Learning to Control Multi-Dimensional Autonomous Agents using Hebbian Learning

A Global Reward Approach

Ajdin Husić

Master of Science Thesis

Delft University of Technology

Delft Center for Systems and Control

Learning to Control Multi-Dimensional Autonomous Agents using Hebbian Learning

A Global Reward Approach

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Ajdin Husić

December 18, 2018

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

The novelty-raahn algorithm has been shown to effectively learn a desired behavior from raw inputs by connecting an autoencoder with a Hebbian network. Hebbian learning is compelling for its biological plausibility and simplicity. It changes the weight of a connection based only on the activations of neurons it connects, and can effectively reinforce good behaviors when combined with neuromodulation. These low-level synaptic weight changes make for a better merge of the three learning tasks of perception, prediction and action. However, the state-of-the art algorithm requires the design of a highly detailed modulation scheme designed for a specific system, which is disconnected from the overall objective it optimizes. In this thesis, we will propose that similar learning behavior can be achieved, by making the autonomous agent react to longer-term rewards, and thus implicitly introducing prediction capabilities. In doing so, the required modulation scheme becomes connected to the global optimization objective.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor Martijn Wisse for his assistance during my master's research. I would further like to thank Wouter Wolfslag for his useful ideas and feedback during my literature research.

Delft, University of Technology — Ajdin Husić
December 18, 2018

Chapter 1

Introduction

Robotic tasks are becoming more complex ever since the progressive developments in computational resources permitted the additional freedom of building more autonomy into the robotic agents. But unlike in industry, the environments in which these agents are employed are not static, and so the complexity grows further in that these agents require not only better hardware, but intelligence to maximally utilize its hardware potential. Ideally, agent intelligence allows not only the generalization of its behavior to unperceived situations, but also the optimal adaptation to it. Since such environments can be very complex (many sensor inputs, many actuator outputs and varying dynamics), the main challenge lies in the design of *scalable learning* algorithms.

An agent is hypothesized to exhibit intelligent behavior when they effectively minimize their prediction errors [3, 4]. It is straightforward that in order to optimize its **predictions** with respect to its **sensation** of the world, which is caused by its **actions**, it needs to accurately *learn to*:

- represent the state (sensory perception),
- predict the state given an action (prediction),
- control to achieve the state (action).

Interestingly, within these three tasks, we recognize three major types of learning. Namely unsupervised, supervised and reinforcement learning respectively. Of course, implementing an algorithm that combines these task must be **meaningful**, and so we must consider a few assumptions:

A.1 The agent features unknown (and uncertain) dynamics. Otherwise, there is no need for prediction, and the point of minimizing an error becomes futile.

A.2 The agent must sufficiently explore the unknown environment, in order to effectively learn the three tasks. If an autonomous agent does not excite sufficiently, it will most likely get stuck at a local equilibrium.

A.3 The algorithm must be applicable to a wide range of system classes, up to higher dimensional state and input spaces, from which follows that it must be not only computationally simple, but also scalable. Indeed, if we have high order complex and unknown system dynamics, learning must be efficient to be applied in either real-time learning, or episodic optimization tasks.

Hebbian learning is a learning type, compelling for its simplicity and biological plausibility. It straightforwardly changes the weight of a synapse based only on the activations of the neurons it connects and can effectively reinforce good behaviors and repel bad behaviors when combined with neuromodulation. Hebbian learning in general is scalable and computationally efficient, since the agent behavior is shaped through low-level neuromodulation as opposed to high-level value-function approximation as in other reinforcement learning types like Q-learning and SARSA. The biologically plausible and local learning rule, makes for a better merge of the three tasks.

The major obstacle to useful applications with Hebbian learning — which is that the performance of Hebbian plasticity is highly sensitive to the choice of inputs — has recently been solved by [1]. Their Novelty-raahn algorithm enables a neural network to **represent** accurately the features of the input domain by inserting a certain type of autoencoder that extrapolates higher level features. It is a real-time learning algorithm that makes the agent effectively learn a good **state representation**, combined with good **feedback control**. A *local* modulation term makes the agent highly adaptive to its environment. These attributes makes that their algorithm is currently the best in the eyes of the aforementioned assumptions and requirements.

The drawbacks of such method however, is that this algorithm still requires a considerable **design effort** to appropriately construct the modulation scheme, which demands detailed logic in order for the agent to learn correctly. The modulation term is also highly local for the current time tick in which the agent fares, and therefore the agent is never really learning to anticipate, i.e. the prediction part of the three learning tasks is missing.

We will propose in this thesis, that if we add a general global reward function from which the modulation scheme is derived, that it will enable the agent to regulate any type of control variable, and also regulate multiple control variables simultaneously. Moreover, with this novel method of deriving the modulation from global rewards, we will show that the agent can react to more distal rewards, and as such implicitly develop prediction capabilities. The utilization of a self-defined global reward function then gives more freedom to the engineers to decide how important each control aspect is. This generalized method poses the research question whether it can achieve similar results as the state-of-the-art algorithm.

Firstly, we will give an introduction to the Hebbian learning process in chapter 2 to familiarize the reader with its mathematical functionality. In chapter 3 we will present the proposal of how to use global rewards with Hebbian learning, and thereby yield a generalized version of novelty-Raahn. In chapter 5 we demonstrate that indeed the new algorithm can compete with the original novelty-Raahn algorithm while using a more complex set of dynamics, and prove its generality by applying the new algorithm on a cart-pole system, where we attempt to control an unstable equilibrium. Finally in chapter 7, we will conclude the thesis with our main findings in this research. Relevant code can further be found in appendix 7.

Chapter 2

Background

In this chapter we will explain briefly the idea of Hebbian learning, and how that idea translates into a mathematical formulation. We will then proceed to demonstrate the state-of-the-art literature and the use case for this algorithm as currently implemented. We will also show some experimental results done in literature and by myself, and finally argue about its drawbacks and propose how to improve this algorithm by eliminating these drawbacks.

2-1 The Hebbian Learning Process

Hebbian, or associative learning, is a theory about learning, hypothesized by psychologist Donald Hebb [5], that explains how biological neurons shape the learning process. The following statement is often quoted among neuroscientists to explain how Hebbian learning works:

> "When an axon of Cell A is near enough to excite a Cell B and repeatedly and persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells, such that A's efficiency, as one of the cells firing B, is increased." [5]

The idea is that any two cells that are repeatedly active at the same time, tend to become associated, so that activity in one facilitates activity in the other. This is often informally summarized as neurons that fire together, wire together [6]. Haykin [7] generalized this statement and rephrased it as a two-part rule:

1. If two neurons on either side of a synapse (connection) are activated simultaneously (i.e., synchronously), then the strength of that synapse is selectively increased.
2. If two neurons on either side of a synapse are activated asynchronously, then that synapse is selectively weakened or eliminated.

This is called a Hebbian synapse. In [7], Haykin explained this concept of synaptic weakening, where he included the processes of *synaptic enhancement* and *synaptic depression*. In this way, a Hebbian synapse is recognized to produce synaptic strengthening (enhancement) by positively correlated activity, or synaptic weakening (depression) by negatively or uncorrelated activity. Opposite from the Hebbian synapse is the anti-Hebbian synapse which weakens the synapse with positively correlating activity, whereas is strengthens the synapse with negatively correlating activity.

Hence, these statements spark an inspiration for the development of biologically plausible learning algorithms in optimization and artificial intelligence.

Altough Hebb did not provide a precise mathematical formulation of his postulate, a general form can be considered from the previous statements in a feedforward artificial neural network, when the weight between an input and output neuron is strengthened, when the input neuron causes a strong activation in the output neuron [8]. In that sense, the simplest form of Hebbian plasticity is described by **Hebb's hypothesis**, where the *plasticity* (change in weight) is given as

$$\Delta w_{ij} = \eta \cdot y_i x_j, \tag{2-1}$$

where η denotes a step size (or learning rate), and x_j denotes the activation of input neuron j, and y_i denotes the activation of output neuron i. This equation represents the standard mathematical form of a Hebbian synapse.

Of course this type of plasticity does not say anything about the neuronal dynamics themselves. If we consider for example that neurons can have activations in the range of real numbers, and their connections is a linear one, then Hebbian plasticity would suffer from exponential growth. That is, if presynaptic neuron fires strongly with the postsynaptic neuron, then their weight will grow strongly. This leads to an even higher value for the rate of weight change, driving the weight into saturation, after which selectivity is lost, i.e. no meaningful information will be stored in the synapse any longer. This is the sensitivity problem of Hebbian learning as mentioned before. I.e., if we feed a synapse too much of a same input, its weight will simply saturate, and this synaptic connection will hold only useless information in its memory.

To counteract this problem, a modulatory signal can be added to the learning rule, as is done by various methods in literature, see e.g. [9, 10] but also by [1] of which the algorithm is the basis for this thesis. This modulatory signal acts as a reinforcement-type feedback, and acts as a reward or punishment for an experienced input-output relationship, in order to develop the desired behavior. Hebbian plasticity then takes the following form:

$$\Delta w_{ij} = \eta \cdot m y_i x_j, \tag{2-2}$$

where m denotes the modulation in this scenario. Note that m can not only vary to change the amount of plasticity, but it can switch to the negative range, at which point anti-Hebbian (or unlearning) is turned on. Now, we can recognize that modulated Hebbian plasticity is in line with selectively reinforcing good behaviors as is the case with animals, where the modulation corresponds to dopamine in the brain. In we consider modulated Hebbian plasticity in a reinforcement learning framework, then the presynaptic activity is regarded as the agent's sensory perception y or state x, and the postsynaptic activity is equivalent to the agent's action u. This means that the Hebbian synapse is simply an adaptive feedback controller.

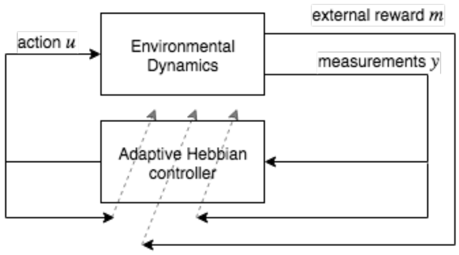

Figure 2-1: Block diagram of a general reinforcement learning framework using Hebbian learning.

For the purpose of consistency in our conventions, let us define the terminology and notations that goes unchanged for this entire thesis. The adaptive Hebbian feedback controller is a nonlinear combination of measurements y (perception) fed back as a signal u to its actuators (action). Generally, we have that

$$u_k = f(w_k y_k). \tag{2-3}$$

Here we will denote the subscript k as a discrete time-index. Then $y_k \in \mathbb{R}^n$ denotes the n-dimensional sensor inputs, $w_k \in \mathbb{R}^{m \times n}$ a matrix contain the plastic Hebbian weights and $u_k \in \mathbb{R}^m$ denotes the m-dimensional control outputs. The function $f(\cdot)$ is a nonlinear activation function. Hebbian adaptation is then generally of the form

$$w_{k+1} = w_k + \eta \cdot m_k u_k y_k^T, \tag{2-4}$$

where m_k is the scalar modulation signal, which is the same for each Hebbian connection. This modulation value assumes a measure of 'correctness' that the input-output sample (y_k, u_k) turned out to cause based on later observations ($k + 1$ in this thesis). A schematic overview of the closed loop structure is depicted in Figure 2-1.

The modulation scheme in this algorithm is the most important aspect for proper learning, as it directly changes the importance and sign of each training sample. But apart from the modulation term, good exploitation of the input space is still important. If, for example, the network gets excited with only *bad* experiences where the modulation is negative, the agent will proceed to unlearn everything it sees, and no viable progress may still occur. So to avoid feeding the agent with redundant and repetitive experiences, [1] includes an autoencoder, where the input space is represented by the autoencoder's higher-level features. In this way, the input as fed to the Hebbian synapses are more carefully selected, so that the entire input space of a given environment is better exploited.

2-2 The state-of-the-art implementation

In [1], the Novelty-Raahn algorithm is proposed, to solve the main obstacle of plain Hebbian learning. The novelty-Raahn algorithm enables an artificial neural network to **represent** accurately the features of the input domain (sensory perceptions) by extrapolating higher level features, based on the collection of novel agent experiences. These features of the autoencoder are fully connected with a second layer, which is considered the Hebbian **control** component of the agent. In the following subsections we will inform about the simulation environment in which the algorithm has been tested, and the functioning of the algorithm itself. Finally,

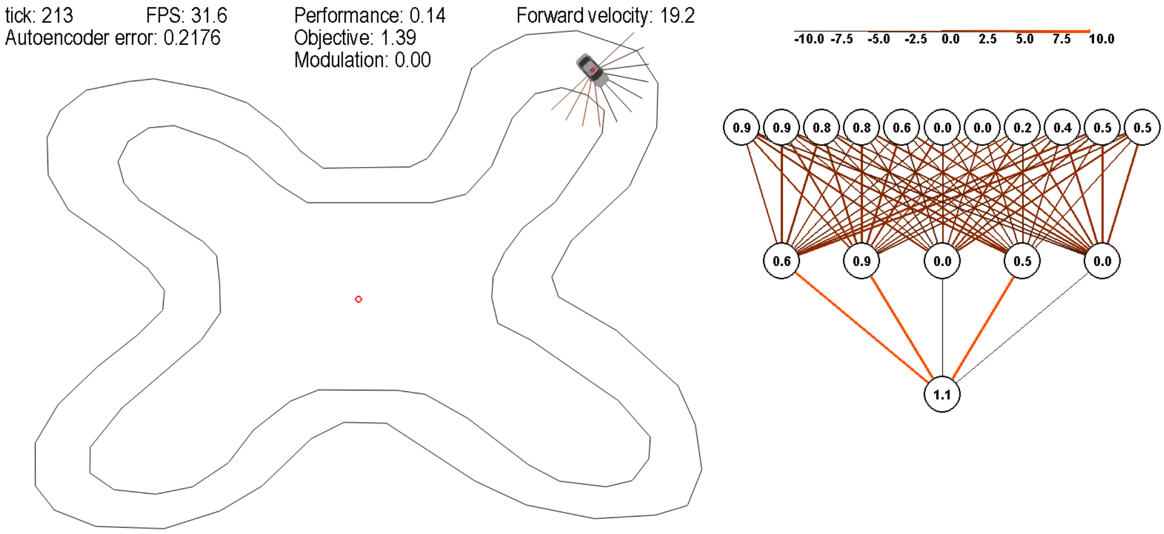

Figure 2-2: Graphical representation of the simulated environment as designed in Python: an autonomous agent traverses the lap shaped map. The autonomous car is required to make as many laps as possible in an allotted amount of time. The non-uniformly shaped map ensures that the agent does not simply repeat one behavior, but rather is required to generalize its policy to any input it experiences.

we will summarize some drawbacks still present, and propose what aspects need to change for improvement.

2-2-1 The environment

The *system* comprises both the *environment* and the *agent*. The environment entails the dynamics which govern the world's state, and the agent entails that which acts upon the world, to change its state, based on its perception of the world. The system in this description corresponds to a closed loop in feedback control.

Specifically, the simulated environment in [1] concerns an autonomous car agent, driving around through a static map in laps, see Figure 2-2. The entire track from the leftmost wall to the rightmost wall has a width of approximately 4120 units, whereas the lowermost wall up to the uppermost wall has a height of approximately 3074 units. The car has no dimensions, and is considered a point mass, which always drives at a constant speed of approximately 15 units[1] per time tick, unless it gets stuck into a wall in which case it stops moving forward. The agent can further issue a control signal between 0 an 1 (which may convolve with additional actuator noise), that determines the change in steering angle, or the car's global orientation θ. Note that if the car crashes into a wall, it may still escape after some time, because it is allowed to correct its steering. A control output of 1 changes the global orientation θ with $-2°$ per time tick (steers to the right) and a control output of 0 changes this orientation with $+2°$ per tick (steers to the left). The agent further has access to a set of eleven rangefinder sensors. These rangefinder sensors are line segments extending from the center of the car

[1] The true forward speed depends on its global orientation, and is equivalent to $\sqrt{15^2 \cos^2\theta + 12^2 \sin^2\theta}$.

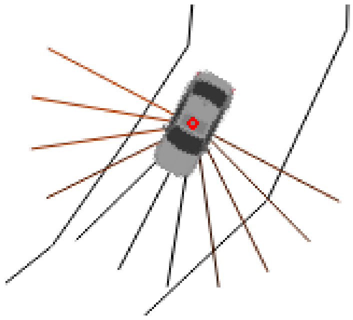

Figure 2-3: Depiction of the car's rangefinder sensors. A sensor's activation denotes a measure of wall intersection.

outward with each a length of 350 units, which sense the presence of walls (Figure 2-3). The first and last rangefinder sensors are separated with 180° from each other, and are oriented perpendicular to the car's facing direction. The other sensors have a spacing of 18° between them. Each rangefinder receives an activation between 0 and 1 at each time tick. A sensor value of 0 indicates no intersection with any wall, and an activation of 1 indicates a full intersection with a wall. Intermediate activations are linear interpolations of where a wall is sensed.

The agent uses the collaborate autoencoder and Hebbian layer as artificial neural network components, in order to learn the optimal control policy in real-time (figure 2-4). It first senses an experience $y_k \in [0,1]^{11}$ at discrete time index k, from which it computes the *perceived* state $\hat{x}_k = \sigma(W_k y_k)$, with $W \in \mathbb{R}^{5\times 11}$ a matrix containing the connections between the input and the perceived state $\hat{x}_k$. This perceived state, or high-level features, are in turn used to compute final control output $u_k = \sigma(w_k \hat{x}_k)$ that determines the change in turn angle of the car, where $w_k \in \mathbb{R}^{1\times 5}$ denotes the Hebbian connections. The nonlinear activation function $\sigma(\cdot)$ is the logistic sigmoid, and can be viewed as a mechanism that determines the activation of neurons in a more biologically plausible way (close to 0 is a non-firing or inactive neuron, whereas close to 1 is a firing or active neuron).

2-2-2 The autoencoder

Since the performance of Hebbian plasticity is given to be very sensitive to how each sample is treated during training, it is naturally important to feed the correct experiences representative of the input domain to the Hebbian plasticity. The functionality of the autoencoder in this sense is twofold. It learns to describe the underlying high-level features of the input space into its perceptive memory, which guarantees that these features are fruitful, given that it can produce a good reconstruction of the original input space. At the same time, this *encoded* representation makes the agent more exploratory, since it considers a more meaningful sense of inputs for its decision making. This part is important, because it solves the main obstacle of Hebbian plasticity, which is its high sensitivity to the choice of inputs.

In order to exploit this input space in an autonomous setting, the novelty-Raahn in [1] uses a *novelty buffer* to select only the most novel experiences for training. This novelty buffer prevents overfilling the perceptive memory with only the same kind of experiences, and hence prevents the agent from settling rather quickly in a bad state. For example, if the agent

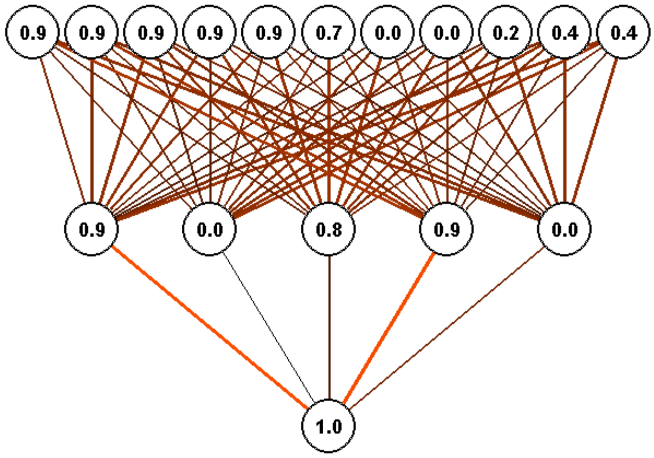

Figure 2-4: Artificial neural network as used in [1]. The network receives an activation y_k of input neurons (top part), from which the autoencoder (first layer) computes the high-level encoded features $\hat{x}_k$ in the hidden neurons (middle part). Finally, these features are used by the Hebbian network (second layer) to compute the control value u_k in the final output neuron (bottom part).

initially gets stuck into a wall, it may erroneously learn as if its state at that moment is representative of the entire input domain, and consequently remain in that state (it has reached a bad local optimum from which it cannot escape).

The novelty buffer contains at most 500 past experiences, where each experience is assigned a *novelty score*. To determine the novelty score of an arbitrary experience i, the Euclidean distance d_i is first computed between it and all the other experiences in the buffer,

$$\forall i \in \mathcal{N} : \quad d_i = ||y_i - y_l||_2 \quad \forall l \in \mathcal{N} \ /i, \tag{2-5}$$

with $\mathcal{N}$ the set of all experiences in the novelty buffer, y_i the activation vector corresponding to experience i. The novelty score n_i of experience i is then computed as the 20 smallest such distances. If the currently sensed experience y_k has a novelty score greater than that of the least novel experience in the novelty buffer $\mathcal{N}$, it replaces that experience in the buffer.

At each training tick, the autoencoder selects 20 random experiences from the novelty buffer $\mathcal{N}$ to train on. When training on experience y_i, the forward activation $\hat{x}_i$ is first computed.

$$\hat{x}_i = \sigma(W y_i), \tag{2-6}$$

where $\sigma(\cdot)$ is the logistic sigmoid activation function, and W a matrix containing the autoencoder weights. This forward activation is used to compute the backward activation (the *reconstruction* of the experience y_i).

$$\hat{y}_i = \sigma(W^T \hat{x}_i), \tag{2-7}$$

where the reconstruction error is computed as

$$e_i = y_i - \hat{y}_i. \tag{2-8}$$

Then, the *delta* of the inputs can be computed from this error, which will be used to computed backpropagated error.

$$\delta_y = e_i \odot \sigma'(\hat{y}_i), \tag{2-9}$$

where the $\odot$ is the component-wise multiplication, and $\sigma'(\cdot)$ is the logistic derivative. Now, the delta for the hidden features can be computed as

$$\delta_{\hat{x}} = (W\delta_y) \odot \sigma'(\hat{x}_i). \tag{2-10}$$

Now, using both the original error deltas δ_y and the backpropagated error deltas $\delta_{\hat{x}}$, we can compute the update of the tied autoencoder weights.

$$\Delta w = \alpha(\hat{x}_i \delta_y^T + \delta_{\hat{x}} y_i^T), \tag{2-11}$$

where α is a learning step size, set to 0.1, and all weights in W are clipped to the interval $[-w_{\max}, w_{\max}]$, with $w_{\max}$ set to 10, in order to prevent weight saturation.

2-2-3 The Hebbian layer

After having inferred the features of the environment through the autoencoder (perception), the agent decides which actions to take (control). The Hebbian layer in this sense can be viewed as an adaptive controller, as it adapts its plastic connections in real-time.

This component uses a modulated form of learning in order to achieve a desired behavior (control policy). This entails that learning is done through the modulated Hebbian learning rule as described in section 2-1. The presynaptic activity, in the case of the autoencoder, is equal to $\hat{x}_k$ at time tick k. And the postsynaptic activity is then just the control output u_k, which is first normalized through a linear transformation $u_{n,k} = 2u_k - 1$, in order to be in the range [-1, 1]. This removes any bias while learning, such that the synapses can learn on their own if the modulation were removed.

$$\Delta w_k = \eta m_k u_{n,k} \hat{x}_k^T + \xi, \quad \xi \sim \mathcal{U}(-0.1, 0, 1)^{1\times 5}, \tag{2-12}$$

where $\eta = 1.0$ is the learning step size, ξ is a vector of random samples drawn from a uniform distribution between -0.1 and 0.1 and m_k is the modulation term determining when and at what rate to use learning, unlearning or no learning. The modulation scheme in [1] always compares the the next state $k+1$ with the current state k, in order to determine how *good* the corresponding training sample (y_k, u_k) turned out to be, and influences all Hebbian connections using the same modulation.

The modulation scheme

The modulation scheme in [1] is defined as whether the car has turned toward a wall ($m_k < 0$, unlearning), turned away from a wall ($m_k > 0$, learning) or detecting no walls ($m = 0$, no learning). In order to measure the modulation, the authors have first defined a *modulation feeler*, an invisible line extending from the center of the car straight outward in the direction it is facing, i.e. with same global orientation θ, and with a maximum range of 400 units. This modulation feeler first senses the presence of walls, just like the rangefinder sensors, but rather than measuring the distance, it computes the angle β it makes with a certain wall. If the modulation feeler detects a different wall at the next time tick $k+1$ than it detected at the current tick k, than it computes the angle β with respect to the wall it detected at time k. If no wall was detected at time k then the corresponding modulation $m_k = 0$.

A modulation value m_k is thus only computed, when a particular wall has been detected at time k. The measure Δ of how much it turned away from this wall is thus computed at time tick $k+1$ as

$$\Delta = \begin{cases} \beta_{k+1} - \beta_k & , \text{ if } \beta_k > 90^\circ \\ \beta_k - \beta_{k+1} & , \text{ if } \beta_k \leq 90^\circ \end{cases} \tag{2-13}$$

Finally, the modulation m_k is computed as the amount in degrees it turned away from the wall normalized with the maximum turning speed per tick, such that the modulation yields a value in the domain of [-1, 1].

$$m_k = \Delta / \Theta_{\max}. \tag{2-14}$$

Here, $\Theta_{\max}$ denotes the maximum turning speed that can occur between two given time ticks, i.e. $\max(\theta_{k+1} - \theta_k)$.

Finally, the general algorithm is summarized in pseudocode 1.

Algorithm 1 the novelty-RAAHN algorithm. This methods assumes:

1. multi-dimensional input space
2. modulation scheme needs be well-defined
3. unknown system dynamics

procedure INITIALIZATION
  Initialize Autoencoder weights $W_0 \leftarrow \mathcal{N}(0,1) \in \mathbb{R}^{m \times n}$
  Initialize Hebbian weights $w_0 \leftarrow \mathcal{N}(0,1) \in \mathbb{R}^{m}$
  Initialize novelty-buffer size $n_{novelty}$ arbitrarily
  Initialize number of training samples per tick $n_{samples}$
end procedure
procedure TRAINING(for every time tick k do)
  take action u_k using policy derived from W_k, w_k
  observe measurements y_{k+1} and reward m_{k+1}
  update novelty buffer and weights W_{k+1}, w_{k+1}
end procedure
Output: learned agent weights W, w

2-2-4 Experiments and results

In the experiments in [1], the simulation is run 200 times for 10,000 ticks each time where an agent with novelty-Raahn is compared against an agent with only a Hebbian network (i.e. without autoencoder). Completing a circle around the center point as shown with the red dot in Figure 2-2, denotes completing one lap.

The results suggest that novelty-Raahn completes 8.8 laps on average, whereas Hebbian alone completes 9.2 laps. Novelty-Raahn performs thus slightly below the lone Hebbian controller, while requiring extra time in the beginning to acquire a novel set of experiences. Also, considering that novelty-Raahn learns both features and the control policy at the same time, it is likely close to performing as well as possible for such a method.

The authors have furthermore shown that novelty-Raahn performs better when degrading the quality of the sensors to a certain extent. They show that if the length of the rangefinders increases, which makes distinguishing different situations more difficult, novelty-Raahn experiences fewer failures in completing laps than Hebbian, because the Hebbian agent is forced to learn from these degraded inputs, whereas novelty-Raahn learns a new representation.

Additionally, I have performed an experiment with Hebbian and novelty-Raahn myself. The claim is that when training a larger network, the agent needs a significantly longer time to learn, as it learns both perception and action at the same time. This reasoning prompted me to do a different experiment, in which both type of algorithms are run first in a training phase for 12,000 time ticks, with the idea that this is enough time for the agents to reach their near-optimal potential. After this phase, the state of the agents is reset, and they are released to follow their learned controller for 10,000 time ticks without training (evaluation phase). The performance in this scenario is only counted in how many laps each agent makes in this evaluation phase. This process is repeated for both algorithms 10 times, which results in a low enough variance to conclude from it their true performances. Namely, the Hebbian agent performed in the range [9.82, 9.92] with an average performance of 9.86 completed laps. Novelty-Raahn on the other hand performed in the range [9.81-10.04] with an average of 9.91 completed laps. This suggest that after sufficient training, novelty-Raahn even overtakes pure Hebbian slightly in its performance.

2-2-5 Current drawbacks

The main issue of the novelty-Raahn algorithm is the **modulation scheme** as currently defined. Firstly, it is only defined for the environment as described in [1], i.e. as in subsection 2-2-1. A closer look at this system, which is relatively simple and static in nature (the entire network only generates the change in steering angle), suggests that achieving the desired behavior requires complex modulation logic. For this system, this means that designing this logic is almost as easy (if not harder) as guessing the weights of the 11-by-1 Hebbian component, since these weights approach rather intuitive steady-state values: using only the Hebbian component without autoencoder results in weights that simply compensate for sensing walls in a particular direction. If such 'simple' agent behavior requires such detailed modulation logic, then we can ask ourselves whether even the slightest changes in the environment will not render the algorithm useless. The algorithm as is, thus **lacks the desired freedom** for the engineer to determine the desired tasks that the agent needs to achieve.

Moreover, the modulation scheme as defined is a **time-local** reward, and therefore the agent develops a short term memory. Rather than just anticipating one step ahead at most, complex autonomous agents may require long-term connections, that enable a more optimal control policy.

Most importantly, the odd observation of this algorithm is that the modulation logic is not explicitly **related** to the **global objective** which is sought to be optimized by the algorithm. In this way, the algorithm suffers from requiring hard-coded logic, which is **not directly** available. Whereas the **directly** available global objective of the system is not used, i.e. the engineers know what they want to make the agent ultimately achieve, but need an intermediate step to reach this achievement.

2-2-6 General proposal

In order to eradicate these difficulties, and thus make the algorithm more useful overall, I will propose in this thesis that the main contribution lies in changing the way in which this modulation scheme is defined. More specifically, **I propose that the modulation scheme should be defined by relating the global objective function explicitly to how each Hebbian input-output training sample should be modulated**.

Assuming that we can successfully achieve the goal in such proposal, we can further recognize some additional advantages: Namely, if we find such a relationship, we can use this global objective to train **any** type of output we wish to control. For example, if we now are to add a control output that sets the agent's acceleration, we can use this more general relationship between how the global objective function should be treated with respect to the modulation of each individual sample, rather than requiring some different detailed modulation logic yet again. Since the change in velocity physically represents something different than the change in steering angle, and suddenly we need not contemplate how this new control variable precisely affects the current modulation behavior.

Furthermore, achieving such a relationship allows for the agent to develop a **long term memory** as well. The idea is that if the algorithm allows for global objective functions, then such an objective function can be one which incorporates the average performance of the agent over multiple time steps in the future. Coincidentally, this means that the agent will implicitly assume **prediction capabilities**. Therefore, we are practically also merging the existing tasks of perception and action, together with prediction, as we have initially considered to be required for prediction error minimization.

Chapter 3

Proposal of the Thesis

The current proposal as put forth in the past section, is to find a relationship between the global objective function given a certain task, and the modulation applied to each training sample in the Hebbian network. In order to find the best conclusion, we must first recall the fundamental function of Hebbian connections, and how modulation affects its plasticity. In this chapter, we will consider the goal of a global reward function and its effect on the behavior of an agent. Subsequently, we will relate one with another based on how they affect the system to achieve its goal, and finally we will conclude the chapter with the obtained answer materialized from the given proposal.

3-1 Defining the general global reward

Since we are working in a reinforcement learning setting, we will define the global objective function, which is to be maximized, the *global reward* function R[1]. Specifically, the *particular* reward function, or otherwise the immediate reward r_k, obtained at discrete time tick k from the environment, is a measure of how 'good' the current *time-local* input-output or measurement-action sample (y_k, u_k) is. The global *long-term* reward function R_k^N, which anticipates a horizon of N discrete time steps ahead, is simply the average of each particular reward over this horizon.

$$R_k^N = \frac{1}{N} \sum_{i=k}^{k+N-1} r_i. \tag{3-1}$$

The general goal of reinforcement learning is to optimize, specifically maximize, this function R_k^N. The parameters on which R_k^N (indirectly) depends, i.e. the neural network weights w_k, are adjusted in such a manner, that R_k^N increases overall.

[1] Rather than a *fitness* function, which is directly related to the parameters (weights) on which said function depends, we call it the *reward* function because it is directly related to the effect of a state-action pair.

3-2 Relating the global reward to modulated Hebbian learning

The hypothesis of Hebbian learning, is that it learns the behavior the agent experiences. That is, the presynaptic activity (sensor input) y results in postsynaptic activity (control output) u trough the synaptic connection w, and updates w by Hebbian learning $\Delta w = uy^T$, in such a manner that input y will more often produce output u. Using this hypothesis, we find that we can modulate said synaptic plasticity by $\Delta w = muy^T$, based on the desired result: If we want that y produces result u, then modulate the training sample positively $m > 0$ such that behavior u, given y gets learned. If we want that y produces result u no longer, then modulate the training sample negatively $m < 0$, such that behavior u given y gets unlearned. If we are satisfied with current behavior, do not learn at all, and modulate zero $m = 0$, such that there will be no change in behavior.

This is a pretty powerful hypothesis, in that it can effectively reinforce desired behavior. If we now consider the global reward function R_k^N based on a given sample[2] (y_k, u_k) then we identify three possible scenarios at each time step. That is, with respect to the previous time step $k-1$, the reward R_k^N has

- increased (positive modulation),
- decreased (negative modulation),
- stayed the same (zero modulation).

From this we can infer that modulation can be related as

$$m_k^N = g(R_{k+1}^N - R_k^N), \tag{3-2}$$

with $g(\cdot)$ a nonlinear weight regularization function. Since R_k^N is defined as the future average of particular rewards r (equation 3-1), we can make the simplification

$$\begin{aligned} m_k^N &= g\left(\frac{1}{N}\left(\sum_{i=k+1}^{k+N} r_i - \sum_{i=k}^{k+N-1} r_i\right)\right), \\ m_k^N &= g\left(\frac{1}{N}(r_{k+N} - r_k)\right). \end{aligned} \tag{3-3}$$

Note that, since we use an arbitrary time horizon N, we have to wait N discrete time steps more with respect to the inference at k, before we update the weight corresponding to that time sample, i.e.

$$\begin{aligned} w_{k+1} &= w_k + m_k y_k u_k^T \\ w_{k+1} &= w_k + g\left(\frac{1}{N}[r_{k+N} - r_k]\right) y_k u_k^T. \end{aligned} \tag{3-4}$$

The idea is that with this generalized form of modulation, the global reward function subject to the environmental dynamics will be maximized, assuming that the global reward is strictly increasing wen a desired change in behavior occurs, and decreasing when an undesired change

[2] The input here is denoted as y_k, but may be an inferred state $\hat{x}_k$ depending on any intermediate filters. The input in this sense is just what gets fed into the Hebbian network.

in behavior occurs between each time step. This assumption needs to be satisfied to learn properly a desired control policy, because of the sensitivity of Hebbian plasticity. But it does not imply that defining a global reward with this algorithm is more prone to failure than any other algorithm that incorporates objective functions. Rather, I would say that any other control method that utilizes explicit objective functions needs to be properly defined too.

3-3 Chapter conclusion: proposal of the thesis

Based upon the previous chapters we have arrived at the following thesis proposal:

Proposal 3-3.1: Thesis proposal

Given a *long term global* reward $R_k^N = \sum_{i=k}^{k+N-1} r_i$, with r_k the *particular* reward, the modulation scheme can appropriately be defined as a function of increment of R_{k+1}^N in the next time step $k+1$ with respect to the current reward R_k^N. Or equivalently, the modulation m_k^N that is assigned to input-output sample (y_k, u_k) is equal to $g(\frac{1}{N}(r_{k+N} - r_k))$.

Proposal 3-3.1 goes hand in hand with the following assumption:

A.4 The global reward R_k^N increases if the environmental state changes desirably and decreases if the environmental state changes undesirably between time ticks k and $k+1$.

In the next section we will set up the exact modulation scheme as used for the the experimental environment in 2-2-1.

3-3-1 application and use-case

If we consider the environment as in 2-2-1, we remark that the true performance of the autonomous car is that in an allotted amount of time the car must create as many laps as possible around the center point. However, we cannot directly use this performance as our global reward R, since the map is shaped in such a way that at many places the agent must first decrease the performance, in order to increase the overall performance later on, like driving around a turn that goes a bit outward. So we must first make an adjustment in choosing our appropriate global reward. Given that the first experiment concerns only to correctly learn steering, we can initially use a particular reward r_k not dissimilar to [1]. If we define $d_k = 1 - y_k$ as distances[3] to objects, then we can say that desirable behavior occurs when the car's front-pointing sensor (6th out of 11 sensors), detects an increase in distance. That is, we define the particular reward r_k as

$$r_k = 1 - y_k(6), \tag{3-5}$$

[3]The 1 in this definition is not strictly necessary, since the algorithm cares about increments of rewards only, but is more intuitive as the input domain now physically represents normalized distances.

where $y_k(6)$ denotes the sixth element out of the 11 rangefinder sensor activations in y_k. A good regularization function g as presented in equation 3-3 is the tangent hyperbolic function

$$g(x) = \tanh(\gamma x), \tag{3-6}$$

where γ is just a positive gain constant. The reason for regularization is that the if rewards change too radically, the weights do not get too large updates, as the tangent hyperbolic functions maps to a value between -1 and 1 (just like the original modulation scheme). Moreover, the tangent hyperbolic function is nearly linear when its argument is close to zero. The gain function γ on the other hand is used to stretch out the weight updates if these are too small. A good value of γ generally is when the modulation becomes approximately 1 with the maximum possible improvement, and -1 with the maximum possible deterioration in behavior.

Chapter 4

Implementation

Before showing the final results in the following parts of this thesis, we will first cover the details of the entire implementation as necessary to generate the subsequent experiments. The original work of [1] was first recreated in Python. This was done mainly because of my own higher competence in Python rather than the original language, which was written in C#. This allowed me to very easily make any additions to the program without further investigation. Also, rewriting the code was crucial in understanding the exact mechanics of the entire simulation, which in turn allowed for further improvements, and helped me achieve the new additions to the existing algorithm.

4-1 Implementing the environment for the autonomous car

The first step of implementation was to create exactly the discrete environment as developed by [1] in Python3, in order to enable stepping through it by giving an control action u at each particular state. This environment consists of the classes `Car.py`, `Wall.py`, `rangefinder.py` and `configs.py`. See appendix -1-1 for the source code. Both the Wall and rangefinder class inherit from a lower-level LineSegment class (`lines.py`). Any line segment object (instance from LineSegment) has the ability to firstly test if it intersects any other line segments present using the AABB (Axis Aligned Bounding Box) method. Secondly, if two such line objects do intersect, then intersections are computed using simple linear line intersection math. An AABB collision is a collision detection method in computer graphics, in which two rectangular shapes collide if they overlap in all dimension axes (axis-aligned). See Figure 4-1 for an example. The same logic can be applied for line segments, in which the upper and lower bounds of said segment is simply seen as the rectangle boundaries in AABB. Of course when using line segments, their AABB collision does not guarantee an actual intersection, so intersection math is still necessary not only to check whether intersections are present between line segments, but also at what point they occur. Skipping the AABB detection part is theoretically possible but not optimal in simulation, because of the considerable difference in computational load, especially when testing multiple line segments against many other line segments.

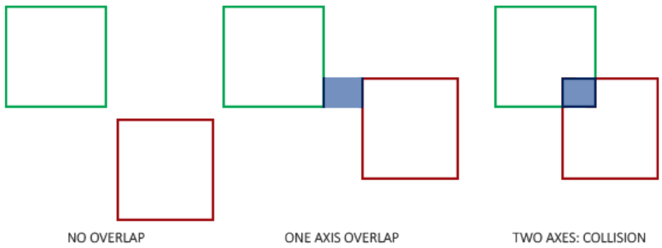

Figure 4-1: Example showcasing AABB collision in 2D. Collision between two rectangular objects only occurs if they overlap in all dimension axes. Adopted from [2].

Furthermore, a line segment object is able to compute the exact intersection coordinates between itself and any other arbitrary line segment. This concept is extended in the rangefinder class, where its objects have additional attributes like a default length, a current activation value y_i, and a car instance to which it is attached. Each instance of rangefinder updates its positioning dynamically, along with its host car, of which its center point corresponds to the rangefinder's starting position. The end position is then related to a fixed relative angle between the car and itself, and its default length. A rangefinder line segment i has the additional ability to sense the value of its activation y_i by first computing the exact intersection point, and simply calculate the distance from its end position to this intersection, divided by its default length.

Whenever the simulation is initialized, first the car and and all the map walls are spawned. This is done from the file `configs.py`. The map configuration is stored in an xml file `XMap.xml`, which contains both the car starting position (1561.93, 505.79), and all the wall coordinates (an (X,Y) coordinate and a relative distance in x,y-direction). Now the simulation can step through the environment by receiving an control output u in the value [-0.1,1.1]. The car object is programmed initially to move with constant speed, but only change its global orientation θ by adding $4u - 2$ degrees to it, so that the increment yields a value in the range [-2.2, 2.2] degrees. The velocity vector is computed as

$$v_k = \left(s_x \cos\theta,\ \frac{4}{3} s_x \sin\theta \right), \tag{4-1}$$

where $s_x = 15.0$ is the constant speed component in the initial experiments. Later on, I added the dynamic speed and acceleration control variable of the car, such that an environmental step accepts the value $u = (u_\theta, u_a)$. Here the control values change the global orientation θ and car's speed s_x respectively. The control values in u are both expected to be given in the range [-0.1, 1.1]. The control value u_a changes the car's speed attribute by incrementing it with $1.16u_a - 0.08$ so that the increment in speed is in the range [-0.08, 0.08] units. The car's speed is bounded to the upper limits of (15.0, 12.0) (in x and y direction resp.) and the lower limits of (10.0, 8.0). This is done to prevent indefinite increase or decrease of speed, and so that the car cannot outperform its previous version by simply driving faster (the upper speed limit is also the starting speed in case of no acceleration).

The environment as described here is used in the following experiments, and in case of constant speed experiments the acceleration control variable u_a is simply set to 0.5 (constant speed).

4-2 Implementing the agent for the autonomous car

The initial version of the autonomous agent is simply a recreation of the existing agent by [1]. The main components are `net3.py`, `modulation.py`, and `agent_functions.py`. A network object (from class `net3`) defines the nonlinear relationship between an arbitrary observation it receives as input, and the control output mapped from this observation. I.e. the network is built to describe exactly the equations as described in the neuronal dynamics in chapter 2. It consists of two layers (an autoencoder layer, and a Hebbian layer) and produces an optimal control policy by interacting with the environment. The network interacts with the environment in two ways. At every time tick k it samples the observed experience y_k (the rangefinder activations) to output the control u_k and send it back to the car, and secondly it adapts its network parameters through sensing a modulation according to the `modulation.py` class. This is done as described in section 2-2-3. Later on, I added the `performance.py` class with the addition of global rewards. The modulation in that case is derived from these global rewards as proposed in chapter 3. Training of the neural network is done through the `agent_functions.py` module, which contains both the autoencoder training process, and the Hebbian learning process as training options. One ambiguity I had found in [1] is the fact that Hebbian learning was actually done by first normalizing the postsynaptic activition. I.e. u_k, the real postsynaptic value was one between 0 and 1, whereas the postsynaptic activation as used during training is first normalized to a range between -1 and 1 to represent the measure of increment (or decrement) in the car's orientation. A detail not precisely mentioned in [1], but actually crucial for proper learning.

After having implemented the original works of [1] in Python, I proceeded with the additional experiments as described in section 2-2-4. These are experiments where the original algorithm is compared with and without autoencoder. Subsequently I tested the new algorithm with my own contribution where global rewards have been introduced. The corresponding particular reward r_k in this experiment was defined somewhat intuitively, where we gave relatively high importance to averting walls, and relatively low importance to increasing speed. That is, this particular reward (particular to each world state) is just a weighted average of several objectives that the agent needs to learn simultaneously (both steering and accelerating). The exact relative weights for these objectives are then tunable, based on what the engineer finds important. From r_k the global reward R_k^N is then constructed according equation 3-1 in `performance.py`, and the corresponding modulation is the difference of two consecutive global rewards as per equation 3-3. This modulation measurement is defined in `modulation.py` and is fed in an N-step delayed fashion to the Hebbian layer's training method in `training.py`. Finally, fundamental experiments are done as shown in chapter 5 to prove the new method works. The final algorithm is shown in pseudocode 2, where the main difference is the generalizing addition of global rewards.

4-3 Implementing the cart-pole environment

The cart-pole environment as used in the later experiments, is a gym-environment from python OpenAI [11]. The graphical representation of the environment is found in Figure 5-4. It concerns a pole attached by an un-actuated joint to the cart, which moves along a horizontal plane. At each moment in the environment the full four-dimensional state x is available to

Algorithm 2 the Hebbian learning with global rewards method. This methods assumes:

1. multi-dimensional input space
2. a global reward function based on the agent's task
3. unknown system dynamics

procedure INITIALIZATION
  Initialize Autoencoder weights $W_0 \leftarrow \mathcal{N}(0,1) \in \mathbb{R}^{m \times n}$
  Initialize Hebbian weights $w_0 \leftarrow \mathcal{N}(0,1) \in \mathbb{R}^m$
  Initialize novelty-buffer size $n_{novelty}$ arbitrarily
  Initialize number of training samples per tick $n_{samples}$
end procedure
procedure TRAINING(for every time tick k do)
  take action u_k using policy derived from W_k, w_k
  observe measurements y_{k+1} and global reward R^N_{k+1}
  compute generalized modulation $m_k \leftarrow R^N_{k+1} - R^N_k$
  update novelty buffer and weights W_{k+1}, w_{k+1}
end procedure
Output: learned agent weights W, w

Table 4-1: Summary of the observation and action spaces of the cart-pole environment. The real-valued fully observable state is shown in the left part, and the discrete-valued one-dimensional action space is shown in the right part.

Observations				Actions	
Nr	**State**	**Min**	**Max**	**Control Action**	**Discrete Value**
1	Cart Position	-4.8	4.8	Right Force	0
2	Cart Velocity	$-\infty$	∞	Left Force	1
3	Pole Angle	-24	24		
4	Angular Velocity	$-\infty$	∞		

the agent. This state is a sensor observation of the cart position, cart velocity, pole angle and pole velocity at the tip respectively. And a 'zero' state corresponds to an upright position of the pendulum. When an episode starts, the pendulum starts upright with a minor random variation for each state variable, sampled uniformly from a distribution in [-0.05, 0.05]. The control action applied onto the environment is a force to the right (1) or an equivalent force to the left (0). The environment is forced to receive either of the control values and may not apply zero force. A neat overview of the full observation space and action space is summarized in table 4-1.

The environmental time evolution occurs in episodes. The state initializes at the start of each episode as described above, and simulates a sequence of states based on the control outputs it receives. The episode is terminated when the pole angle deviates more than 15 degrees from its equilibrium state (the upright position), or when the cart moves more than 2.4 units from its center position. If the pole remains alive (balanced) for at least 200 time ticks, the episode terminates due to a successful episode. According to [11], cart-pole is considered solved when it remains alive for 195 or more time ticks on average for a duration of 100 successive episodes. That same goal will be used for our purposes.

4-4 Implementing the cart-pole agent

The code to implement the agent can be reused from the autonomous car environment. The only differences will be in hyper parameters (structure, objective function, etc.). The main noticeable difference with the cart-pole compared to the autonomous car, is the fact that the sensor inputs measure values in the range of real numbers. So the autoencoder cannot properly learn if the state is not pre-processed (assuming that the nonlinearities are sigmoids still). Another obvious difference is the type of sensor input. Rather than high-level environmental features, the agent receives the physical state that is part of describing its own dynamics. For this reason, an autoencoder is generally not necessary, and might only pollute the network with needless additional information.

In experimentation, two different structures of the agent network are used. First, the exact same network structure as in the autonomous car example, to show that it does not work because of violation of the initial assumptions. Secondly, a different structure with multiple layers and different nonlinear activation functions will be used, such that the controller consists of more complexity, which is required for the proper policy.

All the implementation code is found in `cartpole.py` in appendix 7.

Chapter 5

Experiments

In this chapter, we will demonstrate fundamental experiments using the new generalized modulation rule, and report its results. I have reproduced the original work of [1] in a Python simulation, in which I have validated their method (Figure 2-2). Additionally I have added experiments of my own which will be discussed in this chapter (see source code in appendix 7). In the first section I will demonstrate experiments using the autonomous car environment, where we have made changes that make the system more complex. Showing that the new method can indeed learn to control this environment is the most important contribution of this paper. In the second section we will attempt to demonstrate the generality of this method, by applying it on a cart-pole system with inverted pendulum.

5-1 Experiments with the autonomous car environment

Firstly, let us introduce into the existing dynamics an additional control variable that sets the acceleration of the car. Now $u_k \in \mathbb{R}^2$, where $u_k(2)$ is the second element of the vector u_k and which corresponds to the change of speed of the car. The control output $u_k(2)$ is first linearly mapped to the range [-0.08, 0.08] to denote the change in car speed. The constant bounds of 0.08 were chosen such that the dynamics of accelerating look realistic relative to the car's speed. The precise choice of this value matters not too much in showing that the new method can regulate multiple control variables. The speed however, is capped to a maximum value of its original forward speed in order to prevent indefinite increase of speed. In this way, the new method is not allowed to outperform the old algorithm by simply driving at a faster speed. On the other hand, the lower limit of speed is set to approximately 12 units per tick, to prevent the car from a complete standstill. Also, the car's speed is set to its minimum value once it crashes into a wall, which additionally causes a sudden decrease in reward. Since the car is allowed to drive slower than its maximum speed, it acts as a disturbance to the overall training process as now it needs to learn both control variables simultaneously. Hence, we will demonstrate that even though both training and the dynamics are now more complex, the method is still able to learn reliably the optimal control policy.

Given the new dynamics of the system, we will choose the particular reward r_k to be equivalent to a weighted sum of the front-pointing distance d_k and the absolute speed s_k of the car.

$$r_k = 1.3 \cdot d_k + 0.3 \cdot s_k. \tag{5-1}$$

The precise relative weights are designed by tuning, but are set initially to match an intuitive understanding of the tasks that the agent needs to achieve. We can argue that in order to maximize the amount of laps the car completes, averting walls is indeed the most important aspect. So the larger relative importance of 1.3 is given to the sensor distances, whereas acceleration is only required if the agent is certain that the car will not crash. Hence the smaller value of 0.3 corresponds to the reward contribution of the speed, which was found to work well.

We furthermore chose a time horizon of $N = 5$ steps ahead, and $\gamma = 2$. Taking a too large value for the anticipation horizon N may lead to loss of causality. That is, the agent may seek to reward an action too far in history that had no significant effect on the current world state. Of course, a value as low as 5 time steps anticipation reduces the quality of conclusion about the actual prediction capabilities the car has. As the sampling rate of an arbitrary system gets faster, the less impact a delay will have on that system, since the measurements will be closer to each other in value. Nevertheless the tuned value $N = 5$ worked well in the case of the autonomous car, and it does not take away from the quality of conclusion about the generality of the method.

Next, to encourage exploitation of the input space more, we will introduce episodic learning, in which the environment's state gets reset after a certain amount of time ticks. At each reset, the agent's network keeps in memory, such that it keeps improving its policy. Specifically, we will set the agent to train for 15 episodes, where each episode lasts for 1200 simulation time ticks.

5-1-1 Convergence test

If we consider the true global optimum, as permitted by the environmental dynamics, where the maximum absolute speed s_k equals 19.21 (Euclidean distance of 15 and 12 units per tick), and the maximum distance (1.0 in case of no wall intersection), then we find $\max r_k \approx 6.96$. If we now collect the average values of the global rewards per episode, we find a convergence behavior as shown in Figure 5-1. From this figure we can deduce that indeed the global reward function R_k^N is optimized by the new modulation scheme, as the reward value converges to its optimum of 6.9.

5-1-2 Performance test

In the following experiment, we will demonstrate that the new method can achieve a similar performance as the existing algorithms. First, let us determine the performance range, in which the algorithms can be considered to perform similarly, i.e. near optimal. Past simulations have revealed that a *good* agent is able to complete anywhere between 0.9 up to approximately 1.0 laps in a span of 1000 time ticks. For example, both pure Hebbian and the novelty-Raahn algorithm performed close to 10 laps (at least 9.8 in all cases) in the evaluated

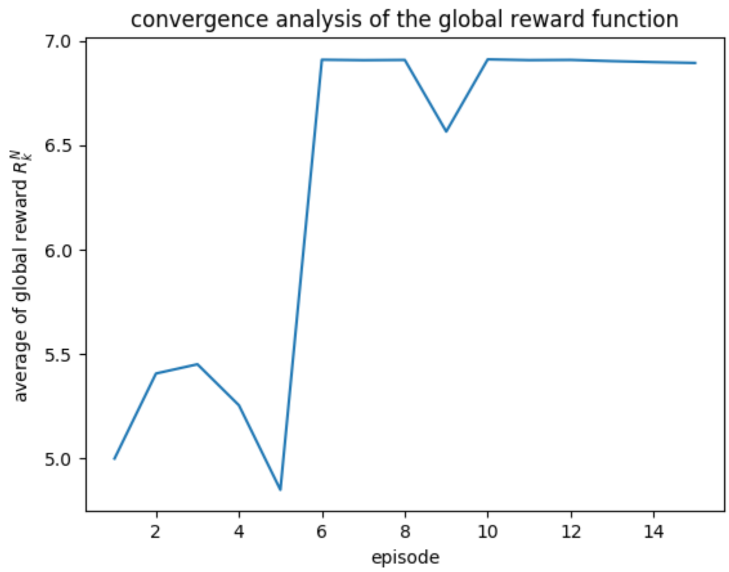

Figure 5-1: Graph of the average global reward occurring at each episode.

10,000 time ticks as tested in section 2-2-4. To define a more strict minimum boundary of when an agent can be considered near-optimal (i.e. satisfactory performance), we will determine an additional metric that measures this. That is, the agent behavior is considered unsatisfactory if it ever crashes. Between crashing and no crashing at all is clear difference in overall performance, because a crash may occupy the car at the same location for a significantly longer time, before it escapes. To test this metric, and hence find the minimum satisfactory performance, we will first run an experiment where the number of crashes is tracked for each algorithm.

The set-up of the experiment is as follows. We will again use episodic learning, where the goal is to achieve zero crashes in the final episode. Each episode takes at least 2500 time ticks, to ensure enough training time. If the car has crashed within the initial period of 2500 ticks, the episode will terminate once the 2500 ticks have passed. If the car crashes later than 2500 ticks, the episode terminates immediately. If no crashes occur at all during said episode for 10,000 time ticks, then learning will terminate for the current algorithm, and the final performance of the episode (corresponding to the final 10,000 time ticks) will be recorded. The number of crashes for each episode is recorded as well, which will be shown in a graph. The entire training experiment is repeated 5 times for all three algorithms, i.e. for pure Hebbian, novelty-Raahn and the novelty-Raahn structure in combination with the global rewards as defined in equation 5-1 (called *global reward*). Afterward, the average number of crashes along the 5 runs is computed for each episode and plotted in Figure 5-2.

We can conclude from the crash experiment that indeed the global reward method learns eventually to not crash along with the other algorithms. Expectedly, the global reward method takes longer to learn, since we train it on two control variables simultaneously, and so the network becomes more complex as well.

Furthermore, we find that indeed we have a similar performance when using global rewards. Namely, the minimum performance of novelty-Raahn and pure Hebbian without crashing is

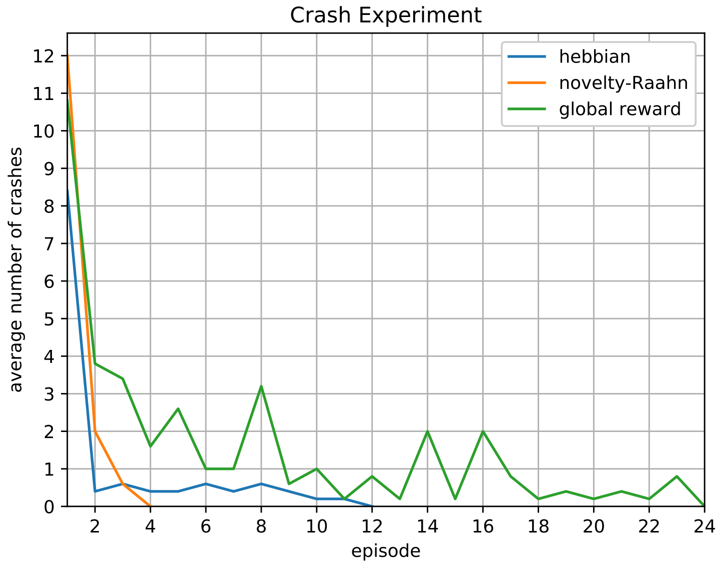

Figure 5-2: Average number of crashes for each algorithm during training. From this figure we can conclude firstly that the global reward method learns the optimal behavior of not crashing along with the other algorithms shown. Secondly, global reward has indeed learned a similar performance, because its final performance is in the range of optimal performances as derived from the other algorithms when they do not crash (not shown in this figure).

reached with a final performance of 9.64. With these methods, the car is forced to always drive at maximum speed, so we can use this value as the minimum satisfactory threshold. Now we can see that similarly, the global reward method achieves a performance of 9.73 on average, which is well above the satisfactory threshold, and from this we conclude that the global reward method can indeed achieve similar performance as the state-of-the-art algorithms while training multiple control variables simultaneously. All the final performances are compared in box plots as shown in Figure 5-3. If we observe the learning process visually, then we see that the autonomous car agent learns to accelerate immediately after it crashes, as it discovers that this is more optimal. It learns that driving at maximum speed is the best in combination with proper steering, and it does not need to slow down to achieve the optimal control policy. A well trained agent will never have to crash, and simply drive at maximum speed, as if it had no influence over the acceleration, just like the initial algorithm assumed.

5-2 Experiments with the cart-pole

In this section, we will do a secondary experiment with the cart-pole environment. The following experiments will show the generality of using Hebbian learning with global rewards in complex control systems.

In the first attempt, we will consider as if the components can be used in the same way as in

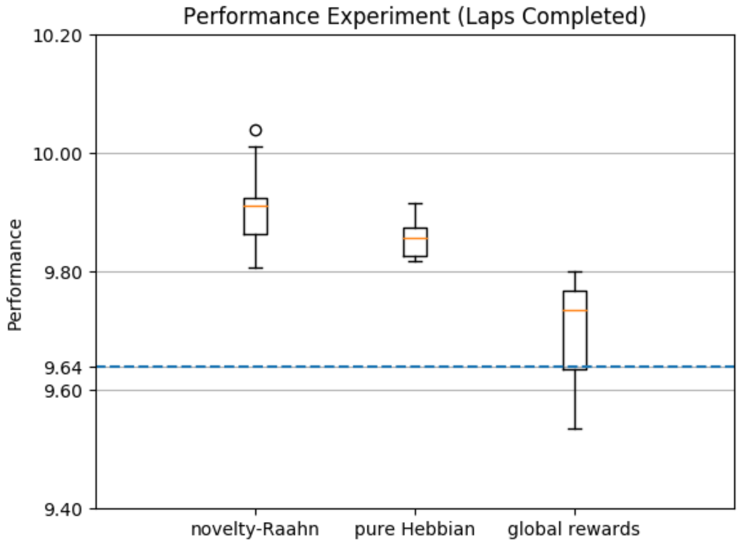

Figure 5-3: Depiction of the amount of laps completed over a certain amount of experiments. The figure shows that the global reward method achieves similar performances as the state-of-the-art algorithms after sufficient training, since the satisfactory threshold of 9.64 laps is achieved on average.

the autonomous car case. We have a cart-pole with four states

$$x_k = \begin{pmatrix} s_k \\ \dot{s}_k \\ \phi_k \\ \dot{\phi}_k \end{pmatrix}, \tag{5-2}$$

where s_k denotes the horizontal positioning, ϕ_k the angle with respect to the upper equilibrium point, and the dot notation represent their respective time-derivative. The graphical representation of the corresponding environment is shown in Figure 5-4.

Note that $x_k \in \mathbb{R}^4$ may have arbitrary real numbers, whereas the assumptions of novelty-Raahn consider an input range of [0, 1]. We can solve this easily by pre-processing the state to a suitable range of values between 0 and 1. After some tests with the environment, the pre-processed state

$$x_{p,k} = \begin{pmatrix} 1 \\ 1/4 \\ 2 \\ 1/6 \end{pmatrix} \odot x_k + \begin{pmatrix} 0.5 \\ 0.5 \\ 0.25 \\ 0.5 \end{pmatrix} \tag{5-3}$$

turns out to produce suitable values close to the range [0,1], such that the autoencoder error could theoretically reach close to zero.

The remaining layers of the agent network remain the same as before, along with the training process. The only obvious difference now is the choice of global reward. If we consider the state x_k, then we can consider the main objective to bring this state to its desired equilibrium

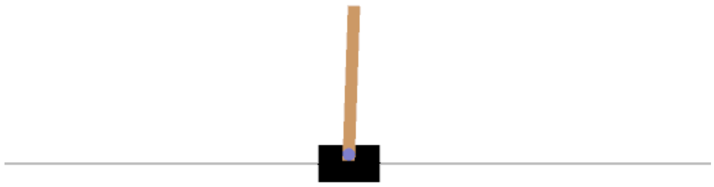

Figure 5-4: Graphical representation of the cart-pole environment, with four known states. Objective is to stabilize the pole in its upper equilibrium (unstable point).

state x_d.

$$x_d = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}. \tag{5-4}$$

The measure of negative discrepancy between the actual state and the desired state is considered the particular reward. Namely, the Euclidean distance of the weighted state:

$$r_k = -\sqrt{w_r^T(x_d - x_k)} = -\sqrt{w_r^T x_k}. \tag{5-5}$$

After some trial and error with tuning, the relative weights w_r consists of the values

$$w_r = \begin{pmatrix} 0.1 \\ 0.1 \\ 1.3 \\ 0.1, \end{pmatrix} \tag{5-6}$$

because most importance is given to stabilizing the angle ϕ_k. Various anticipation steps ahead N have also been tried, but unfortunately the algorithm did learn to stabilize the environment with these settings. After reflecting back upon the precise learning method in combination with the given environment, I have concluded the following: Firstly, one significant difference now is that the state x_k is already representative of the input space. The four state values represent the physical features of all important aspects, so we could argue that an autoencoder is not necessary, just like with the pure Hebbian controller in section 2-2-4.

Another main problem is still the uncontrollability or instability of the environment. If we observe visually at what happens during training, we see that because the system is unstable without a working controller initially, it keeps receiving negative modulation and hence keeps unlearning all it observes. This is now a major obstacle to proper learning, because the methods of raahn and global rewards assume that the environment has enough time for the modulation signal to directly drive it to the next state in which it receives a higher global reward. Whereas the unstable cart-pole environment becomes quickly unstable ones the pole deviates too much from the equilibrium state.

Finally, the network structure as proposed in [1] also does not contain enough complexity for the purposes of cart-pole. With the network structure as proposed in [1], i.e. as described in chapter 2, only a single layer is effectively used for learning control. The other part (autoencoder), if present, learns the high level perception of the same input space that the sensors supply. We have to take note that a single-layered feedback controller is not sufficient in complexity to control the cart-pole as used in this experiment. Contrary to controlling classic cart-pole systems, which are controllable via a single-layered linear mapping, where the main difference is that our controller is forced to be non-linear. The reason being is that our used cart-pole example has a discrete action space of two values (u=1 or u=0), so deriving a controller through linearization is not possible.

With this in mind, we will make some major changes in the agent's architecture. Firstly, we have established above that an autoencoder is not really necessary. The states however, are still in the neurally unsuitable real domain (i.e. it does not corresponds to biologically plausible neuronal activations, like a voltage spike that determines firing or non-firing). For this, we will change the neuronal dynamics of the hidden layers to be either -1 (non-firing) or +1 (firing), where specifically the activation function at each layer is the sign function. Such a structure is similar as in literature in [12], which is used as a supervised learning network. Apart from handling the input neurons correctly the main reasons for such a structure choice are the following: Firstly, this structure allows for a multi-layered network, which is required for more control complexity as we have established before. The version of Hebbian learning in the raahn algorithm normalizes the control output first for proper learning, which is not defined for a multilayered structure. The Hebbian weight updates in [12] are now simply ηm or $-\eta m$, based on the pre- and postsynaptic activations. Here, η denotes the learning rate and m the global modulation signal. The network structure as such is strictly defined to learn in a supervised fashion, from which we can infer a different understanding in the sense of reinforcement learning.

In order to make the agent react to global rewards, we will now introduce what the global reward R_k^N will be. Since an episode of cart-pole terminates ones it deviates too much from the equilibrium, we can assign a particular reward r_k of 1, at every time tick that it is alive. The anticipation step value N is now dynamic, and based on the duration of each episode. We can then construct the global reward R_k^N to be equal to

$$R_k^N = \frac{1}{200}\sum_{i=k}^{N} r_i. \tag{5-7}$$

The 200 in the denominator corresponds to a perfect episode of 200 steps alive, giving a value of $R_k^N = 1$. If we consider the performance of cart-pole, then we anticipate that a lot of learning is necessary initially when the performance is bad, and little to no learning is required once the performance is good. In other words, this means that we already now the optimal global reward that is achievable ($R_k^N = 1$), and so rather than defining the modulation as

$$m_k = g(R_{k+1}^N - R_k^N), \tag{5-8}$$

we can directly write the modulation as

$$m_k = 1 - R_k^N. \tag{5-9}$$

Note that no negative modulation can occur (R_k^N is between 0 an 1), and so the network will simply learn based on how well it performs. Note also, that we now use true episodic optimization where we wait each episode before learning, and update the weights based on the rolled out policy in that episode, whereas in the previous experiments with autonomous car we used a strictly real-time adaptive learner.

For this experiment, we will also change the amount of layers to three, with first layer a mapping from the four states x_k to 12 hidden features $h_{1,k}$. The second layer a mapping from the previous features to 24 hidden features $h_{2,k}$. And finally the last layer sigmoidal logistic mapping (like in raahn) from the previous 24 features to a single scalar value u_k in the range [0, 1], where 1 denotes a right-force to the cart, and 0 denotes a left-force to the cart. The actual control value is rounded, and so the cart is not allowed to stand still, i.e. it must get either a left- or right-force.

With these new agent settings, training works quite well. The cart-pole is considered *solved* if it remains alive[1] for at least 195 time steps, for 100 consecutive episodes. The algorithm was tested 10 times, and cart-pole was solved in an average of 343.3 episodes, using the global reward method as described above.

[1]The cart-pole environment stays alive as long as the pole does not deviate more than 15 degrees from its upper equilibrium, and the position not more than 2.4 meters away from the center.

Chapter 6

Discussion and Future Work

We have shown in this thesis that Hebbian learning can effectively be used to control autonomous systems. The autonomous car has learned to control both its steering angle and its velocity simultaneously, through the maximization of a single global reward function. The sensitivity of the performance is in this way shifted from the specific modulation scheme, to a global reward function. Even though care still needs to be taken to correctly set up a global reward function, I would argue that it must be easier, since it is designed in a general context, rather than a specific one for a certain control variable. For future work, we can look at potentially even better ways to come up with this general global reward. We could for example look to use a different idea of the long term effect on the agent with some form of eligibility trace, like in different methods in literature. More importantly, I would say that establishing reward-based Hebbian learning theory using convergence proofs would have a high priority, since this would allow for an easier approach given a set of assumptions about the problem. Furthermore, we could potentially think about different methods of storing sample specific rewards, like a value function approximation as in other reinforcement learning approaches, and use our general modulation scheme along with these reward values to learn more robustly. Overall, the idea of using global rewards opens up a new door to novel algorithms with Hebbian learning, as the definition space of objective functions is very broad.

Chapter 7

Conclusion

I proposed in this thesis that global reward functions can be used to train autonomous systems, which solves the main drawbacks of the existing novelty-Raahn algorithm. In this way, it makes sure that the agent can learn all control agent simultaneously. This is supported by the results in chapter 5. These experiments have shown a couple of interesting things. Firstly, we have shown that autonomous control with Hebbian learning now actually becomes an optimization problem, in which the reward converges. Secondly, the new algorithm enables a similar performance as the existing algorithms, while having to control a more complex system. Finally, we have shown that the method is indeed more general, since it is able to learn to control a completely different system, with an unstable equilibrium. The type of design effort required is now different in that it focuses on a more general concept of optimization, rather than how the desired behavior is related to the exact dynamics. From this we can conclude that the algorithm has become more useful, which is also promising for future research.

Bibliography

[1] K. Stanley, J. Pugh, and J. Bowren, "Fully autonomous real-time autoencoder-augmented hebbian learning through the collection of novel experiences," *The 2018 Conference on Artificial Life: A Hybrid of the European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE)*, pp. 382–389, 2016.

[2] J. de Vries, "Collision detection," in *https://learnopengl.com/In-Practice/2D-Game/Collisions/Collision-detection*, 2018.

[3] J. Hohwy, *The Predictive Mind.* OUP Oxford, 2013.

[4] K. J. Friston, "The free-energy principle: a rough guide to the brain?," *Trends in cognitive sciences*, vol. 13 7, pp. 293–301, 2009.

[5] D. O. Hebb, *The organization of behavior : a neuropsychological theory.* New York [etc.]: Wiley,, 1949.

[6] S. Lowel and W. Singer, "Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity," *Science*, vol. 255, no. 5041, pp. 209–212, 1992.

[7] S. Haykin, *Neural networks : a comprehensive foundation.* Upper Saddle River, N.J.: Prentice Hall,, 1999.

[8] B. Baruque and E. Corchado, *Fusion methods for unsupervised learning ensembles.* Studies in computational intelligence ; v. 322; Studies in computational intelligence ; v. 322., Berlin ;: Springer,, 2011.

[9] J. Burms, K. Caluwaerts, and J. Dambre, "Reward-modulated hebbian plasticity as leverage for partially embodied control in compliant robotics," *Frontiers in Neurorobotics*, vol. 9, p. 9, 2015.

[10] Q. Sheng, Z. Xianyi, W. Changhong, X. Z. Gao, and L. Zilong, "Design and implementation of an adaptive pid controller using single neuron learning algorithm," in *Proceedings*

of the 4th World Congress on Intelligent Control and Automation (Cat. No.02EX527), vol. 3, pp. 2279–2283 vol.3, 2002.

[11] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *CoRR*, vol. abs/1606.01540, 2016.

[12] R. W. Strom, "Hebbian learning in multilayer neural networks," Master's thesis, California State University, Los Angeles, 2007.

Glossary

List of Acronyms

Index

Source Code

Appendices are found in the back.

-1 Python code for the autonomous car agent

-1-1 The environment

The following code is part of the autonomous car environment.

raahn.py

```


# -*- coding: utf-8 -*-
"""
Created on Tue Jun  5 15:12:41 2018


@author: ajdin
"""
import pygame

from CarMazeEnv import CarMazeEnv

display_width = 1600
display_height = 1440

res = (display_width, display_height)

simul = None


def main():
  global simul
  simul = CarMazeEnv()
```

```
  render = False
  RUNNING, PAUSE = 0, 1
  state = RUNNING

  while not simul.game_ext:
    for event in pygame.event.get():
      if event.type == pygame.QUIT or simul.game_ext:
        simul.close()
        return

      if event.type == pygame.KEYDOWN:
        if event.key == pygame.K_p and state == RUNNING: state = PAUSE
        elif event.key == pygame.K_p and state == PAUSE: state = RUNNING

    if state == RUNNING:
      simul.crashcounting_test()
    try:
      if render:
        simul.render(res)
    except Exception as E:
      simul.close()
      raise E



main()
pygame.quit()
#quit()

#%%
#import envmanager
#import matplotlib.pyplot as plt
###
###
#import numpy as np
#
# plt.scatter(envmanager.performance_vector, envmanager.modulation_vector)
#performance_vector = envmanager.performance_vector
#modulation_vector = envmanager.modulation_vector
#p = performance_vector.argsort()
#plt.plot(np.tanh(12*performance_vector[p]), modulation_vector[p])
#sensor_vector = envmanager.sensor_vector
#
#
#
#delta_performance_vector = np.zeros(len(performance_vector))
#delta_modulation_vector = np.zeros(len(modulation_vector))
#delta_sensor_vector = np.zeros(len(sensor_vector))
#batchsize = 2
#for x in range(len(performance_vector)-batchsize+1):
#    delta_performance_vector[x+1] = performance_vector[x+batchsize-1] - performance_vector[x]
```

```
#    delta_modulation_vector[x+1] = np.mean(modulation_vector[x:x+batchsize])
#    delta_sensor_vector[x+1] = -sensor_vector[x+batchsize-1] + sensor_vector[x]
#
##sort the indices
#p = delta_sensor_vector.argsort()
#
#
#plt.plot(delta_sensor_vector[p], modulation_vector[p])
```

Car.py

```
# -*- coding: utf-8 -*-
"""
Created on Wed Jun 13 21:11:27 2018

@author: ajdin
"""
#from Line import Line

import numpy as np

import xml.etree.ElementTree as ET
import pygame

from lines import LineSegment

redcolor = (0xff,0,0)

class CarConfig:
  """
  Configures initial parameters or constants, used by the Car class.
  """
  root = ET.parse('Maps/XMap.xml').getroot()
  ROOT = root.find('Robot')
  def __init__(self, root=ROOT):
    self.x = float(root.find('X').text)
    self.y = float(root.find('Y').text)
    self.angle = float(root.find('Angle').text)


class Car(object):
  """
  Agent that moves through a 2D environment, which can be visualized using a pygame surface.

  Parameters
  ----------
  center: tuple, optional.
    Sets the initial position in a 2D plane using Cartesion coordinates. default is (0,0).

```

```
    angle: float, optional.
      Sets the initial angle in degrees of the agent relative to a 2D frame
         . default is 0.

    Attributes
    ----------
    center:

    velocity:

    speed:

    acceleration:

    image:

    angle:

    """

    CONTROL_THRESHOLD = 0.5
    MIN_SPEED_X = 10.0
    MIN_SPEED_Y = 8.0
    MAX_SPEED_X = 15.0
    MAX_SPEED_Y = 12.0
    MAX_ROTATE = 1.0
    MIN_ROTATE = 0.0
    ROTATE_SPEED = 2.0
    ROTATE_RANGE = 2.0*ROTATE_SPEED
    ACCELERATION = 0.08
    ACCELERATION_RANGE = 2.0*ACCELERATION
    RADIUS = 3.0

    def __init__(self, center=(0,0), angle=0):
      self.center = center
      self.lastpos = center
      self.velocity = (0, 0)
      self.speed = (self.MAX_SPEED_X, self.MAX_SPEED_Y)
      self.speed_x = self.speed[0]
      self.acceleration = 0
      self.image = pygame.image.load('Textures/CarResized.png')
      self.angle = angle % 360
      self.last_can_move = True
      self.can_move = True
      self.reset_speed = False
      self.num_crashes = 0
      self.upper_bounds = (self.MAX_SPEED_X, self.MAX_SPEED_Y)
      self.lower_bounds = (self.MIN_SPEED_X, self.MIN_SPEED_Y)

    def update(self, walls):
      self.lastpos = self.center
      radians = np.radians(self.angle)
      yx_ratio = self.MAX_SPEED_Y / self.MAX_SPEED_X
```

```
    resulting_speed = (self.speed_x, yx_ratio*self.speed_x)
    self.speed = np.clip(resulting_speed, self.lower_bounds, self.upper_bounds)
    self.speed_x = np.clip(self.speed_x, self.MIN_SPEED_X, self.MAX_SPEED_X)
    self.velocity = (np.cos(radians)*self.speed[0], np.sin(radians)*self.speed[1])
    original = self.center
    projected = tuple(np.add(original, self.velocity))
    collision_line = LineSegment(original, projected)
    walls_in_bounds = collision_line.entities_in_bounds(walls)
    self.can_move = True
    for wall in walls_in_bounds:
      intersections = collision_line.intersects(wall)
      if  len(intersections) > 0:
        if self.last_can_move:
          self.num_crashes += 1
        self.can_move = False
        if self.reset_speed:
          self.speed = (self.MIN_SPEED_X, self.MIN_SPEED_Y)
        break
    if self.can_move:
      self.center = tuple(np.add(self.center, self.velocity))
    self.last_can_move = self.can_move

  def draw(self, display, options):
    x_scale = options['x_scale']
    x_translate = options['x_translate']
    x = self.center[0]*x_scale + x_translate
    y = self.center[1]*x_scale + x_translate
    angle = self.angle
    angle %= 360
    rotated_img = pygame.transform.rotate(self.image, -angle)
    original_rect = self.image.get_rect(center=(x,y))
    rotated_rect = rotated_img.get_rect(center=original_rect.center)
    display.blit(rotated_img, rotated_rect)
    pygame.draw.circle(display, redcolor, tuple(int(i) for i in (x, y)), int(self.RADIUS), 2)
```

configs.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 16 08:22:29 2018

@author: ajdin
"""

import json

file = 'Networks/HebbianNet.json'

class Net3Config:
```

```python

  def __init__(self, filename=file):
    self.layers = []
    with open(filename) as rfile:
      container = json.load(rfile)
    for layer in container['NeuralNetwork3']['NetworkLayers']:
      try:
        neuron_count = layer['neuron_count']
        learning_rate = layer['learning_rate']
        training_method = layer['training_method']
        modulation_scheme = layer['modulation_scheme']
        layerconfig = LayerConfig(neuron_count, learning_rate,
            training_method, modulation_scheme)
        self.layers.append(layerconfig)
      except KeyError:
        break
    self.input_count = container['NeuralNetwork3']['input_count']
    self.output_noise_mag = container['NeuralNetwork3']['output_noise_mag
        ']
    self.weight_noise_mag = container['NeuralNetwork3']['weight_noise_mag
        ']
    self.weight_cap = container['NeuralNetwork3']['weight_cap']


class LayerConfig:

  def __init__(self, neuron_count, learning_rate, training_method,
      modulation_scheme):
    self.neuron_count = neuron_count
    self.learning_rate = learning_rate
    self.training_method = training_method
    self.modulation_scheme = modulation_scheme
```

lines.py

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Aug 25 16:27:41 2018

@author: ajdin
"""
import numpy as np

def get_dist(p1 , p2):
  return ((p2[1]-p1[1])**2 + (p2[0]-p1[0])**2)**(1/2)

class LineSegment:

  def __init__(self, startPoint, endPoint):
    self.startPoint = startPoint
    self.endPoint = endPoint
    self.setUp()

```

```
  def setUp(self):
    deltaX = self.endPoint[0] - self.startPoint[0]

    #Set up the bounds
    self.upperBoundY = max(self.endPoint[1], self.startPoint[1])
    self.lowerBoundY = min(self.endPoint[1], self.startPoint[1])

    self.upperBoundX = max(self.endPoint[0], self.startPoint[0])
    self.lowerBoundX = min(self.endPoint[0], self.startPoint[0])

    #If the change in x is 0, the slope is undefined
    if abs(deltaX) <= 1e-5:
      self.vertical = True
      self.slope = None
      #If the line segment is just a point, it's x coordinate is stored in lowerBounds
      self.lowerBoundX = self.startPoint[0]
    #End if
    else:
      self.vertical = False
      self.slope = (self.endPoint[1] - self.startPoint[1]) / deltaX
      self.yIntercept = self.startPoint[1] - (self.slope * self.startPoint[0])

  def getY(self, x):
    if x >= self.lowerBoundX and x <= self.upperBoundX and not self.vertical:
      return self.slope*x + self.yIntercept
    else:
      return np.inf

  def intersects(self, line):
    intersection = []
    bothVertical = self.vertical and line.vertical
    parallel = self.slope == line.slope and not self.vertical and not line.vertical

    if bothVertical or parallel:
      if (bothVertical and self.lowerBoundX == line.lowerBoundX) or (parallel and self.yIntercept == line.yIntercept):
        lowerInBounds = self.valueInBounds(self.lowerBoundY, line.lowerBoundY, line.upperBoundY)
        upperInBounds = self.valueInBounds(self.upperBoundY, line.lowerBoundY, line.upperBoundY)

        if lowerInBounds and upperInBounds:
          intersection.append((self.lowerBoundX, self.lowerBoundY))
          intersection.append((self.lowerBoundX, self.upperBoundY))
        elif lowerInBounds:
          intersection.append((self.lowerBoundX, self.lowerBoundY))
          intersection.append((self.lowerBoundX, line.upperBoundY))
        elif upperInBounds:
          intersection.append((self.lowerBoundX, line.lowerBoundY))
```

```
          intersection.append((self.lowerBoundX, self.upperBoundY))
        elif self.valueInBounds(line.lowerBoundY, self.lowerBoundY, self.upperBoundY) and self.valueInBounds(line.upperBoundY, self.lowerBoundY, self.upperBoundY):
          intersection.append((self.lowerBoundX, line.lowerBoundY))
          intersection.append((self.lowerBoundX, line.upperBoundY))
      return intersection
    elif self.vertical:
      y = line.getY(self.lowerBoundX)
      #Maker sure the returned y is valid
      #GetY not returning infinity makes sure that the point is in bounds of this line
      if y == np.inf or not line.valueInBounds(y, self.lowerBoundY, self.upperBoundY):
        return intersection

      intersection.append((self.lowerBoundX, y))
      return intersection
    elif line.vertical:
      y = self.getY(self.lowerBoundX)
      #Make sure the returned y is valid
      #GetY not returning infinity makes sure that the point is in bounds of line
      if y == np.inf or not self.valueInBounds(y, line.lowerBoundY, line.upperBoundY):
        return intersection

      intersection.append((line.lowerBoundX, y))
      return intersection
    else:
      intersectionX = (line.yIntercept - self.yIntercept) / (self.slope - line.slope)
      if self.valueInBounds(intersectionX, self.lowerBoundX, self.upperBoundX) and self.valueInBounds(intersectionX, line.lowerBoundX, line.upperBoundX):
        intersection.append(((intersectionX), self.getY(intersectionX)))

      return intersection


  def valueInBounds(self, value, lowerBound, upperBound):
    if value >= lowerBound and value <= upperBound:
      return True
    else:
      return False

  def angle_between(self, line2):
    vector1 = (self.endPoint[0] - self.startPoint[0], self.endPoint[1] - self.startPoint[1])
    vector2 = (line2.endPoint[0] - line2.startPoint[0], line2.endPoint[1] - line2.startPoint[1])
    inner_product = vector1[0]*vector2[0] + vector1[1]*vector2[1]
    len1 = np.linalg.norm(vector1)
```

```
        len2 = np.linalg.norm(vector2)
        return np.degrees(np.arccos(inner_product/(len1*len2)))

    @staticmethod
    def overlapsInDimension(pointsA, pointsB):
        Amin = min(pointsA)
        Amax = max(pointsA)
        Bmin = min(pointsB)
        Bmax = max(pointsB)
        if Amin > Bmax:
            return False
        if Amax < Bmin:
            return False
        return True

#   def entities_in_bounds(self, entities):
#       entities_in_bounds = []
#       for entity in entities:
#           lowerInBoundsY = self.valueInBounds(self.lowerBoundY, entity.
    lowerBoundY, entity.upperBoundY)
#           upperInBoundsY = self.valueInBounds(self.upperBoundY, entity.
    lowerBoundY, entity.upperBoundY)
#           lowerInBoundsX = self.valueInBounds(self.lowerBoundX, entity.
    lowerBoundX, entity.upperBoundX)
#           upperInBoundsX = self.valueInBounds(self.upperBoundX, entity.
    lowerBoundX, entity.upperBoundX)
#           entityInBounds = lowerInBoundsY or upperInBoundsY or
    lowerInBoundsX or upperInBoundsX
#           if entityInBounds:
#               entities_in_bounds.append(entity)
#       return entities_in_bounds

    def entities_in_bounds(self, entities):
        entities_in_bounds = []
        pointsAx = (self.lowerBoundX, self.upperBoundX)
        pointsAy = (self.lowerBoundY, self.upperBoundY)
        for entity in entities:
            pointsBx = (entity.lowerBoundX, entity.upperBoundX)
            pointsBy = (entity.lowerBoundY, entity.upperBoundY)
            overlapsInX = LineSegment.overlapsInDimension(pointsAx, pointsBx)
            overlapsInY = LineSegment.overlapsInDimension(pointsAy, pointsBy)
            entityInBounds = overlapsInX and overlapsInY
            if entityInBounds:
                entities_in_bounds.append(entity)
        return entities_in_bounds
```

rangefinder.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 26 20:08:34 2018

@author: ajdin
```

```python
"""
from lines import LineSegment
from lines import get_dist

import numpy as np
import pygame

class RangeFinder(LineSegment):

  def __init__(self, car, default_length=350, relativeAngle=0):
    self.car = car
    self.default_length = default_length
    self.length = self.default_length
    self.relativeAngle = relativeAngle
    super().__init__(self.getStartPoint(), self.getEndPoint())
    self.activation = 0.0

  def getStartPoint(self):
    return self.car.center

  def getEndPoint(self):
    rads = np.radians(self.car.angle + self.relativeAngle)
    change = (np.cos(rads), np.sin(rads))
    return tuple(np.add(self.car.center, tuple(x*(self.default_length) for x in change)))

  def update_position(self):
    self.startPoint = self.getStartPoint()
    self.endPoint = self.getEndPoint()
    self.setUp()

  def update(self, walls):
    self.update_position()
    walls_in_bounds = self.entities_in_bounds(walls)
    nearestWallDistance = self.default_length
    for wall in walls_in_bounds:
      intersections = self.intersects(wall)
      if len(intersections) > 0:
        distance = get_dist(self.car.center, intersections[0])
        nearestWallDistance = min(distance, nearestWallDistance)
    self.length = nearestWallDistance
    self.activation = (self.default_length - self.length) / self.default_length

  def draw(self, display, options):
    color = tuple([self.activation*x for x in (255, 80, 0)])
    x_scale = options['x_scale']
    x_translate = options['x_translate']
    startPoint = tuple(x_scale*x+x_translate for x in self.startPoint)
    endPoint= tuple(x_scale*x+x_translate for x in self.endPoint)
    pygame.draw.aaline(display, color, startPoint, endPoint)


```

```python
class RangeFinderGroup(list):

  def __init__(self, car, size=11):
    self.car = car
    self.count = size
    self.activations = []
    self.configure()

  def configure(self, default_length=350, angleOffset=-90, angle_between
      =18):
    self.default_length = default_length
    self.startAngle = angleOffset
    self.angleSpacing = angle_between

    for i in range(self.count):
      relativeAngle = self.startAngle + (self.angleSpacing*i)
      self.append(RangeFinder(self.car, self.default_length,
          relativeAngle))
      self.activations.append(self[i].activation)

  def update(self, walls):
    for i in range(self.count):
      self[i].update(walls)
      self.activations[i] = self[i].activation

  def draw(self, display, options):
    for i in range(self.count):
      self[i].draw(display, options)
```

Wall.py

```python
# -*- coding: utf-8 -*-
"""
Created on Thu Sep 13 13:26:34 2018

@author: ajdin
"""
from lines import LineSegment
import pygame
import xml.etree.ElementTree as ET

root = ET.parse('Maps/XMap.xml').getroot()
ROOT = root.find('Entity')

class EntityConfig:
  def __init__(self, root=ROOT):
    self.x = float(root.find('X').text)
    self.y = float(root.find('Y').text)
    self.relX = float(root.find('RelX').text)
    self.relY = float(root.find('RelY').text)
    self.angle = float(root.find('Angle').text)
    self.type_ = None
    if root.get('Type'):
```

```
        self.type_ = root.get('Type')

black = (0, 0, 0)

class Wall(LineSegment):
    def __init__(self, startPoint, endPoint):
        super().__init__(startPoint, endPoint)

    def draw(self, display, options):
        x_scale = options["x_scale"]
        x_translate = options["x_translate"]
        startPoint = tuple(x_scale*x+x_translate for x in self.startPoint)
        endPoint = tuple(x_scale*x+x_translate for x in self.endPoint)
        pygame.draw.aaline(display, black, startPoint, endPoint)
```

-1-2 Xmap.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<Map xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Robot>
    <X>1561.9331742243444</X>
    <Y>505.79080279525812</Y>
    <Angle>0</Angle>
  </Robot>
  <Entity Type="Wall">
    <X>1301.3842482100238</X>
    <Y>675.13128038897889</Y>
    <RelX>769.83293556080639</RelX>
    <RelY>1.4438228390645236E-11</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2071.21718377083</X>
    <Y>675.13128038899333</Y>
    <RelX>210.78758949880694</RelX>
    <RelY>-52.512155591572082</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2282.0047732696371</X>
    <Y>622.61912479742125</Y>
    <RelX>155.79952267303088</RelX>
    <RelY>-171.53970826580246</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2437.804295942668</X>
    <Y>451.07941653161879</Y>
    <RelX>128.30548926014308</RelX>
    <RelY>-297.56888168557509</RelY>
    <Angle>0</Angle>
  </Entity>
```

```
<Entity Type="Wall">
  <X>2566.1097852028111</X>
  <Y>153.5105348460437</Y>
  <RelX>142.05250596658698</RelX>
  <RelY>-150.53484602917376</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2708.162291169398</X>
  <Y>2.9756888168699334</Y>
  <RelX>164.96420047732727</RelX>
  <RelY>-38.508914100486209</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2873.1264916467253</X>
  <Y>-35.533225283616275</Y>
  <RelX>155.79952267303088</RelX>
  <RelY>45.510534846029145</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3028.9260143197562</X>
  <Y>9.97730956241287</Y>
  <RelX>105.39379474940324</RelX>
  <RelY>94.521880064829816</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3134.3198090691594</X>
  <Y>104.49918962724269</Y>
  <RelX>-13.747016706443901</RelX>
  <RelY>143.53322528363049</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3120.5727923627155</X>
  <Y>248.03241491087317</Y>
  <RelX>-128.30548926014308</RelX>
  <RelY>196.04538087520177</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2992.2673031025724</X>
  <Y>444.07779578607494</Y>
  <RelX>-160.38186157517885</RelX>
  <RelY>227.55267423014607</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2831.8854415273936</X>
  <Y>671.630470016221</Y>
  <RelX>-201.622911694511</RelX>
```

```
    <RelY>161.03727714748766</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2630.2625298328826</X>
    <Y>832.66774716370867</Y>
    <RelX>-114.55847255369918</RelX>
    <RelY>154.03565640194506</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2515.7040572791834</X>
    <Y>986.70340356565373</Y>
    <RelX>-68.7350835322195</RelX>
    <RelY>147.03403565640167</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2446.9689737469639</X>
    <Y>1133.7374392220554</Y>
    <RelX>-9.164677804295934</RelX>
    <RelY>129.52998379254495</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2437.804295942668</X>
    <Y>1263.2674230146004</Y>
    <RelX>36.658711217183736</RelX>
    <RelY>154.03565640194483</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2474.4630071598517</X>
    <Y>1417.3030794165452</Y>
    <RelX>206.20525059665852</RelX>
    <RelY>105.02431118314416</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2680.6682577565102</X>
    <Y>1522.3273905996894</Y>
    <RelX>531.55131264916554</RelX>
    <RelY>367.58508914100548</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3212.2195704056758</X>
    <Y>1889.9124797406948</Y>
    <RelX>206.20525059665852</RelX>
    <RelY>234.55429497568866</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
```

```
      <X>3418.4248210023343</X>
      <Y>2124.4667747163835</Y>
      <RelX>-9.164677804295934</RelX>
      <RelY>133.03079416531591</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>3409.2601431980383</X>
      <Y>2257.4975688816994</Y>
      <RelX>-100.81145584725527</RelX>
      <RelY>133.03079416531546</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>3308.4486873507831</X>
      <Y>2390.5283630470149</Y>
      <RelX>-160.38186157517976</RelX>
      <RelY>38.508914100485526</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>3148.0668257756033</X>
      <Y>2429.0372771475004</Y>
      <RelX>-288.68735083532192</RelX>
      <RelY>-35.008103727714115</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>2859.3794749402814</X>
      <Y>2394.0291734197863</Y>
      <RelX>-192.45823389021461</RelX>
      <RelY>-101.5235008103723</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>2666.9212410500668</X>
      <Y>2292.505672609414</Y>
      <RelX>-375.75178997613375</RelX>
      <RelY>-287.06645056726074</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>2291.169451073933</X>
      <Y>2005.4392220421532</Y>
      <RelX>-261.19331742243435</RelX>
      <RelY>-147.034035656402</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>2029.9761336514987</X>
      <Y>1858.4051863857512</Y>
      <RelX>-169.546539379475</RelX>
      <RelY>-24.505672609400335</RelY>
```

```
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>1860.4295942720237</X>
      <Y>1833.8995137763509</Y>
      <RelX>-490.31026252983293</RelX>
      <RelY>-3.5008103727714115</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>1370.1193317421908</X>
      <Y>1830.3987034035795</Y>
      <RelX>-174.1288782816232</RelX>
      <RelY>70.016207455429139</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>1195.9904534605676</X>
      <Y>1900.4149108590086</Y>
      <RelX>-229.11694510739824</RelX>
      <RelY>178.54132901134517</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>966.87350835316931</X>
      <Y>2078.9562398703538</Y>
      <RelX>-155.79952267303111</RelX>
      <RelY>203.04700162074505</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>811.07398568013821</X>
      <Y>2282.0032414910988</Y>
      <RelX>-219.95226730310253</RelX>
      <RelY>147.034035656402</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>591.12171837703568</X>
      <Y>2429.0372771475008</Y>
      <RelX>-229.11694510739858</RelX>
      <RelY>45.510534846029259</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>362.0047732696371</X>
      <Y>2474.54781199353</Y>
      <RelX>-256.61097852028638</RelX>
      <RelY>-28.006482982171747</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>105.39379474935072</X>
```

```
    <Y>2446.5413290113584</Y>
    <RelX>-151.21718377088303</RelX>
    <RelY>-105.02431118314416</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-45.823389021532307</X>
    <Y>2341.5170178282142</Y>
    <RelX>2</RelX>
    <RelY>-154.03565640194483</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-43.823389021532307</X>
    <Y>2187.4813614262694</Y>
    <RelX>192.45823389021473</RelX>
    <RelY>-238.05510534846053</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>146.63484486868242</X>
    <Y>1949.4262560778088</Y>
    <RelX>279.52267303102622</RelX>
    <RelY>-238.05510534845939</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>426.15751789970864</X>
    <Y>1711.3711507293494</Y>
    <RelX>215.36992840095468</RelX>
    <RelY>-185.54294975688777</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>641.52744630066331</X>
    <Y>1525.8282009724617</Y>
    <RelX>119.14081145584737</RelX>
    <RelY>-168.03889789303093</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>760.66825775651068</X>
    <Y>1357.7893030794307</Y>
    <RelX>-4.582338902147967</RelX>
    <RelY>-185.54294975688754</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>1301.3842482100238</X>
    <Y>675.13128038897889</Y>
    <RelX>-183.29355608597143</RelX>
    <RelY>-115.52674230144339</RelY>
    <Angle>0</Angle>
```

```
  </Entity>
  <Entity Type="Wall">
    <X>1118.0906921240523</X>
    <Y>559.6045380875355</Y>
    <RelX>-187.87589498806688</RelX>
    <RelY>-171.53970826580229</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>930.21479713598546</X>
    <Y>388.06482982173321</Y>
    <RelX>-183.29355608591868</RelX>
    <RelY>-94.521880064829816</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>746.92124105006678</X>
    <Y>293.5429497569034</Y>
    <RelX>-270.35799522673051</RelX>
    <RelY>-94.521880064829588</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>476.56324582333627</X>
    <Y>199.02106969207381</Y>
    <RelX>-155.79952267303088</RelX>
    <RelY>14.00324149108576</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>320.7637231503054</X>
    <Y>213.02431118315957</Y>
    <RelX>-128.30548926014319</RelX>
    <RelY>84.019448946515354</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>192.45823389016221</X>
    <Y>297.04376012967492</Y>
    <RelX>-64.152744630071652</RelX>
    <RelY>126.02917341977309</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>128.30548926009055</X>
    <Y>423.072933549448</Y>
    <RelX>36.658711217183736</RelX>
    <RelY>133.03079416531608</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>164.96420047727429</X>
    <Y>556.10372771476409</Y>
```

```
      <RelX>91.646778042959454</RelX>
      <RelY>101.52350081037207</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>256.61097852023374</X>
      <Y>657.62722852513616</Y>
      <RelX>192.45823389021473</RelX>
      <RelY>126.02917341977172</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>449.06921241044847</X>
      <Y>783.65640194490788</Y>
      <RelX>197.04057279236281</RelX>
      <RelY>126.02917341977309</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>646.10978520281128</X>
      <Y>909.685575364681</Y>
      <RelX>91.646778042959568</RelX>
      <RelY>94.5218800648297</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>737.75656324577085</X>
      <Y>1004.2074554295107</Y>
      <RelX>18.329355608591868</RelX>
      <RelY>168.03889789303253</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>1397.8854415274473</X>
      <Y>462.13495863941552</Y>
      <RelX>535.21718377084085</RelX>
      <RelY>-7.5740259740153988</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>1933.1026252982881</X>
      <Y>454.56093266540012</Y>
      <RelX>109.97613365155121</RelX>
      <RelY>-53.2987012987013</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>2043.0787589498393</X>
      <Y>401.26223136669881</Y>
      <RelX>91.646778042959568</RelX>
      <RelY>-129.03896103896102</RelY>
      <Angle>0</Angle>
    </Entity>
```

```
<Entity Type="Wall">
  <X>2134.7255369927989</X>
  <Y>272.22327032773779</Y>
  <RelX>91.646778042959334</RelX>
  <RelY>-162.7012987012987</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>1397.8854415274473</X>
  <Y>462.13495863941552</Y>
  <RelX>-150.30071599049575</RelX>
  <RelY>-114.171428571418</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>1247.5847255369515</X>
  <Y>347.96353006799751</Y>
  <RelX>-128.3054892601433</RelX>
  <RelY>-143.06493506493496</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>1119.2792362768082</X>
  <Y>204.89859500306255</Y>
  <RelX>-190.62529832935547</RelX>
  <RelY>-145.87012987012997</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>928.65393794745273</X>
  <Y>59.028465132932581</Y>
  <RelX>-234.615751789976</RelX>
  <RelY>-84.155844155844363</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>694.03818615747673</X>
  <Y>-25.127379022911782</Y>
  <RelX>-340.92601431980904</RelX>
  <RelY>-64.519480519480737</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>353.11217183766769</X>
  <Y>-89.646859542392519</Y>
  <RelX>-157.63245823389019</RelX>
  <RelY>16.831168831168725</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>195.4797136037775</X>
  <Y>-72.8156907112238</Y>
  <RelX>-260.27684964200449</RelX>
```

```
    <RelY>53.298701298701076</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-64.797136038227</X>
    <Y>-19.516989412522719</Y>
    <RelX>-164.96420047732698</RelX>
    <RelY>129.03896103896096</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-229.76133651555398</X>
    <Y>109.52197162643824</Y>
    <RelX>-80.649164677804322</RelX>
    <RelY>165.50649350649354</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-310.4105011933583</X>
    <Y>275.02846513293179</Y>
    <RelX>7.3317422434367927</RelX>
    <RelY>232.83116883116918</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-303.07875894992151</X>
    <Y>507.85963396410096</Y>
    <RelX>120.97374701670645</RelX>
    <RelY>207.5844155844415</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>-182.10501193321505</X>
    <Y>715.444049548516</Y>
    <RelX>278.60620525059659</RelX>
    <RelY>193.55844155844147</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>96.501193317381535</X>
    <Y>909.00249110695745</Y>
    <RelX>208.95465393794734</RelX>
    <RelY>165.5064935064936</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>305.45584725532888</X>
    <Y>1074.508984613451</Y>
    <RelX>109.97613365155144</RelX>
    <RelY>145.87012987012963</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
```

```
      <X>415.43198090688031</X>
      <Y>1220.3791144835807</Y>
      <RelX>-10.997613365154905</RelX>
      <RelY>120.623376623377</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>404.43436754172541</X>
      <Y>1341.0024911069577</Y>
      <RelX>-153.96658711217174</RelX>
      <RelY>151.48051948051943</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>250.46778042955367</X>
      <Y>1492.4830105874771</Y>
      <RelX>-245.61336515513125</RelX>
      <RelY>193.55844155844147</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>4.854415274422422</X>
      <Y>1686.0414521459186</Y>
      <RelX>-245.61336515513119</RelX>
      <RelY>235.63636363636351</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>-240.75894988070877</X>
      <Y>1921.6778157822821</Y>
      <RelX>-142.96897374701678</RelX>
      <RelY>238.44155844155807</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>-383.72792362772554</X>
      <Y>2160.11937422384</Y>
      <RelX>1</RelX>
      <RelY>266.49350649350663</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>-382.72792362772554</X>
      <Y>2426.6128807173468</Y>
      <RelX>113.64200477326949</RelX>
      <RelY>143.06493506493507</RelY>
      <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
      <X>-270.08591885445605</X>
      <Y>2569.6778157822819</Y>
      <RelX>263.94272076372317</RelX>
      <RelY>103.79220779220759</RelY>
```

```
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>-6.1431980907328807</X>
        <Y>2673.4700235744895</Y>
        <RelX>421.57517899761331</RelX>
        <RelY>14.025974025973483</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>415.43198090688043</X>
        <Y>2687.4959976004629</Y>
        <RelX>344.59188544152744</RelX>
        <RelY>-109.40259740259808</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>760.02386634840786</X>
        <Y>2578.0934001978649</Y>
        <RelX>260.27684964200466</RelX>
        <RelY>-154.28571428571377</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>1020.3007159904125</X>
        <Y>2423.8076859121511</Y>
        <RelX>304.26730310262519</RelX>
        <RelY>-291.74025974025972</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>1324.5680190930377</X>
        <Y>2132.0674261718914</Y>
        <RelX>201.62291169450987</RelX>
        <RelY>-100.9870129870128</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>1526.1909307875476</X>
        <Y>2031.0804131848786</Y>
        <RelX>263.94272076372363</RelX>
        <RelY>5.610389610389575</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>1790.1336515512712</X>
        <Y>2036.6908027952682</Y>
        <RelX>546.2147971360373</RelX>
        <RelY>356.25974025973892</RelY>
        <Angle>0</Angle>
    </Entity>
    <Entity Type="Wall">
        <X>2336.3484486873085</X>
```

```
    <Y>2392.9505430550071</Y>
    <RelX>318.93078758949878</RelX>
    <RelY>148.6753246753251</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2655.2792362768073</X>
    <Y>2541.6258677303322</Y>
    <RelX>421.57517899761342</RelX>
    <RelY>84.155844155843624</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3076.8544152744207</X>
    <Y>2625.7817118861758</Y>
    <RelX>377.58472553699266</RelX>
    <RelY>-22.4415584415583</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3454.4391408114134</X>
    <Y>2603.3401534446175</Y>
    <RelX>194.29116945107398</RelX>
    <RelY>-58.909090909090082</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3648.7303102624874</X>
    <Y>2544.4310625355274</Y>
    <RelX>87.9809069212406</RelX>
    <RelY>-218.80519480519524</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3736.711217183728</X>
    <Y>2325.6258677303322</Y>
    <RelX>-40.324582338901564</RelX>
    <RelY>-277.71428571428532</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3696.3866348448264</X>
    <Y>2047.9115820160469</Y>
    <RelX>-109.97613365155121</RelX>
    <RelY>-246.85714285714266</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>3586.4105011932752</X>
    <Y>1801.0544391589042</Y>
    <RelX>-373.91885441527484</RelX>
    <RelY>-336.62337662337677</RelY>
    <Angle>0</Angle>
```

```
</Entity>
<Entity Type="Wall">
  <X>3212.4916467780004</X>
  <Y>1464.4310625355274</Y>
  <RelX>-362.92124105011862</RelX>
  <RelY>-210.38961038961043</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2849.5704057278817</X>
  <Y>1254.041452145917</Y>
  <RelX>-40.324582338902474</RelX>
  <RelY>-117.81818181818198</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2809.2458233889793</X>
  <Y>1136.223270327735</Y>
  <RelX>120.97374701670651</RelX>
  <RelY>-224.41558441558402</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>2930.2195704056858</X>
  <Y>911.807685912151</Y>
  <RelX>359.2553699284008</RelX>
  <RelY>-336.62337662337654</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3289.4749403340866</X>
  <Y>575.18430928877444</Y>
  <RelX>168.63007159904555</RelX>
  <RelY>-361.87012987012986</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3458.1050119331321</X>
  <Y>213.31417941864458</Y>
  <RelX>7.3317422434374748</RelX>
  <RelY>-272.10389610389575</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3465.4367541765696</X>
  <Y>-58.789716685251165</Y>
  <RelX>-131.97136038186181</RelX>
  <RelY>-204.77922077922074</RelY>
  <Angle>0</Angle>
</Entity>
<Entity Type="Wall">
  <X>3333.4653937947078</X>
  <Y>-263.5689374644719</Y>
```

```
    <RelX>-366.58711217183782</RelX>
    <RelY>-123.4285714285715</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2966.87828162287</X>
    <Y>-386.9975088930434</Y>
    <RelX>-315.26491646778049</RelX>
    <RelY>25.246753246753258</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2651.6133651550895</X>
    <Y>-361.75075564629014</Y>
    <RelX>-205.28878281622883</RelX>
    <RelY>100.98701298701297</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2446.3245823388606</X>
    <Y>-260.76374265927717</Y>
    <RelX>-131.97136038186181</RelX>
    <RelY>109.40259740259688</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2314.3532219569988</X>
    <Y>-151.36114525668029</Y>
    <RelX>0</RelX>
    <RelY>0</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Wall">
    <X>2314.3532219569988</X>
    <Y>-151.36114525668029</Y>
    <RelX>-87.9809069212406</RelX>
    <RelY>260.88311688311938</RelY>
    <Angle>0</Angle>
  </Entity>
  <Entity Type="Point">
    <X>1619.212410501194</X>
    <Y>1275.6414521459071</Y>
    <RelX>0</RelX>
    <RelY>0</RelY>
    <Angle>0</Angle>
  </Entity>
</Map>
```

-1-3 Agent

CarMazeEnv.py

```
# -*- coding: utf-8 -*-
"""
Created on Thu Jun 14 00:28:13 2018

@author: ajdin
"""
import numpy as np

import pygame
import visualnet
import agent_functions as funs
import novelty

from configs import MapBuilder
from envmanager import EnvManager
from controllers import ControlScheme as control
from modulation import ModulationScheme, ModulationSignal
from rangefinder import RangeFinderGroup as sensors
from net3 import NeuralNetwork3 as nn3
from net3 import Agent
from performance import Performance

###########################################################

black = (0, 0, 0)
white = (255, 255, 255)
red = (255, 0, 0)
redcolor = (0xff,0,0)

def drawText(font, text, pos, display):
  textSurf = font.render(text, False, black)
  display.blit(textSurf, pos)

def drawCenter(pos, display, options):
  x_scale = options['x_scale']
  x_translate = options['x_translate']
  position = tuple(int(x_scale*x+x_translate) for x in pos)
  pygame.draw.circle(display, redcolor, position, 5, 2)


mapbuilder = MapBuilder()

class CarEnv:
  OPTIONS = {'x_scale': 0.2184233207295375,
             'yScale': 0.22767977831846448,
             'x_translate': 133.81512733541817,
             'yTranslate': 138.1115070345661,
             'rotation': 180}

  def __init__(self):
    MAP_BUILDER = MapBuilder()
    self.car = MAP_BUILDER.car
    self.walls = MAP_BUILDER.walls
```

```
    self.center_point = MAP_BUILDER.point
    self.rangefinder_group = sensors(self.car)
    self.performer = Performance(self.rangefinder_group, self.center_point)
    # initialize environment variables
    self.rangefinder_group.update(self.walls)
    self.performance = 0
    self.ticks = 0
    # render variables
    self.options = self.OPTIONS
    self.game_display = None
    self.font = None
    self.clock = None

  def step(self, action):
    if len(action) == 1:
      u_theta, u_speed = action[0], 0.5
      self.car.reset_speed = False
    elif len(action) == 2:
      self.car.reset_speed = True
      u_theta, u_speed = action
    else:
      self.car.reset_speed = False
      u_theta, u_speed = 0.5, 0.5
    self.car.angle += float(u_theta)*self.car.ROTATE_RANGE - self.car.ROTATE_SPEED
    self.car.speed_x += float(u_speed)*self.car.ACCELERATION_RANGE - self.car.ACCELERATION
    self.car.update(self.walls)
    self.rangefinder_group.update(self.walls)
    self.performance += self.performer.normlaps() / 360
    self.ticks += 1
    info = {'position': self.car.center,
        'orientation': self.car.angle,
        'can move': self.car.can_move,
        'time tick': self.ticks}
    return self.rangefinder_group.activations, np.abs(self.performance), False, info

  def reset(self):
    MAP_BUILDER = MapBuilder()
    self.car = MAP_BUILDER.car
    self.rangefinder_group = sensors(self.car)
    self.rangefinder_group.update(self.walls)
    self.ticks = 0
    return self.rangefinder_group.activations

  def draw_environment(self):
    for wall in self.walls:
      wall.draw(self.game_display, self.options)
    self.rangefinder_group.draw(self.game_display, self.options)
    self.car.draw(self.game_display, self.options)

```

```
  def render(self, size=(1600, 1440), fps=45, drawparams={'functions':
      [], 'arguments': []}):
    # Draw the display and objects
    if self.game_display is None:
      pygame.init()
      pygame.font.init()
      self.game_display = pygame.display.set_mode(size)
      pygame.display.set_caption('2D Autonomous Car Driving Simulation')
    if self.font is None:
      self.font = pygame.font.SysFont('Arial', 30)
    if self.clock is None:
      self.clock = pygame.time.Clock()
    self.game_display.fill(white)
    self.draw_environment()
    # Draw the texts
    drawText(self.font, 'tick: '+str(self.ticks), (10, 10), self.
        game_display)
    drawText(self.font, 'FPS: '+"%.1f" % self.clock.get_fps(), (200, 10),
         self.game_display)
    drawText(self.font, 'Performance: '+ '%.2f' % np.abs(self.performance
        ), (400, 10), self.game_display)
    drawText(self.font, 'Forward velocity: ' + '%.1f' % np.linalg.norm(
        list(self.car.speed)),
              (700, 10), self.game_display)
    drawCenter(self.center_point, self.game_display, self.options)
    if self.game_display is not None:
      for function, args in zip(drawparams['functions'], drawparams['
          arguments']):
        function(*args)
    # Render the next frame
    pygame.display.flip()
    self.clock.tick(fps)

  def close(self):
    pygame.quit()


class CarMazeEnv:
  # MAP_BUILDER = MapBuilder()
  DISPLAY_WIDTH = 1000
  DISPLAY_HEIGHT = 800
  hebbian_hyper_parameters = {"num_nodes": [11, 1],
                    "activation_functions": [funs.linu, funs.logistic],
                    "training_functions": [funs.hebbian1],
                    "learning_rates": [1.0],
                    "output_noise": [0, 0.1],
                    "weight_noise": [0.1],
                    "buffers": [None],
                    "tag": 'Pure Hebbian'}

  raahn_hyper_parameters = {"num_nodes": [11, 5, 1],
                    "activation_functions": [funs.linu, funs.logistic,
                        funs.logistic],
```

```
                        "training_functions": [funs.autoencoder ,funs.
                            hebbian1],
                        "learning_rates": [0.1, 1.0],
                        "output_noise": [0, 0, 0.1],
                        "weight_noise": [0, 0.1],
                        "buffers": [novelty.NoveltyBuffer(500), None],
                        "tag": 'novelty-Raahn'}

  grewards_hyper_params = {"num_nodes": [11, 5, 2],
                        "activation_functions": [funs.linu, funs.logistic,
                            funs.logistic],
                        "training_functions": [funs.autoencoder ,funs.
                            hebbian1],
                        "learning_rates": [0.1, 1.0],
                        "output_noise": [0, 0, 0.1],
                        "weight_noise": [0, 0.1],
                        "buffers": [novelty.NoveltyBuffer(500), None],
                        "tag": 'Hebbian with global rewards'}

  def __init__(self):
    # initialize map objects
    self.environment = CarEnv() # --> new!
    # initialize game variables
    self.resolution = (self.DISPLAY_WIDTH, self.DISPLAY_HEIGHT)
    self.game_ext = False
    self.episode = 1
    # initialize algorithm settings
    self.agent = Agent(self.raahn_hyper_parameters)
    self.activations = None
    self.modulation_scheme = ModulationScheme(self.environment.
        rangefinder_group, self.environment.walls)
    # Initialize visual settings
    self.options = self.environment.OPTIONS
    self.bounds = None
    self.calculate_bounds()
    self.network_visualizer = visualnet.NetworkVisualizer(self.
        network_parameters,
                                                           [self.bounds
                                                               [0], 0,
                                                               1600, self.
                                                               bounds[1]])

    self.crashes = []
    self.performances = []
    self.avg_objectives = []
    pygame.init()
    pygame.font.init()

  def step(self):
    # num_crashes = self.env.car.num_crashes
    # control.range_finder_control(self.network, self.rangefinder_group,
        angle_control=True)
    observation = self.environment.rangefinder_group.activations
```

```
    # self.network.add_experience(observation)
    self.activations = self.agent.propagate(observation)
    # action = tuple(self.network.activations[-1])
    action = tuple(self.activations[-1])
    self.environment.step(action)
    self.modulation_scheme.wall_avoidance()
    modulation = self.modulation_scheme.modulations[0]
    # self.network.modulation_signal.modulations[0] = modulation
    # self.network.train()
    self.agent.train(modulation, self.activations)

  def convergence_test(self):
    if self.episode == 15:
      self.game_ext = True
    observation = self.environment.rangefinder_group.activations
    self.activations = self.agent.propagate(observation)
    action = tuple(self.activations[-1])
    self.environment.step(action)
    self.modulation_scheme.wall_avoidance2()
    global_reward = self.environment.performer.objective_func()
    self.avg_objectives.append(global_reward)
    modulation = self.modulation_scheme.modulations[0]
    if self.environment.ticks == 1200:
      print(np.mean(self.avg_objectives))
      self.avg_objectives = []
      del self.environment
      self.environment = CarEnv()
      self.modulation_scheme = ModulationScheme(self.environment.rangefinder_group, self.environment.walls)
      self.episode += 1
    else:
      self.agent.train(modulation, self.activations)


  def crashcounting_test(self):
    if self.episode >= 50 or self.environment.ticks == 10000:
      self.game_ext = True
      self.performances.append(self.environment.performance)
      self.crashes.append(self.environment.car.num_crashes)
      print('Episode: ', self.episode, ' Num crashes: ', self.crashes[-1],
            ' Perf.: ', abs(self.performances[-1]), '\n')
    # num_crashes = self.environment.car.num_crashes
    observation = self.environment.rangefinder_group.activations
    self.activations = self.agent.propagate(observation)
    action = tuple(self.activations[-1])
    self.environment.step(action)
    self.modulation_scheme.wall_avoidance()
    global_reward = self.environment.performer.objective_func()
    self.avg_objectives.append(global_reward)
    modulation = self.modulation_scheme.modulations[0]
    if self.environment.car.num_crashes > 0 and self.environment.ticks >= 2500:
```

```
      self.crashes.append(self.environment.car.num_crashes)
      self.performances.append(self.environment.performance)
      print('Episode: ', self.episode, ' Num crashes: ', self.crashes[-1],
            ' Perf.: ', abs(self.performances[-1]), '\n')
      del self.environment
      self.environment = CarEnv()
      self.modulation_scheme = ModulationScheme(self.environment.rangefinder_group, self.environment.walls)
      self.episode += 1
    else:
      self.agent.train(modulation, self.activations)

    # self.Reward = self.envmanager.Reward
    # self.avg_objectives.append(self.Reward)
#     if self.car.num_crashes > num_crashes and self.ticks >= 1000:
#       num_crashes = self.car.num_crashes
#       self.performances.append(self.performance)
#       self.crashes.append(num_crashes)
#       self.reset()
#       self.performance = 0
#       print(self.crashes)
#     elif self.episode >= 100 or self.ticks == 11000:
#       self.game_ext = True
#       self.performances.append(self.performance)
#       self.crashes.append(self.car.num_crashes)
#     else:
#       self.envmanager.handle_env(training=True)
#     if self.ticks == 1000:
#       self.performances.append(self.performance)
#       self.crashes.append((self.episode, num_crashes))

  def reset(self, keep_performance=True):
    MAP_BUILDER = MapBuilder()
    if keep_performance:
      performances = self.performances
    self.car = MAP_BUILDER.car
    self.rangefinder_group = sensors(self.car)
    self.rangefinder_group.update(self.walls)
    self.envmanager = EnvManager(self)
    self.performances = performances
    self.episode += 1
    self.ticks = 0
    return self.rangefinder_group.activations

  def render(self, size=(None)):
    drawparams = {'functions': [], 'arguments': []}
    if self.environment.game_display is not None:
      text_args1 = (self.environment.font, 'Modulation: ' + '%.3f' % self.modulation_scheme.modulations[0],
                    (400, 40), self.environment.game_display)
      text_args2 = (self.environment.font, 'Episode: ' + str(self.episode),
```

```
                    (10,40), self.environment.game_display)
      text_args3 = (self.environment.font, 'Global Reward: ' + '%.2f' % self.avg_objectives[-1],
                    (400, 70), self.environment.game_display)
      visual_args = (self.network_parameters, self.environment.game_display)
      drawparams = {'functions': [drawText, drawText, drawText, self.network_visualizer.visualize],
                    'arguments': [text_args1, text_args2, text_args3, visual_args]}
      # self.network_visualizer.visualize(self.network_parameters, self.environment.game_display, self.environment.font)
    self.environment.render(drawparams=drawparams)

  def close(self):
    self.game_ext = True
    self.environment.close()

  def calculate_bounds(self):
    if not self.bounds:
      xbound = 0
      ybound = 0
      for wall in self.environment.walls:
        maxboundWallx = max(wall.startPoint[0], wall.endPoint[0])
        maxboundWally = max(wall.startPoint[1], wall.endPoint[1])
        xbound = max(xbound, maxboundWallx)
        ybound = max(ybound, maxboundWally)
      xbound = self.options['x_scale']*xbound+self.options['x_translate']
      ybound = self.options['x_scale']*ybound+self.options['x_translate']
      self.bounds = (xbound, ybound)

  @property
  def network_parameters(self):
    network_parameters = {'num_nodes': self.agent.num_neurons,
                          'weights': self.agent.weights,
                          'activations': self.activations,
                          'weight_cap': self.agent.weight_cap}
    return network_parameters


  def visualize_network(self, display, options):
    self.network_visualizer.visualize(display)

#%% Experiments with supervised Hebbian learning
#from collections import deque
#
#if __name__== "__main__":
#  supervised_params = {"num_nodes": [11, 15, 15, 5],
#                   "activation_functions": [funs.linu, np.sign, np.sign, np.sign],
#                   "training_functions": [funs.hebbian3, funs.hebbian3, funs.hebbian3],
#                   "learning_rates": [1.0, 1.0, 1.0],
```

```
#                   "output_noise": [0, 0, 0, 0],
#                   "weight_noise": [0, 0, 0],
#                   "buffers": [None, None, None]}
#   agent = Agent(supervised_params)
#   shape = (supervised_params["num_nodes"][-1], supervised_params["
    num_nodes"][0])
#   weight_des = np.random.uniform(-1,1, shape)
#   errors = deque(maxlen=100)
#   for i in range(10000):
#     activations = agent.propagate(np.random.choice([-1,1], (11,)))
#     output_des = weight_des @ activations[0]
#     output = activations[-1]
#     distance1 = np.abs(output-output_des)
#     modulation1 = np.multiply(distance1, output)
#     distance2 = np.linalg.norm(output-output_des)
#     modulation2 = distance2*output
#     agent.train(modulation2
#                 , activations)
#     errors.append(distance1)
#     if i%20==0:
#       print('error:', np.mean(errors))
```

agent_functions.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Mar  6 15:22:56 2018

@author: ajdin
"""
import numpy as np

# Activation functions
def linu(x):
  #return arg
  return x

def step(x):
  #step function
  return 1 * (x > 0)

def relu(x):
  #rectified linear unit
  return np.maximum(x, 0)

def logistic(x):
  return 1.0 / (1.0 + np.exp(-x))

def logistic_derivative(x):
  return x * (1.0 - x)

#tangent hyperbolic
tanh = np.tanh
```

```
#Exponentailly Varied Weight Adjustment
def evwa(self, learning_rate, reward):
  rho = 0.02
  adjustment = []
  for i in range(self.num_layers):
    adjustment.append(learning_rate*(rho**reward))
  return adjustment

#Linearly Varied Weight Adjustment
def lvwa(self, learning_rate, reward):
  adjustment = []
  for i in range(self.num_layers):
    adjustment.append(learning_rate*(1-reward))
  return adjustment

# Compute node values in layers
def layer_output(activation_fun, parameters, input_nodes, noise_magtd
    =0.0):
  output_nodes = activation_fun(np.dot(parameters, input_nodes))
  if noise_magtd:
    output_nodes += np.random.uniform(-noise_magtd, noise_magtd, np.shape
        (output_nodes))
  return output_nodes

# train weights
def train(mod_vector, input_samples, output_samples, learning_rate):
  for i in range(len(mod_vector)):
    modulation = mod_vector[i]
    input_sample = np.reshape(input_samples[i], (-1,1))
    output_sample = np.reshape(output_samples[i], (-1,1))
    plasticity = output_sample @ input_sample.T
    if i == 0:
      weight_delta = learning_rate*modulation*plasticity
    else:
      weight_delta += learning_rate*modulation*plasticity
  return weight_delta

def hebbian1(layer_parameters, presynaptic, postsynaptic):
  lr = layer_parameters["learning_rate"]
  modulation = layer_parameters["modulation"]
  noise_magtd = layer_parameters["weight_noise"]
  presynaptic = np.array(presynaptic)
  postsynaptic = np.array(postsynaptic)*2-1
  if presynaptic.ndim == 1:
    presynaptic = np.reshape(presynaptic, (-1,1))
  if postsynaptic.ndim == 1:
    postsynaptic = np.reshape(postsynaptic, (-1,1))
  update = lr*modulation*(postsynaptic @ presynaptic.T)
  if noise_magtd:
    update += np.random.uniform(-noise_magtd, noise_magtd, np.shape(
        update))
  return update
```

```
def hebbian2(layer_parameters, presynaptic, postsynaptic):
  lr = layer_parameters["learning_rate"]
  modulation = layer_parameters["modulation"]
  noise_magtd = layer_parameters["weight_noise"]
  presynaptic = np.array(presynaptic)
  postsynaptic = np.array(postsynaptic)
  if presynaptic.ndim == 1:
    presynaptic = np.reshape(presynaptic, (-1,1))
  if postsynaptic.ndim == 1:
    postsynaptic = np.reshape(postsynaptic, (-1,1))
  update = lr*modulation*(postsynaptic @ presynaptic.T)
  if noise_magtd:
    update += np.random.uniform(-noise_magtd, noise_magtd, np.shape(
        update))
  return update

def hebbian3(layer_parameters, presynaptic, postsynaptic):
  """
  Supervised-type hebbian algorithm.
  Modulation equals the 2 norm of the discrepancy between actual and
      desired output,
  times the desired output.
  """
  lr = layer_parameters["learning_rate"]
  modulation = layer_parameters["modulation"]
  noise_magtd = layer_parameters["weight_noise"]
  presynaptic = np.array(presynaptic)
  postsynaptic = np.array(postsynaptic)
  if presynaptic.ndim == 1:
    presynaptic = np.reshape(presynaptic, (-1,1))
  if postsynaptic.ndim == 1:
    postsynaptic = np.reshape(postsynaptic, (-1,1))
  modulation.shape = postsynaptic.shape
  update = lr*(modulation @ presynaptic.T)
  if noise_magtd:
    update += np.random.uniform(-noise_magtd, noise_magtd, np.shape(
        update))
  return update


def autoencoder(layer_parameters, presynaptic, postsynaptic):
  lr = layer_parameters["learning_rate"]
  weights = layer_parameters["weights"]
  noise_magtd = layer_parameters["weight_noise"]
  presynaptic = np.array(presynaptic)
  postsynaptic = np.array(postsynaptic)
  if presynaptic.ndim == 1:
    presynaptic = np.reshape(presynaptic, (-1,1))
  if postsynaptic.ndim == 1:
    postsynaptic = np.reshape(postsynaptic, (-1,1))
  reconstruction = logistic(weights.T @ postsynaptic)
  error = presynaptic - reconstruction
```

```
  deltas = np.multiply(error, logistic_derivative(reconstruction))
  back_prop_deltas = np.multiply(logistic_derivative(postsynaptic), (
      weights @ deltas))
  error_weight_delta = lr*(postsynaptic @ deltas.T)
  backprop_weight_delta = lr*(back_prop_deltas @ presynaptic.T)
  weight_delta = error_weight_delta + backprop_weight_delta
  if noise_magtd:
    weight_delta += np.random.uniform(-noise_magtd, noise_magtd, np.shape
        (weight_delta))
  return weight_delta

def select_several(layer_parameters):
  novelty_buffer = layer_parameters['buffers']
  # error = 0.0
  samples = [] #linkedList
  experiences = []
  for occupant in iter(novelty_buffer):
    samples.append(occupant.experience)
  sample_count = min(20, len(novelty_buffer))
  for i in range(sample_count):
    index = np.random.randint(len(samples))
    sample = samples.pop(index)
    experiences.append(np.array(sample))
  return experiences
```

modulation.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Aug 19 18:40:16 2018

@author: ajdin
"""
import numpy as np

import buffer

from rangefinder import RangeFinder
from lines import get_dist
from collections import deque


class ModulationSignal:
  NO_MODULATION = 0.0

  def __init__(self):
    self.modulations = []

  def add_signal(self, *args):
    if len(args) == 0:
      self.modulations.append(self.NO_MODULATION)
    elif len(args) == 1:
      self.modulations.append(args[0])
```

```python
    return len(self.modulations) - 1


class ModulationScheme:
  MODULATION_STRENGTH = 1.0
  MODULATION_RESET = 0.0
  MODULATION_NOT_RESET = -1.0
  PERPENDICULAR = 90.0
  SCHEME_STRINGS = ['WallAvoidance', 'Acceleration', 'NaiveWallAvoidance', 'NaiveAcceleration']

  def __init__(self, rangefinder_group, walls):
    self.rangefinder_group = rangefinder_group
    self.previous_fitness = None
    self.N_size = 5
    self.reward_history = deque(maxlen=self.N_size)
    self.Reward = 0
    self.acceleration
    self.walls = walls
    self.view_distance = 400.0
    self.viewline = RangeFinder(self.rangefinder_group.car, self.view_distance)
    self.modulations = [0.0]*len(self.SCHEME_STRINGS)
    # Initializes for no former experience
    self.last_angle_between = None
    self.last_wall_in_range = None
    self.compare_wall = None
    self.last_nearest_dist = None

  def wall_avoidance2(self):
    """
    Minimize the average sensor values. Total reward is Negative mean of sensor outputs.
    Change in gradient (objective increase) equals the negative mean sensor values of current time
    tick, minus the negative mean sensor values of previous time tick.
    """
    params = (self.rangefinder_group.activations, self.rangefinder_group.car.speed)
    self.reward_history.append(self.local_reward(*params))
    self.modulations[0] = 0.0

    if len(self.reward_history)==self.reward_history.maxlen:
      change_in_objective =  self.reward_history[-1] - self.Reward
      self.modulations[0] = np.tanh(10*change_in_objective)
      self.modulations[0] = change_in_objective
    self.Reward = self.reward_history[0]

  def local_reward(self, *params):
    activations, speed = params
    avg_distance = np.mean(1 - activations[5])
    norm_speed = np.linalg.norm(speed)
    local_reward = (1.3*avg_distance + 0.3*norm_speed)/self.N_size
```

```
    return local_reward

  def modulate(self):
    """
    Modulates all schemes ever defined, so that a combination of schemes may be integrated.
    """
    self.viewline.update_position()
    walls_in_bounds = self.viewline.entities_in_bounds(self.walls)
    last_angle = None
    nearest_wall = None
    nearest_dist = self.view_distance
    # Get nearest wall or None
    for wall in walls_in_bounds:
      intersections = self.viewline.intersects(wall)
      if len(intersections) > 0:
        dist = get_dist(self.rangefinder_group.car.center, intersections[0])
        if dist < nearest_dist:
          nearest_dist = dist
          nearest_wall = wall

    # No previous wall yields nothing to modulate
    if not self.last_wall_in_range:
        self.modulations[0] = 0.0
        self.modulations[1] = 0.0
    else:
      angle_between = self.viewline.angle_between(self.last_wall_in_range)
      last_intersection = self.viewline.intersects(self.last_wall_in_range)[0]
      distance = get_dist(self.rangefinder_group.car.center, last_intersection)
      delta = angle_between - self.last_angle_between
      gamma = distance - self.last_nearest_dist
      if angle_between > self.PERPENDICULAR:
        modulation1 = self.MODULATION_STRENGTH*delta / self.rangefinder_group.car.ROTATE_SPEED
      else:
        modulation1 = -self.MODULATION_STRENGTH*delta / self.rangefinder_group.car.ROTATE_SPEED
      modulation2 = self.MODULATION_STRENGTH*gamma / self.rangefinder_group.car.ACCELERATION
      self.modulations[0] = modulation1
      self.modulations[1] = modulation2

    # No current wall yields no angle to store
    if nearest_wall:
      last_angle = self.viewline.angle_between(nearest_wall)
    # Store last wall and angle in any case (may be None)
    self.last_angle_between = last_angle
    self.last_nearest_dist = nearest_dist
    self.last_wall_in_range = nearest_wall
```

```python

  def wall_avoidance(self):
    # print('computing wall avoidance modulation: ', self.modulations[0])
    self.viewline.update_position()
    walls_in_bounds = self.viewline.entities_in_bounds(self.walls)
    compare_wall = None
    nearest_dist = self.viewline.default_length
    # Get the nearest wall in the view_distance if any
    for wall in walls_in_bounds:
      intersections = self.viewline.intersects(wall)
      if len(intersections) > 0:
        dist = get_dist(self.rangefinder_group.car.center, intersections
            [0])
        if dist < nearest_dist:
          nearest_dist = dist
          compare_wall = wall
    self.compare_wall = compare_wall
    # The angle to use for modulation. Should never be zero when the
      angle delta is calculated.
    # If it is, then there must be a bug.
    angle_between = 0.0
    new_last_angle = self.MODULATION_NOT_RESET

    # If there is no nearest wall
    if not compare_wall:
      # If there is no previous wall, set the modulation to zero and
        reset the last angle
      if not self.last_wall_in_range:
        if self.last_angle_between != self.MODULATION_RESET:
          self.last_angle_between = self.MODULATION_RESET
          self.modulations[0] = 0.0
        # nothing to modulate, and nothing to save, Don't Continue
        return
      # Just left a wall
      else:
        angle_between = self.viewline.angle_between(self.
            last_wall_in_range)

    # If the wall has changed
    elif compare_wall != self.last_wall_in_range:
      # There was a last wall that is different from the current wall
      if self.last_wall_in_range:
        angle_between = self.viewline.angle_between(self.
            last_wall_in_range)
        new_last_angle = self.viewline.angle_between(compare_wall)
      # It is the first time any wall was hit, don't continue
      # Save the angle between last and current wall
      else:
        angle_between = self.viewline.angle_between(compare_wall)
        self.last_angle_between = angle_between
        self.last_wall_in_range = compare_wall
        return

```

```python
    # The usual case, the last wall is equal to the current wall
    else:
      angle_between = self.viewline.angle_between(compare_wall)
      new_last_angle = angle_between
    delta = angle_between - self.last_angle_between
    modulation = self.MODULATION_STRENGTH

    if angle_between > self.PERPENDICULAR:
      modulation *= delta / self.rangefinder_group.car.ROTATE_SPEED
    else:
      modulation *= -delta / self.rangefinder_group.car.ROTATE_SPEED
    self.modulations[0] = modulation
    self.last_angle_between = new_last_angle
    self.last_wall_in_range = compare_wall

  def performance_increase(self):
    pass

  def naive_wall_avoidance(self):
    print(self.modulations)
    self.modulations[0] = self.last_forward_activation - self.rangefinder_group.activations[5]
    self.last_forward_activation = self.rangefinder_group.activations[5]
    pass

  def naive_acceleration(self):
    # if self.last_wall_in_range:
    # intersections = self.viewline.intersects(self.last_wall_in_range)
    self.modulations[1] = 1-2*abs(self.modulations[0])
    # print('computing acceleration modulation: ', self.modulations[1])
    return

  def acceleration(self):
    # If there is no nearest wall in range
    if not self.last_wall_in_range:
      if self.last_nearest_dist:
        self.last_nearest_dist = None
        self.modulations[1] = 0.0
      return
    else:
      intersection = self.viewline.intersects(self.last_wall_in_range)[0]
      nearest_dist = get_dist(self.rangefinder_group.car.center, intersection)

    pass

  SCHEMES = [wall_avoidance, acceleration, naive_wall_avoidance, naive_acceleration]

  @staticmethod
  def get_scheme_from_string(scheme_string):
    for i in range(len(ModulationScheme.SCHEME_STRINGS)):
      if scheme_string == ModulationScheme.SCHEME_STRINGS[i]:
```

```
            return i
        return -1

    @staticmethod
    def getSchemeFunction(scheme):
        if scheme >= 0 and scheme < len(ModulationScheme.SCHEMES):
            return ModulationScheme.SCHEMES[scheme]
        else:
            return None

    def reset(self):
        self.last_angle_between = self.MODULATION_RESET
        self.last_wall_in_range = None
```

net3.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri Oct 12 12:52:38 2018

@author: ajdin
"""

import numpy as np
import configs
import novelty

# from RaahnXmlConfig import NeuralNetworkConfig, LayerConfig
from collections import deque

from functions import Activation
from training import TrainingMethod
from modulation import ModulationScheme, ModulationSignal
from buffer import Buffer
from eligibility import Eligibility

class NeuralNetwork3:
    """
    Creates a NeuralNetwork that strictly uses three layers at max:
        inputlayer, hiddenlayer,
    and outputlayer. hiddenlayer is optional.
    parameters (optional keyword arguments):
        inputcount: number of neurons in the input layer
        hiddencount: number of neurons in the hidden layer.
        outputcount: number of neurons in the output layer

    attributes:
        activations: list of neuron values for each layer
        layers: list of NetworkLayers containing the neuron connections
        output_noise_mag: magnitude of random noise to be applied after
            computing activation
        weight_noise_mag: magnitude of random noise to be applied after
            updating weights
```

```
    weight_cap: weights will not grow beyond upper limit weight_cap and lower limit -weight_cap.
    # use_novelty: boolean determing whether a novelty buffer will be used.

  methods:
    set_up: initializes the NeuralNetwork3 layers based on parameters arguments.
    configure: configures hyper-parameters of the network, based on RaahnXmlConfig.
    init_neurons: returns numpy array of neurons with values 0.0
    add_experience: adds a sample experience to the network's input layer
    propagate_signal: maps the input activation to the output activation
    train: trains all connections that are present within the network
  """
  WEIGHT_SCALE = 6.0
  ACTIVATION = Activation.logistic
  ACTIVATION_DERIVATIVE = Activation.logistic_derivative
  TRAINING_METHODS = {'Hebbian': TrainingMethod.hebbian_learning,
                      'HebbianHistory': TrainingMethod.hebbian_history_learning,
                      'HebbianEligibility': TrainingMethod.hebbian_eligibility,
                      'HebbianLongterm': TrainingMethod.hebbian_longterm,
                      'HebbianEpisodic': TrainingMethod.hebbian_episodic,
                      'NoTraining': TrainingMethod.no_training,
                      'Autoencoder': TrainingMethod.sparse_autoencoder,
                      'LinAutoencoder': TrainingMethod.linear_autoencoder}
  MOD_SCHEMES = {'WallAvoidance': ModulationScheme.wall_avoidance,
                 'WallAvoidance2': ModulationScheme.wall_avoidance2}

  def __init__(self, **kwargs):
    self.activations = []
    self.layers = []
    # set up the layers if specified
    self.set_up(**kwargs)
    # configure hyper-parameters of the network.
    self.configure()
    # ready up for training
    self.compile_network()

  @classmethod
  def default(cls, filename=None):
    if filename:
      config = configs.Net3Config(filename)
    else:
      config = configs.Net3Config()
    net3 = NeuralNetwork3()
```

```
        net3.activations.append(net3.init_neurons(config.input_count))

        return config


    def set_up(self, **kwargs):
        inputcount = kwargs.get('inputcount')
        hiddencount = kwargs.get('hiddencount')
        outputcount = kwargs.get('outputcount')
        if self.valid_neuron_count(inputcount) and self.valid_neuron_count(
            outputcount):
            self.activations.append(self.init_neurons(inputcount))
            if self.valid_neuron_count(hiddencount):
                self.activations.append(self.init_neurons(hiddencount))
                add_layer = NetworkLayer(self, hiddencount)
                self.layers.append(add_layer)
            self.activations.append(self.init_neurons(outputcount))
            add_layer = NetworkLayer(self, outputcount)
            self.layers.append(add_layer)

    def valid_neuron_count(self, count):
        if isinstance(count, int):
            if count > 0:
                return True
        return False

    def configure(self, output_noise_mag=0.1, weight_noise_mag=0.1):
        self.output_noise_mag = output_noise_mag
        self.weight_noise_mag = weight_noise_mag
        self.weight_cap = 10.0
        self.modulation_signal = ModulationSignal()
        self.modulation_signal.add_signal()
        #activation functions
        self.activation_func = NeuralNetwork3.ACTIVATION
        self.activation_deriv = NeuralNetwork3.ACTIVATION_DERIVATIVE
        # specify buffer settings if any
        self.initlen = 1
        self.maxlen = 200
        self.growth = 1.0

    def init_neurons(self, count):
        return np.ones(count)*0.0

    def add_experience(self, experience):
        # add experience to the very first layer, and propagate it
        self.activations[0] = np.array(experience)
        self.propagate_signal()
        # add experience to any buffers if specified
        for layer in self.layers:
            input_sample = np.array(layer.network.activations[layer.
                current_layer])
            if not layer.usenovelty:
```

```python
          output_sample = np.array(layer.network.activations[layer.current_layer + 1])
          sample = (input_sample, output_sample)
          layer.history_buffer.append(sample)
        # layer.estim_window.add_sample(sample)
        if layer.usenovelty:
          new_occupant = novelty.NoveltyOccupant(experience=input_sample)
          new_distances = layer.novelty_buffer.compute_new_distances(new_occupant)
          if len(layer.novelty_buffer) == layer.history_size:
            least_novel = layer.novelty_buffer[0]
            if new_occupant.novelty_score > least_novel.novelty_score:
              layer.novelty_buffer.remove_novelty(least_novel)
              layer.novelty_buffer.add_novelty(new_occupant, new_distances)
          else:
            layer.novelty_buffer.add_novelty(new_occupant, new_distances)

    def propagate_signal(self):
      for layer in self.layers:
        layer.propagate_signal()

    def compile_network(self, training_method='HebbianLongterm', modulation_scheme='WallAvoidance2'):
      for layer in self.layers:
        layer.training_method = NeuralNetwork3.TRAINING_METHODS[training_method]
        layer.modulation_scheme = NeuralNetwork3.MOD_SCHEMES[modulation_scheme]
        # layer.history_buffer = Buffer(self.initlen, maxlen=self.maxlen, growth_fact=self.growth)
        layer.learning_rate = 1.0
      if len(self.layers) > 1:
        self.layers[0].usenovelty = True
        self.layers[0].history_size = 500
        self.layers[0].samples_per_tick = 20
        self.layers[0].training_method = NeuralNetwork3.TRAINING_METHODS['Autoencoder']
        self.layers[0].learning_rate = 0.1
        self.layers[0].novelty_buffer = novelty.NoveltyBuffer(self.layers[0].history_size)
        self.layers[0].errorbuffer = deque(maxlen=self.layers[0].history_size)
        self.layers[1].history_buffer = deque(maxlen=self.initlen)

    def train(self):
      for layer in self.layers:
        layer.train()


class NetworkLayer:
  """
  Creates a fully connected NetworkLayer in a given NeuralNetwork3
```

```
    parameters:
      network: NeuralNetwork3 in which the NetworkLayer is added
      neuron_count: number of neurons in current layer
      learning_rate: sets the learning rate for training (optional), default is 1.0
      training_method: set the training method for current layer, None value averts training

    attributes:
      current_layer: denotes the index of the current layer
      shape: denotes the dimension that the weights Matrix will assume
      weights: weight matrix, used to map the preceding neurons linearly to the current neurons

    """
    def __init__(self, network, neuron_count, learning_rate=1.0, training_method=None):
      self.network = network
      self.current_layer = len(self.network.layers)
      self.neuron_count = neuron_count
      self.usenovelty = False
      self.learning_rate = learning_rate
      self.training_method = training_method
      self.history_buffer = deque(maxlen=1)
      self.history_size = 1
      # self.estim_window = Buffer(140, maxlen=140)
      self.averages = np.zeros((neuron_count, 1))
      # initialize the weights
      self.init_weights()
      # self.elig_trace = Eligibility(self)

      self.average_error = 0.0

    def init_weights(self):
      input_count = len(self.network.activations[self.current_layer])
      self.shape = (self.neuron_count, input_count)
      total_neurons = sum(self.shape)
      range_ = np.sqrt(self.network.WEIGHT_SCALE / total_neurons)
      self.weights = np.random.uniform(-range_, range_, self.shape)

    def propagate_signal(self):
      forwarded_activation = self.weights @ self.network.activations[self.current_layer]
      magnitude = self.network.output_noise_mag
      noise = np.random.uniform(-magnitude, magnitude, forwarded_activation.shape) if not self.usenovelty else 0
      forwarded_activation = NeuralNetwork3.ACTIVATION(forwarded_activation) + noise
      self.network.activations[self.current_layer + 1] = forwarded_activation

    def train(self):
      if self.usenovelty:
```

```
      error = self.train_several()
      self.errorbuffer.append(error)
      self.average_error = np.mean(self.errorbuffer)
    else:
      return self.training_method(self, self.network.modulation_signal.modulations[0])

  def train_several(self):
    error = 0.0
    samples = [] #linkedList
    for occupant in iter(self.novelty_buffer):
      samples.append(occupant.experience)
    sample_count = min(self.samples_per_tick, len(self.novelty_buffer))
    for i in range(sample_count):
      index = np.random.randint(len(samples))
      sample = samples.pop(index)
      self.network.activations[0] = np.array(sample)
      self.network.propagate_signal()
      self.update_averages()
      error += self.training_method(self)
    error /= sample_count
    return error

  def train_recent(self):
    return

  def update_averages(self):
    exponent = 1.0 / self.history_size
    decay = 0.01**exponent
    neurons = np.reshape(self.network.activations[self.current_layer + 1], (-1,1))
    self.averages = (decay*self.averages) + (1.0-decay)*neurons

#%%
import agent_functions as funs
# import novelty

hyper_parameters = {"num_nodes": [4, 12, 24, 1],
                    "activation_functions": [funs.linu, np.tanh, np.tanh, np.tanh],
                    "training_functions": [funs.hebbian2, funs.hebbian2, funs.hebbian2],
                    "learning_rates": [1.0, 1.0, 1.0],
                    "output_noise": [0, 0, 0, 0.1],
                    "weight_noise": [0, 0, 0.1],
                    "buffers": [novelty.NoveltyBuffer(500),None,None]}

class Agent:
  def __init__(self, hyper_params):
    self.tag = hyper_params.get('tag', 'Not Specified')
    self.num_layers = len(hyper_params["num_nodes"]) - 1
    self.num_neurons = hyper_params["num_nodes"]
    self.activation_funs = hyper_params["activation_functions"]
```

```
    self.training_funs = hyper_params["training_functions"]
    self.learning_rates = hyper_params["learning_rates"]
    self.buffers = hyper_params["buffers"]
    self.output_noise = hyper_params["output_noise"]
    self.weight_noise = hyper_params["weight_noise"]
    self.weight_cap = 10
    self.weights = []
    for i in range(self.num_layers):
      randomw = np.random.randn(self.num_neurons[i+1], self.num_neurons[i])
      self.weights.append(randomw)
    print('Initialized Agent: ' + self.tag + '\n')

  def propagate(self, experience):
    input_activation = np.array(experience)
    activations = []
    activations.append(self.activation_funs[0](input_activation))
    for i in range(self.num_layers):
      activations.append(funs.layer_output(self.activation_funs[i+1], self.weights[i],
                                           activations[i], noise_magtd=self.output_noise[i+1]))
      if self.buffers[i] is not None:
        self.update_novelty(activations[i], self.buffers[i])
    return activations

  def update_novelty(self, experience, novelty_buffer):
    input_sample = np.array(experience)
    new_occupant = novelty.NoveltyOccupant(experience=input_sample)
    new_distances = novelty_buffer.compute_new_distances(new_occupant)
    if len(novelty_buffer) == novelty_buffer.history_size:
      least_novel = novelty_buffer[0]
      if new_occupant.novelty_score > least_novel.novelty_score:
        novelty_buffer.remove_novelty(least_novel)
        novelty_buffer.add_novelty(new_occupant, new_distances)
    else:
      novelty_buffer.add_novelty(new_occupant, new_distances)

  def train(self, modulation, activations):
    for i in range(self.num_layers):
      layer_parameters = {'weights': self.weights[i],
                          'weight_noise': self.weight_noise[i],
                          'learning_rate': self.learning_rates[i],
                          'buffers': self.buffers[i],
                          'modulation': modulation}
      if self.buffers[i] is not None:
        experiences = funs.select_several(layer_parameters)
        for experience in experiences:
          postsynaptic = funs.layer_output(self.activation_funs[i+1], self.weights[i],
                                           experience, noise_magtd=self.output_noise[i+1])
```

```
          delta_weights = self.training_funs[i](layer_parameters, experience, postsynaptic)
          self.weights[i] = np.clip(self.weights[i] + delta_weights, -self.weight_cap, self.weight_cap)
          layer_parameters['weights'] = self.weights[i]
      else:
        delta_weights = self.training_funs[i](layer_parameters, activations[i], activations[i+1])
        self.weights[i] = np.clip(self.weights[i] + delta_weights, -self.weight_cap, self.weight_cap)


# --------------------------------
if __name__ == "__main__":
  agent = Agent(hyper_parameters)
  rand_exp = np.random.uniform(0,1, (4,))
  activations = agent.propagate(rand_exp)
  for i in range(agent.num_layers):
    print("weight shape: ", agent.weights[i].shape)
    print("activation shape: ", activations[i+1].shape, "\n")
```

performance.py

```
# -*- coding: utf-8 -*-
"""
Created on Sun Oct 14 21:05:14 2018

@author: ajdin
"""
import numpy as np

from collections import deque

class Performance:

  def __init__(self, rangefinder_group, center):
    self.rangefinder_group = rangefinder_group
    self.center_point = center
    self.N_size = 5
    self.gamma = 1.0
    self.reward_history = deque(maxlen=self.N_size)
    self.Reward = 0

  def normlaps(self):
    center2_last = tuple(np.subtract(self.rangefinder_group.car.lastpos, self.center_point))
    center2_current = tuple(np.subtract(self.rangefinder_group.car.center, self.center_point))
    cur_angle = np.arctan2(*center2_current[::-1])
    last_angle = np.arctan2(*center2_last[::-1])
    # anti-clockwise gives positive performance
    angle_diff = np.degrees(cur_angle - last_angle)
    if np.abs(angle_diff) > 90:
```

```
      change = 360 - np.abs(angle_diff)
      if angle_diff < 0.0:
        return change
      else:
        return -change
    return angle_diff

  def objective_func(self):
    params = (self.rangefinder_group.activations, self.rangefinder_group.
        car.speed)
    self.reward_history.append(self.local_reward(*params))
    weighted_avg = np.multiply(np.array(self.reward_history), self.gamma
        **np.arange(len(self.reward_history)))
    return np.mean(weighted_avg)

  def local_reward(self, *params):
    activations, speed = params
    avg_distance = np.mean(1 - activations[5])
    norm_speed = np.linalg.norm(speed)
    local_reward = (1.3*avg_distance + 0.3*norm_speed)
    return local_reward
```

visualnet.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Sep 15 12:51:39 2018

@author: ajdin
"""
import pygame
import numpy as np

RADIUS = 25
THICKNESS = 2
BLACK = (0, 0, 0)
WHITE = (255, 255, 255)

class NetworkVisualizer:
  FONT = 'Arial'
  FONT_SIZE = 18

  def __init__(self, network, box):
    self.network = network
    self.box = box #(lower_x, lower_y, upper_x, upper_y)
    self.neuron_descriptions = []
    self.connection_descriptions = []
    self.layer_count = len(network.layers) + 1

    self.set_up()
    self.set_up_grid()
    self.set_up_connections()

```

```
    pygame.font.init()
    self.font = pygame.font.SysFont(self.FONT, self.FONT_SIZE, bold=True)

  def set_up(self):
    for i in range(self.layer_count):
      neuron_description = NeuronGroupDescription(self.network, i)
      self.neuron_descriptions.append(neuron_description)

  def set_up_grid(self):
    y_values = np.linspace(self.box[1], self.box[3], self.layer_count+1, endpoint=False)[1:]
    for i in range(self.layer_count):
      self.neuron_descriptions[i].y_value = y_values[i]
      neuron_count = len(self.neuron_descriptions[i].values)
      self.neuron_descriptions[i].x_values = np.linspace(self.box[0], self.box[2], \
                                                          neuron_count+1, endpoint=False)[1:]

  def set_up_connections(self):
    for i in range(self.layer_count - 1):
      connection_description = ConnectionDescription(self.network, i)
      connection_description.input_neuron_description = self.neuron_descriptions[i]
      connection_description.output_neuron_description = self.neuron_descriptions[i+1]
      self.connection_descriptions.append(connection_description)

  def visualize(self, display):
    self.draw_connections(display)
    self.draw_neurons(display)
    self.draw_legend(display)

  def draw_neurons(self, display):
    for i in range(self.layer_count):
      self.neuron_descriptions[i].draw(display, self.font)

  def draw_connections(self, display):
    for i in range(len(self.connection_descriptions)):
      self.connection_descriptions[i].draw(display, self.font)

  def draw_legend(self, display):
    y_offset = 50
    bar_length = 400
    box_center = (self.box[0]+self.box[2]) / 2
    x_start = box_center - 1/2*bar_length
    x_end = box_center + 1/2*bar_length
    color_positions = np.arange(x_start, x_end, 1)
    cap = self.network.weight_cap
    color_bar = np.linspace(-cap, cap, len(color_positions)-1 )
    for i in range(len(color_bar)):
      weight = color_bar[i]
```

```
      color = [(cap+weight)/(2*cap)*x for x in (255, 80, 0)]
      thickness = 2*weight/cap + 3
      start_point = (color_positions[i], y_offset)
      end_point = (color_positions[i+1], y_offset)
      pygame.draw.line(display, color, start_point, end_point, int(
          thickness))
      if i==0 or (i+1)% 50 == 0 or i==398:
        text = '%.1f' % weight
        text_surface = self.font.render(text, False, BLACK)
        display.blit(text_surface, start_point)
    pass


class NeuronGroupDescription:

  def __init__(self, network, values_index):
    self.network = network
    self.idx = values_index
    self.y_value = None
    self.x_values = None

  @property
  def values(self):
    return self.network.activations[self.idx]

  def draw(self, display, font):
    color = BLACK
    radius = RADIUS
    thickness = THICKNESS
    values = self.values
    for i in range(len(values)):
      position = (int(self.x_values[i]), int(self.y_value))
      pygame.draw.circle(display, WHITE, position, radius, 0)
      pygame.draw.circle(display, color, position, radius, thickness)
      text = '%.1f' % values[i]
      text_surface = font.render(text, False, BLACK)
      rect = text_surface.get_rect(center=position)
      display.blit(text_surface, rect)


class ConnectionDescription:

  def __init__(self, network, layer_index):
    self.network = network
    self.idx = layer_index
    self.input_neuron_description = None
    self.output_neuron_description = None
    self.cap = self.network.weight_cap

  @property
  def connections(self):
    return self.network.layers[self.idx].weights

```

```
  def draw(self, display, font):
    for i in range(self.connections.shape[0]):
      for j in range(self.connections.shape[1]):
        in_index = j
        out_index = i
        weight = self.connections[i][j]
        start_point = (self.input_neuron_description.x_values[in_index],
                       self.input_neuron_description.y_value)
        end_point = (self.output_neuron_description.x_values[out_index],
                     self.output_neuron_description.y_value)
        color = [(self.cap+weight)/(2*self.cap)*x for x in (255, 80, 0)]
        thick = 2*weight/self.cap + 3
        pygame.draw.line(display, color, start_point, end_point, int(
           thick))
```

-2 Python Code for the cartpole implementation

-2-1 Cartpole.py

```
# -*- coding: utf-8 -*-
"""
Created on Sat Nov 24 13:20:24 2018

@author: ajdin
"""

import numpy as np
import gym
import time
import functions
import agent_functions as funs

from net3 import NeuralNetwork3 as nn3
from collections import deque

# make the environment
cartpole = gym.make('CartPole-v0')

# agent settings
max_episodes = 100
network = nn3(inputcount=4, hiddencount=7, outputcount=1)
N_size = 1
reward_history = deque(maxlen=N_size)
R = 0
old_R = 0
Reward = lambda queue : np.mean(queue)
p_reward = lambda state : -np.linalg.norm(np.multiply(state, np.array
    ([0.1, 0.1, 1.3, 0.1])))
gamma = 0.9
raw_rewards = []

# options
```

```
rendering = False

# data sampling
s1 = []
s2 = []
s3 = []
s4 = []




# discount rewards
def discount_rewards(r, gamma=0.9):
  # Init discount reward matrix
  discounted_reward= np.zeros_like(r)
  # Running_add: store sum of reward
  running_add = 0
  # Foreach rewards
  for t in reversed(range(0, len(r))):
      running_add = running_add * gamma + r[t] # sum * y (gamma) + reward
      discounted_reward[t] = running_add
  return discounted_reward

#linearly varied weight adjustment
def lvwa(learning_rate, reward):
  return learning_rate*(1-reward)

last_num_steps = None
for episode in range(1, max_episodes+1):
  old_R = None
  modulations = []
  R = 0
  state = cartpole.reset()
  network.modulation_signal.modulations[0] = 0.0
  network.layers[1].history_buffer.clear()
  network.layers[1].history_size = 200
  features = []
  outputs = []
  for n in range(200):
    if rendering:
      start = time.time()
      cartpole.render()
    preprocess_state = np.multiply(state, np.array([1, 1/4, 2, 1/6])) +
       np.array([0.5, 0.5, 0.25, 0.5])
    reward_history.append(p_reward(state))
    modulation = 0.0

    if len(reward_history) == reward_history.maxlen:
      R = Reward(reward_history)
      if old_R is not None:
        modulation = np.tanh(2*(R - old_R))#*(1-last_num_steps/200)
        modulation = modulation*1 if modulation > 0 else modulation
      else:
```

```
        modulation = 0
      # print(modulation)
      old_R = R
#    print('modul: ', modulation)

    network.add_experience(preprocess_state)
    features.append(network.activations[1])
    outputs.append(network.activations[-1])
    modulations.append(modulation)

    state, reward, flag, info = cartpole.step(int(round(network.activations[-1][0])))
    network.modulation_signal.modulations[0] = modulation
    network.train()
    # print(np.mean(network.layers[1].weights))

    network.layers[1].weights += funs.train(modulations, features, outputs, 1.0)
    modulations.clear()
    features.clear()
    outputs.clear()



    if rendering:
      time.sleep(max(1./40 - (time.time() - start), 0))
    if flag:
      last_num_steps = (n+1)
      if last_num_steps is not None:
        mean_mod = 1 - last_num_steps/200
      else:
        mean_mod = 0.0

      reward = last_num_steps/200
      # mean_mod = np.mean(modulations)
      modulations = [mean_mod for x in modulations]

#      modulation = lvwa(1.0, reward)
#      network.modulation_signal.modulations[0] = modulation
#      network.layers[1].history_size = last_num_steps

#      particular_reward = (n+1)  - old_R
#      raw_rewards = [0.02*particular_reward]*(n + 1)
#      old_R = n+1
#      discounted = discount_rewards(raw_rewards, gamma=0.99)
#      network.modulation_signal.modulations[0] = discounted

#      print('episode: ', ' steps: ', n+1, ' history: ', network.layers[1].history_buffer)
#      print('final training: ', episode ,'\n\n')
#      print('num samples: ', len(network.layers[1].history_buffer), discounted)

```

```
      print('episode: ', episode, '. Steps completed: ', n+1, ' avg error: ', '%.4f' % network.layers[0].average_error,
            ' mean mod: %.3f' %mean_mod, 'max mod: %.2f' %0.0)
#      print('\n', ' discounted: ', discounted)
#      print(network.layers[1].history_buffer, '\n\n')

#      network.train()
      reward_history.clear()
      break
#  network.train()

cartpole.close()


#%% render the learned policy


cartpole = gym.make('CartPole-v0')
state = cartpole.reset()
for i in range(200):
  start = time.time()
  cartpole.render()
  preprocess_state = np.multiply(state, np.array([1, 1/4, 2, 1/6])) + np.array([0.5, 0.5, 0.25, 0.5])
  network.add_experience(preprocess_state)
  state, reward, done, info = cartpole.step(int(round(network.activations[-1][0])))
  time.sleep(max(1./40 - (time.time() - start), 0))
  if done:
    break
print('done after ', i+1, ' steps.')
cartpole.close()
```