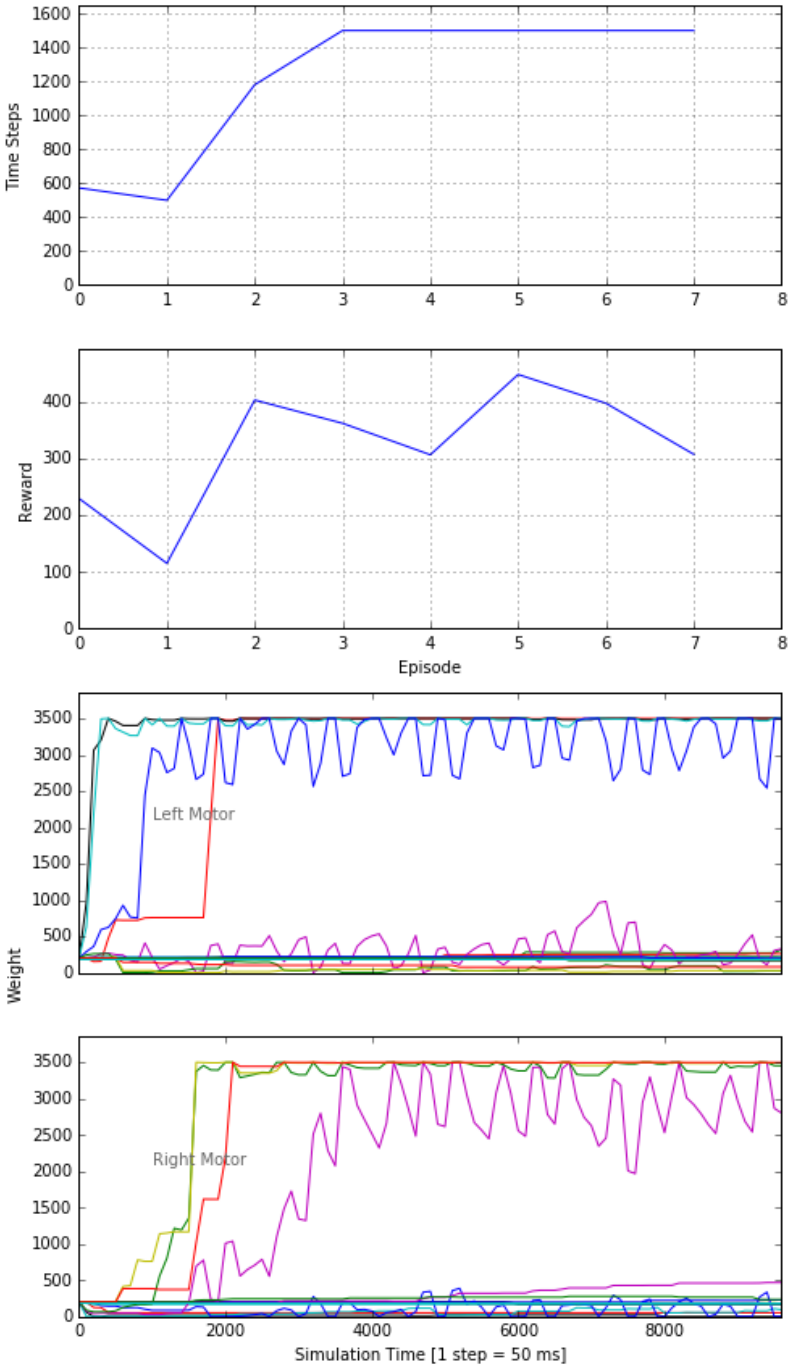
```
# Old getState()
def getState(self):
        new_state =
        np.zeros((resolution[0],resolution[1]),dtype=int)
        if self.imgFlag == True:
                for y in range
                (img_resolution[0] - crop_top - crop_bottom):
                        for x in range(img_resolution[1]):
                                if self.img[y + crop_top,x] > 0:
                                new_state[x//(img_resolution[1]
                                //resolution[0]),
                                y//(img_resolution[0]//resolution[1])]
                                += 4
        return new_state

# New getState()
def getState(self):
        new_state = np.zeros((resolution[0],resolution[1]),dtype=int)
        if self.imgFlag == True:
            for y in range(img_resolution[1] - crop_top - crop_bottom):
                for x in range(img_resolution[0]):
                    if self.img[y + crop_top, x] > 0:
                        new_state[x//(img_resolution[0]
                        //resolution[0]), y//((img_resolution[1]
                        - crop_top - crop_bottom)//resolution[1])]
                        += 4
        return new_state
```

1.6 Update to RSTDP training plotting script

The original plotting script did not work. This was fixed and new plots were added. The script now plots the steps per episode, the cumulative reward per episode, and the weights of the left and right motor neurons over the episodes.

1.7 Learning success

The hitherto mentioned changes resulted in the first learning successes as exemplified by the following graphs:


Time Steps
Reward
Episode
Left Motor
Weight
Right Motor
Simulation Time [1 step = 50 ms]

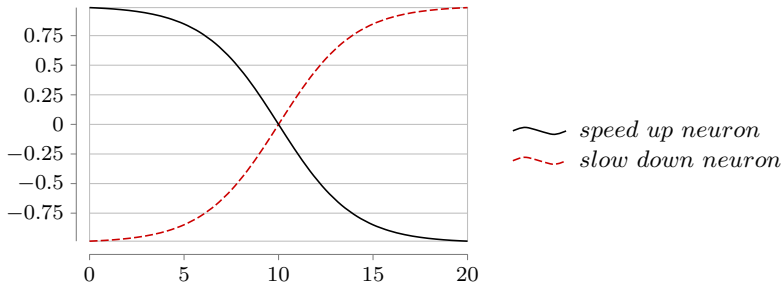
2 Anselm's updates

2.1 Proposal for speed learning

Up to now the snake's speed was set and then "manually" adjusted by reading the value of a proximity sensor and then increasing or decreasing the speed by a set delta. It would be better if the snake could also learn how fast it should move by itself. To do so two new output neurons were introduced: one for speeding up and one for slowing down. The reward is based on how many of the DVS sensor's pixels are red. It is modelled as a logistic function of the form:

$$r = \frac{2}{e^{-k*(x-x_0)} + 1} - 1 \tag{1}$$

where x is the number of red pixels, x_0 is the number of red pixels for the ideal distance, and k is the steepness of change in reward. This function is for the slow down neuron. For the speed up neuron the reward is negated. Here is an example of when the ideal number of pixels is set to 10:



The speed is then updated (and published to V-REP) every step as follows:

```
# Snake speed
    self.speed = self.speed + float(n_faster-n_slower)/(100*n_max)

    if(self.speed < 0):
        self.speed = 0
```

This current implementation does make the snake speed up and slow down appropriately but the learning especially in conjunction with the turning radius doesn't yet work and I am trying to figure out why by tweaking the learning parameters.

2.2 Proposal for hidden layer

The student project that uses hidden layers unfortunately doesn't implement it via NEST. I am unsure whether to do it via NEST's topology library as described here or whether to just insert a layer like this:

```
spike_generators = nest.Create("poisson_generator", 8)
input_neurons = nest.Create("parrot_neuron",8)
hidden_layer = nest.Create("parrot_neuron",4)
output_neurons = nest.Create("iaf_psc_alpha",2)
spike_detectors =
```

```
nest.Create("spike_detector", 2,
            params={"withgid": True, "withtime": True})

syn_dict_generator = {"weight": 100}
syn_dict_neurons = {"weight": 100}

nest.Connect(spike_generators,
input_neurons, "one_to_one", syn_spec=syn_dict_generator)
nest.Connect(input_neurons, hidden_layer, "all_to_all",
syn_spec=syn_dict_neurons)
nest.Connect(hidden_layer, output_neurons, "all_to_all",
syn_spec=syn_dict_neurons)
nest.Connect(output_neurons, spike_detectors, "one_to_one")
```

The next question would be if the hidden layer is inserted in between the input layers and ALL output neurons or just the motor or speed neurons, for instance. Or if there are two separate hidden layers: one for the motor neurons and one for the speed neurons. Maybe then there could only be one speed neuron.

3 Christoph's updates

3.1 Wall following task (updated on 06/01/2018)

A possible field of application for a snake robot would be disaster rescue. One could image a scenario where a snake robot equipped with a head camera is assisting search and rescue efforts in an earthquake zone. It then needs to be able to autonomously move through buildings and smaller wholes. This is why I want to implement a wall following task for the snake in V-REP.

3.1.1 V-REP scene (updated on 06/05/2018)

I removed the Pioneer robot from the scene and added 10 walls forming a corridor. The walls are 0.25m thick, 7.5m long and 2m high. The distance between to opposite walls is 2.75m and the scene now looks as shown in Figure 1.

3.1.2 Reward calculation (updated on 06/05/2018)

The optimal position of the snake is the path in the middle between two opposite walls. In order to calculate the reward for the left and right motor neuron, the distance of the snake to the optimal path is needed. In environment.py, the following parameters and four points are defined for the distance calculation:

```
# Values for distance calculation
# Length of the wall in meter
self.length_wall = 7.5
# Init position of the snake in the global coordinate system of the scene
self.snake_init_position = [20.0, 0.0]
# Angle in degrees between the section 1 & 2 and 2 & 3
self.gamma1_deg = 20
```

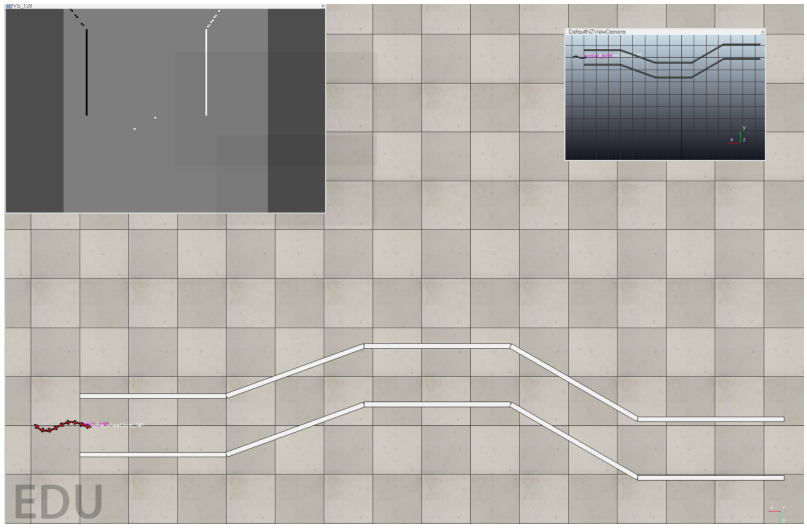


Figure 1: The snake moves through a corridor made of wall segments. The corridor is made up of five sections (section 1: straight, section 2: 20 degrees up, section 3: straight, section 4: 30 degrees down, section 5: straight

```
# Gamma1 in radians
self.gamma1_rad = self.gamma1_deg*math.pi/180
# Angle in degrees between the section 3 & 4 and 4 & 5
self.gamma2_deg = 30
# Gamma2 in radians
self.gamma2_rad = self.gamma2_deg*math.pi/180

# Points of the optimal path
# p1 = [20.0, 0]
self.p1 = self.snake_init_position
# p2 = [12.5, 0]
self.p2 = np.add(self.p1, [-self.length_wall, 0])
# p3 = [5.45, -2.57]
self.p3 = np.add(self.p2, [-self.length_wall*math.cos(self.gamma1_rad),
                           -self.length_wall*math.sin(self.gamma1_rad)])
# p4 = [-2.05, -2.57]
self.p4 = np.add(self.p3, [-self.length_wall, 0])
# p5 = [-8.54, 1.18]
self.p5 = np.add(self.p4, [-self.length_wall*math.cos(self.gamma2_rad),
                           self.length_wall*math.sin(self.gamma2_rad)])
# p6 = [-16.04, 1.18]
self.p6 = np.add(self.p5, [-self.length_wall, 0])
```

The distance is calculated with the following two functions. Depending on in which section the snake is in, the distance will be calculated with the appropriate end points of the optimal path.

```
def calculateDistance(self, snake_position, p1, p2):

    distance = np.cross(
                np.subtract(p2,p1),
                np.subtract(p1,snake_position))
                /
                norm(np.subtract(p2,p1))
    return distance

def getDistance(self, snake_position):

    snake_position = [snake_position[0], snake_position[1]]

    # Section 1
    if (self.p2[0] < snake_position[0] < self.p1[0]):
        section = "section 1"
        distance = self.calculateDistance(snake_position, self.p1, self.p2)
        return distance, section

    # Section 2
    if (self.p3[0] < snake_position[0] < self.p2[0]):
        section = "section 2"
        distance = self.calculateDistance(snake_position, self.p2, self.p3)
        return distance, section

    # Section 3
    if (self.p4[0] < snake_position[0] < self.p3[0]):
        section = "section 3"
        distance = self.calculateDistance(snake_position, self.p3, self.p4)
        return distance, section

    # Section 4
    if (self.p5[0] < snake_position[0] < self.p4[0]):
        section = "section 4"
        distance = self.calculateDistance(snake_position, self.p4, self.p5)
        return distance, section

    # Section 5
    if (self.p6[0] < snake_position[0] < self.p5[0]):
        section = "section 5"
        distance = self.calculateDistance(snake_position, self.p5, self.p6)
        return distance, section
```

In the step function of environment.py, the reward is then calculated like this:

```
# Get distance
d = self.getDistance(self.pos_data)
```

```
# Set reward signal
r = abs(d)
```

3.2 Episode termination (updated on 06/05/2018)

An episode is terminated when the snake reaches 1000 steps or it is 0.8m away from the optimal position. In parameters.py max_steps and reset_distance are set and in the step function of environment.py the following code is implemented:

```
# Terminate episode if  max_steps or reset_distance are reached
if self.steps > max_steps or abs(d) > reset_distance:
    self.terminate = True

t = self.terminate
if t == True:
    self.steps = 0
    self.reset()
    self.terminate = False
```

3.3 Training evaluation (updated on 06/05/2018)

The training seems to work well in the beginning. With each episode the snake goes further until it reaches max_steps after five episodes as can be seen in Figure 2. But after five episodes something happens that I cannot explain. The cumulative reward per episode decreases significantly as well as the total steps per episode during episodes six to nine and then increases again. Figure 3 displays the weights after the training process.

3.4 Current status and outlook (updated on 06/05/2018)

At the time of writing I am looking into the problem described above. In addition to that I am reading a lot about backpropagation and reinforcement learning in general because the resources for reinforcement learning with SNNs are sparse.

My next steps are the following:

- Fix the following minor problem: sometimes during training, the training stops with an error because the DVS doesn't send any data.
- Evaluate different parameters: training_length, reset_distance, dvs_resolution, reduced resolution, w0_min, w0_max
- Model different wall following scenarios (curved, 8-shaped, etc.)
- Work on the theory contribution

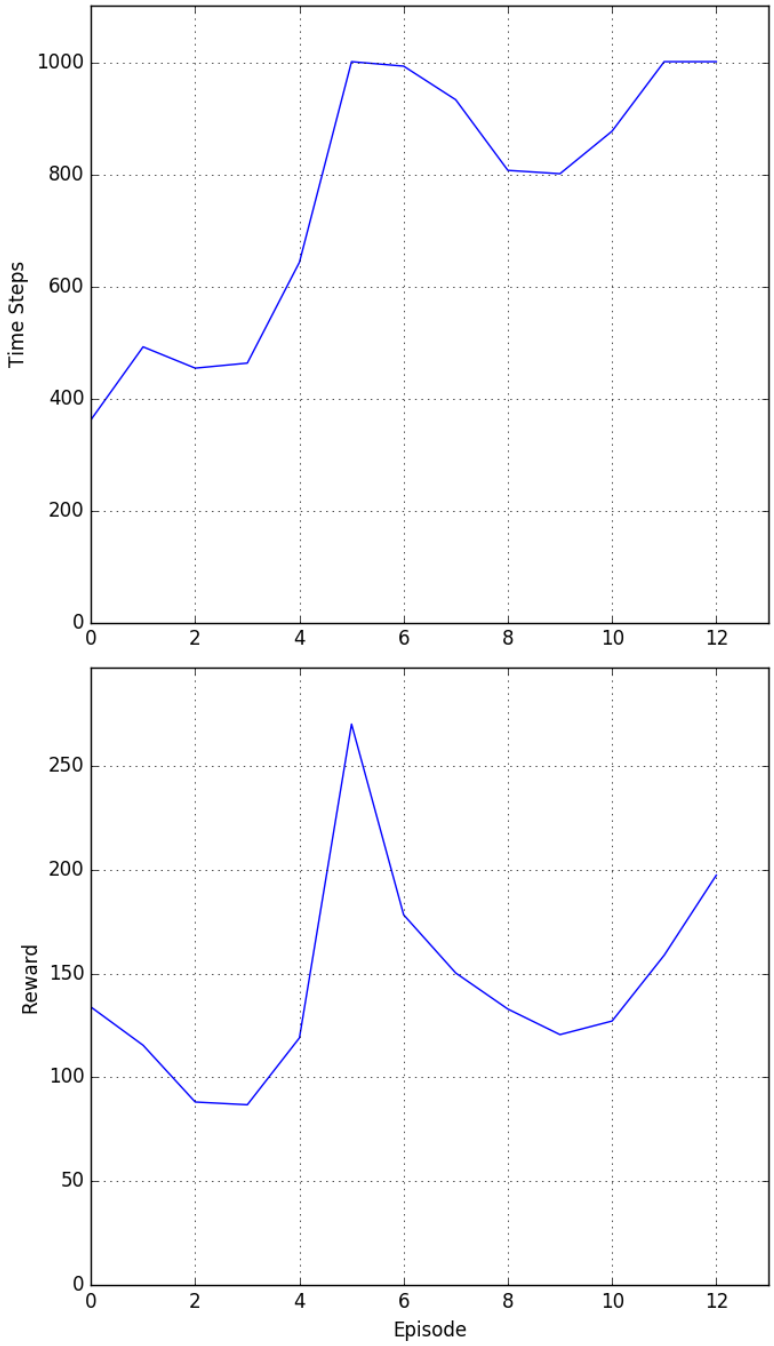


Figure 2: The top plot shows the time steps per episode and the bottom plot the cumulative reward per episode

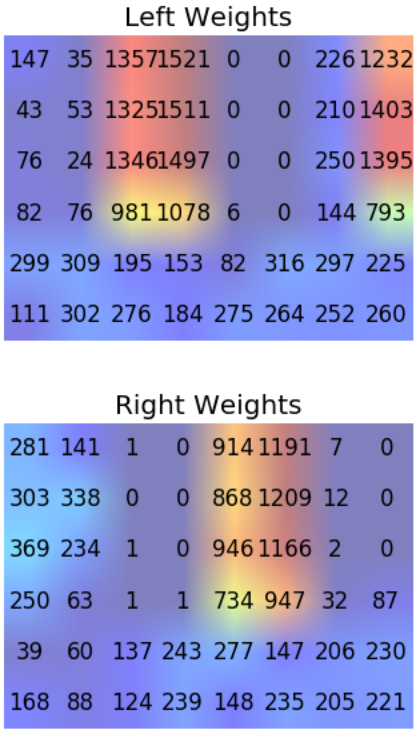


Figure 3: The top plot shows the total steps per episode and the bottom plot the cumulative reward per episode