

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS
DEPARTMENT OF APPLIED INFORMATICS

**A COMPUTATIONAL MODEL OF INTRINSIC AND EXTRINSIC
MOTIVATION FOR DECISION MAKING AND ACTION-
SELECTION**



COMENIUS UNIVERSITY, BRATISLAVA
FACULTY OF MATHEMATICS, PHYSICS AND INFORMATICS

**A COMPUTATIONAL MODEL OF INTRINSIC AND EXTRINSIC
MOTIVATION FOR DECISION MAKING AND ACTION-
SELECTION**

DIPLOMA THESIS

Study programme: Cognitive Science

Field of Study: 9211 Cognitive Science

Supervisor: RNDr. Martin Takáč, PhD.

2015

Bc. Igor Lacík



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Igor Lacík
Študijný program: kognitívna veda (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: 9.2.11. kognitívna veda
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: A computational model of intrinsic and extrinsic motivation for decision making and action-selection
Výpočtový model vplyvu intrinsickej a extrinsickej motivácie na rozhodovanie a výber akcii

Cieľ: Navrhnuť a implementovať výpočtový model dieťaťa, ktoré vykonáva akcie v simulovanom prostredí na základe intrinsickej a extrinsickej motivácie. Preskúmať vplyv motivácie na rozhodovanie v rôznych scenároch. Poskytnúť webovskú aplikáciu umožňujúcu vytváranie scenárov, nastavovanie parametrov a spúšťanie a zaznamenávanie simulácií.

Literatúra: Pileckyte, I., Takáč, M.: Computational model of intrinsic and extrinsic motivation for decision making and action-selection. Technical report TR-2013-038. Faculty of Mathematics, Physics, and Informatics Comenius University, Bratislava. 2013.
Oudeyer P., Kaplan F., Hafner V. (2007). Intrinsic Motivation Systems for Autonomous Mental Development. IEEE Transactions on evolutionary computation, Vol. 11, No. 2
Oudeyer P., Kaplan F. (2007). What is intrinsic motivation? A typology of computational approaches. Frontiers in Neurorobotics 1:6

Anotácia: Hlavným hnacím motorom agentovho správania je udržiavanie homeostázy - hodnoty potrieb blízko bodu nula. Model bude založený na konekcionistickej verzii algoritmu aktér-kritik učenia posilňovaním, pričom parameter exploraácie bude previazaný s intrinsickou motiváciou (potrebou zahaňania nudy).

Vedúci: RNDr. Martin Takáč, PhD.
Katedra: FMFLKAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, PhD.
Dátum zadania: 27.11.2013

Dátum schválenia: 09.04.2015

prof. RNDr. Pavol Zlatoš, PhD.
garant študijného programu

.....
študent

.....
vedúci práce



Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname: Bc. Igor Lacik
Study programme: Cognitive Science (Single degree study, master II. deg., full time form)
Field of Study: 9.2.11. Cognitive Science
Type of Thesis: Diploma Thesis
Language of Thesis: English
Secondary language: Slovak

Title: A computational model of intrinsic and extrinsic motivation for decision making and action-selection

Aim: Design and implement a computational model of a developing child that performs a set of actions dependent on intrinsic and extrinsic motivation variables. Explore impacts of intrinsic and extrinsic motivation on decision making in various scenarios. Provide a web-based application that enables full parameter and scenario customization.

Literature: Pileckyte, I., Takáč, M.: Computational model of intrinsic and extrinsic motivation for decision making and action-selection. Technical report TR-2013-038. Faculty of Mathematics, Physics, and Informatics Comenius University, Bratislava. 2013.
Oudeyer P., Kaplan F., Hafner V. (2007). Intrinsic Motivation Systems for Autonomous Mental Development. IEEE Transactions on evolutionary computation, Vol. 11, No. 2
Oudeyer P., Kaplan F. (2007). What is intrinsic motivation? A typology of computational approaches. Frontiers in Neurobotics 1:6

Annotation: The main drive of the agent's behaviour is maintaining homeostatis - keeping the vector of values of needs close to zero. The model will utilize a connectionist implementation of actor-critic RL architecture, where exploration parameter will be dependent on intrinsic motivational variable (boredom).

Supervisor: RNDr. Martin Takáč, PhD.
Department: FMFI.KAI - Department of Applied Informatics
Head of department: prof. Ing. Igor Farkaš, PhD.
Assigned: 27.11.2013
Approved: 09.04.2015
prof. RNDr. Pavol Zlatoš, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

DECLARATION

I hereby certify that this diploma thesis is entirely the result of my own work and I have faithfully and properly cited all sources used in the thesis.

Date:

Signature:

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor RNDr. Martin Takáč PhD. for endless guidance during both implementation and theoretical stages of the thesis. Furthermore I would like to thank Indre Pileckyte for proposing the model this diploma thesis is based on. Also, I would like to thank my friend Ján Tóth for engaging in countless dialogs that helped me gain a deeper understanding of the project.

ABSTRAKT

Plačúce dieťa môže byť zavše utíšené len materským objatím. Inokedy je treba ho nakrmiť. Ak bude znova hladné, alebo v úzkosti, bude plakať znovu. Tieto akcie sú naučené a následne vyberané vďaka tomu, že dieťa je motivované vykonávať rôzne akcie v rôznych scenároch. V posledných rokoch sa intrinsitná a extrinsitná motivácia skúma za zámerom budovania autonómnych systémov a skúmania rozhodovania v organizmoch.

V tejto práci opisujeme implementáciu navrhovaného modelu rozhodovania a výberu akcií na základe intrinsitnej a extrinsitnej motivácie. Opisujeme abstracie motivačných premenných a architektúr umelých neurónových sietí tak, aby sme umožnili programátorom a kognitívnym vedcom vychádzať z našej práce pri budovaní iných modelov. Zároveň ponúkame aplikáciu, pomocou ktorej je možné špecifikovať množstvo scenárov pre navrhovaný model, čo umožní kognitívnym vedcom študovať motiváciu integrovanú do nášho modelu.

Kľúčové slová: intrinsitná a extrinsitná motivácia, učenie posilňovaním, neurónové siete, strojové učenie, simulácie

ABSTRACT

Sometimes a child might cry and can only be calmed by a mothers hug. Other times it has to be fed. When hungry or in distress, it will cry again. These actions have been selected and learnt thanks to the child being driven to perform different actions in various situations. In recent years, intrinsic and extrinsic motivation is being explored in order to build autonomous systems and research decision making in organisms.

In this thesis we describe the implementation of a proposed model of decision making and action-selection based on intrinsic and extrinsic motivation. We delineate the abstractions of used motivational variables and neural network architectures enabling computer scientists to replicate our work on different models and at the same time offer an application that serves for preparation of test scenarios in order to assist cognitive science researchers to study motivation.

Key words: intrinsic & extrinsic motivation, reinforcement learning, neural networks, machine learning, simulation

FOREWORD

This diploma thesis is written as part of completion of a master study programme Cognitive science offered by the Department of Applied Informatics in the Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava.

In this diploma thesis, students and researchers of cognitive sciences and artificial intelligence might find relevant information regarding reinforcement learning and neural network architectures and reuse our work by using attached source codes.

The thesis was part of a project that is worked on by a small international team consisting of RNDr. Martin Takáč PhD., who is in charge of the project and provides expertise regarding machine learning and cognitive modelling, Indre Pileckyte who proposed the model and will base psychology research on the implemented simulator and Igor Lacík responsible for implementing the model and the simulator.

Table of Contents

Table of Contents.....	10
1 Motivation.....	13
1.1 What is motivation?	13
1.2. Extrinsic motivation	14
1.3 Intrinsic motivation	15
1.4 Motivation in autonomous agents.....	16
1.5 Chapter summary	20
2 Proposed model.....	21
2.1 General architecture.....	21
2.2 Internal state system.....	22
2.2.1 Extrinsically motivating needs	24
2.2.2 Intrinsically motivating needs.....	25
2.2.3 Actions	25
2.3 Motivation	26
2.4 Learning & Action-selection.....	27
2.4.1 Multilayer perceptron	27
2.4.2 Actor Critic algorithm	31
2.4.3 Softmax action selection with Boltzmann distribution	36
2.5 Chapter summary	37
3 Implementation	38
3.1 Application overview.....	38
3.2 Environment overview.....	40
3.3 Agent overview.....	41

3.4 A JavaScript implementation of a Multilayer perceptron	45
3.4.1 Forward pass	46
3.4.2 Backpropagation algorithm	47
3.4.3 Using our MLP Library	48
3.5 A JavaScript implementation of Actor Critic	49
3.5.1 Act	50
3.5.2 Choosing action	51
3.5.3 Reward and Train	52
3.6 Chapter summary	53
4 Simulation & Results	55
4.1 AI Parameter Discussion	55
4.1.1 Learning rate	56
4.1.2 Number of neurons in a hidden layer	57
4.2 Scenario experiments	58
4.2.1 Extrinsic motivation with one active need	58
4.2.2 Extrinsic motivation – two dependent motives	60
4.2.3 One extrinsic motive with moving without curiosity	62
4.2.4 Two extrinsic motives with moving without curiosity	63
4.2.5 Two extrinsic motives with moving with curiosity	65
4.2.6 Combination of four motivating needs	68
4.2.7 Action with vastly different outcomes based on intensity	70
5 Summary & Discussion	74
References	76
Appendix – User Manual	80

Create a new agent entity	81
Give actions to the agent	82
Set agent as currently active.....	83
Create a new scenario.....	83
Add objects to a scenario	85
Create a new action	86
Create a new object.....	87
Run agent in the simulator	88
Save simulation logs	90
Browse agent logs	90

1 Motivation

In the following sections we lay down the theoretical background concerning the proposed model. First we attempt to define motivation and its' purpose in decision making and action-selection according to the works of various psychologists. We follow up with definitions and the differences between extrinsic and intrinsic motivation as described both in psychology and machine learning. We close up this chapter by analyzing examples and concepts of implementations of autonomous agents (Russel & Norvig, 1995) with integrated motivation.

1.1 What is motivation?

In response to similar stimuli, organisms tend to perform various actions. We can just think of a domestic cat with constant access to a food source. Sometimes it will approach the food source and eat, while on other occasions it will pass the resource seemingly uninterested. If the stimuli provided by the environment remained unchanged, the reason for this behavior must be grounded in some form of an internal change (Barnard, 1983). Regarding this, it has been argued that organisms perform various actions in an attempt to remain in a proximity of a homeostatic equilibrium (Friston & Klaas, 2007). This means, that in order to survive, the entity will try to select actions optimal for its current internal settings.

Interestingly enough a team of researchers studying a patient diagnosed with a rare genetic condition – a lack of amygdala – performed a series of experiments in order to test her lack of fear. A vast amount of the experiments included real-world scenarios such as exposure to snake and spiders (Feinstein, Adolphs, Damasio, & Tranel, 2011). While the findings of the experiment are no doubt of much benefit, the cost of the actions at the time of working on the project was of no direct utility for the survival of the researchers

– and perhaps also the participant. Nevertheless, their motivation for taking these actions was without doubt as high as the motivation to satisfy hunger.

Motivation can be defined as a reference to a set of needs that *activate, direct, and sustain goal-directed behavior* (Nevid, 2013). In literature, it is often associated with a definition of motives (needs) as driving forces of the selected actions (Bernard, Mills, Swenson, & Walsh, 2015) (Nevid, 2013) (Maslow, Frager, & Cox, 1970).

Both examples – the very primal one (eating) and another immensely complex (studying brain structures) – comply with the above mentioned definition of motivation. Eating is driven by a hunger motive, while motives for studying brain can be more complex – ranging from curiosity to preservation of physical integrity of other human beings. The terms intrinsic and extrinsic motivation have been established to simplify the description of various motives.

1.2. Extrinsic motivation

When first encountering the terms extrinsic and intrinsic, it is easy to understand them as if they were used to describe internal, or external motivation. In fact, this is a confusion. A cause of an action can be clearly external, or internal, while the motivation is extrinsic or intrinsic.

An example of an internal extrinsic motivation would be a cat feeding itself. An extrinsic motive for an external cause would be a cat running away from a splashing garden hose in order not to get wet (in order to maintain physical integrity). Simply put, a behavior is extrinsically motivated in order to satisfy physical needs. According to (Ryan & Deci, 2000) *extrinsic motivation is a construct that pertains whenever an activity is done in order to attain some separable outcome. Extrinsic motivation thus contrasts with intrinsic motivation, which refers doing an activity simply for the enjoyment of the activity itself, rather than its instrumental value.*

1.3 Intrinsic motivation

The definition of intrinsic motivation provided in contrast to extrinsic motivation by (Ryan & Deci, 2000): *Intrinsic motivation is defined as the doing of an activity for its inherent satisfaction rather than for some separable consequence. When intrinsically motivated, a person is moved to act for the fun or challenge entailed rather than because of external products, pressures or reward.*

It is obvious, that the behavior of animals and humans is too complex to be able to strictly describe actions as being carried out because of either intrinsic or extrinsic drives. It is possible, that the researchers from our example have been partly intrinsically motivated by their interests in brain structures and partly extrinsically motivated by the financial outcome of going to work. In their report, (Oudeyer & Kaplan, 2008a) describe 4 psychological theories that attempt to describe which features of activities make them intrinsically motivating and how these could be functionally implemented in organisms:

- Based on a **theory of drives** (Hull, 1943 in Oudeyer & Kaplan, 2008a) where drives are described as specific needs such as hunger, or pain that the organism strives to reduce, **drives for exploration** (Montgomery, 1954 in Oudeyer & Kaplan, 2008a) or **manipulation** (Harlow, 1950 in Oudeyer & Kaplan, 2008a) have been introduced. This approach was eventually criticized – in organisms there is no stress response for the lack of exploration (White, 1959 in Oudeyer & Kaplan, 2008a). Yet, we believe it still might provide inspiration for building abstract autonomous systems.
- In a **theory of cognitive dissonance** it was asserted, that organisms are motivated to reduce the incompatibilities between the internal cognitive model of the world and the actually perceived scenarios (Festinger, 1957 in Oudeyer & Kaplan, 2008a). Later criticism was

based around the observation that much of human behavior tends to increase uncertainty. Humans seem to seek a form of optimality between completely certain and uncertain situations.

- The last group of researchers mentioned in the report listed the features of a rewarding activity as effectance, personal causation, competence and self-determination, basically describing the situation of being in “**Flow**”. Flow can be described as being in a state where the time passes in a seemingly rapid pace and the memories of the activity are inherently pleasant (Csikszentmihalyi, 1991 in Outdeyer & Kaplan, 2008a).

1.4 Motivation in autonomous agents

Psychological definitions mentioned in the previous sections were a source of inspiration for AI researchers and roboticists, who started conceptualizing similar theories applicable for cognitive modelling and implement systems that could resemble life-like intelligence. An example of such an implementation would be a *praying mantis robot* named Miguel (Arkin, Ali, Weitzenfeld, & Cervantes-Perez, 1999). It internally stores extrinsic needs *hunger*, *fear* and a *mating need* and derives actions from the settings of these needs. The hunger and mating needs increase linearly with time, while the fear remains zero until a predator is observed. An implementation of Miguel’s behavior shows a simple, but suitable approach for achieving a behavior desired by developers. The downside of such an approach is that it does not provide much use for biologists or neurologists, since the implementation is not based on any neural abstractions.

Miguel is preprogrammed to satisfy its highest need – if the need satisfying action is available. For example, if the robot is in homeostasis and it observes a predator, it hides, but when it has been hidden for a long time and its hunger need value is higher than the fear value and prey has been perceived, it crawls out of the shelter and feeds even with a predator being in

a dangerously close distance (Figure 1 for code example). From the practical point of view, the main downside to this approach is, that the robot does not learn – if another need would be implemented, engineers behind the robot would have to write new behavior defining rules for the robot to satisfy the said need.

```

(1) Increment sex-drive and hunger, and set fear.
    sex-drive := sex-drive + 1;
    hunger := hunger + 2;      /* increment hunger twice as fast as sex-drive */
    if predator is detected,
        then fear := 10,000;   /* set fear at a high level */
        else fear := 0;        /* reset fear when no predator is visible */

(2) Check if mate or prey are close enough to eat.
    if mate is contacted,
        then sex-drive := 0;   /* reset sex-drive after mating */
    if prey is contacted,
        then hunger := 0;      /* reset hunger after eating */

(3) Each behavior produces a direction or Stop command, based on the input from its corresponding perceptual schema.
    (a) move-to-prey, move-to-mate, and move-to-hiding-place
        if prey/mate/hiding-place blob is in upper-right of image,
            then output Forward Right;
        if prey/mate/hiding-place blob is in middle-right or lower-right of image,
            then output Right;
        if prey/mate/hiding-place blob is in upper-left of image,
            then output Forward Left;
        if prey/mate/hiding-place blob is in middle-left or lower-left of image,
            then output Left;
        if prey/mate/hiding-place blob is in middle, upper-middle, or lower-middle of image
            then output Forward;

    (b) hide-from-predator
        if predator is detected,
            then output Stop,
            else output DONT-CARE;

(4) Choose an output from a behavior, to pass along to the robot.
    if there is an associated stimulus for the motivational variable with greatest value,
        then output direction from behavior corresponding to this variable,
    else if there is an associated stimulus for the motivational variable with second greatest value,
        then output direction from behavior corresponding to this variable,
    else if there is an associated stimulus for the motivational variable with third greatest value,
        then output direction from behavior corresponding to this variable,
    else if there is a hiding-place visible,
        then output direction from move-to-hiding-place behavior,

```

Figure 1: Code example of the praying mantis robot behavior as presented by Arkin et al. (1999)

Systems that implement a notion of *motives* do not only need to have action-selection hardcoded (e.g. as an inference engine (Russel & Norvig, 1995)). *Computational reinforcement learning (CRL)* framework (Sutton & Barto, 1998) can be used for self-learning systems. In CRL, the learner is not being told which actions are optimal, but instead, it explores which actions yield the maximal *reward* potential. Reward is specified as a numeric signal. When integrating motivation into autonomous systems, reward can act as a common currency for multiple motivations (Oudeyer & Kaplan, 2007b).

A challenge that can arise in reinforcement learning is that the learning happens as a trade-off between *exploration* and *exploitation*. In order to attain maximal reward, an agent must *exploit* the knowledge it has, while *explore* for unknown situations. In artificial environments, exploration can be viewed as an intrinsic motivational variable of the autonomous system.

In (Oudeyer & Kaplan, 2007b), researchers report a number of computational models inspired by psychological theories describing features of intrinsically motivating activities (see section 1.3), such as *uncertainty motivation* (tendency to be intrinsically attracted by novelty), *intermediate level of novelty motivation* or *learning progress motivation* (tendency to be attracted by situations of optimal incongruity between an internal model of the environment in given state and the real-world state), *maximizing competence progress* (where researchers attempt to simulate “flow”).

An example implementation of an autonomous system integrating intrinsic motivation is a *player of a multiplayer role playing game based on text environment* (Malfaz & Salichs, 2006). The implemented system is a player of a text-based dungeon game that is driven by intrinsic motivation based around emotions (sadness and happiness). The implemented agent performs actions that alter its internal variables, such as eating, drinking, taking medicine or interacting in a friendly, or unfriendly manner with other agents. These actions are learnt using Q-Learning – a standard reinforcement learning algorithm. A positive change reflected on the internal variables produces happiness, a negative change produces sadness. These emotions are used as a reward in the CRL for learning to optimize action-selection.

Intrinsic motivation is considered a key component in cognitive development (Oudeyer, Kaplan, & Hafner, 2007a). When we think of a child, playing in a room, repeatedly dropping objects or mimicking its parents when trying to communicate, all of this behavior seems to be performed out of an intrinsic drive. Such an embodiment of learning might be helpful in the pursuit

for intelligent autonomous systems. An abstraction of such learning process has been implemented (Chentanez, Barto, & Singh, 2004) as an *intrinsically motivating reinforcement learning* algorithm based on rewards being generated by unexpected events – thus rewarding the agent based when an unexpected situation arises.

The implemented agent is located in an artificial “playground” – an environment containing objects like a toy monkey, light switch, a ball, and a bell. Agent was represented by having a hand and an eye. When both the eye and the hand are located on an object, the agent is able to perform one of the suggested actions (like turning the light off, ringing the bell or kicking the ball). The agent learns various action sequences on the objects in order to gain extrinsic reward – can be imagined as a child being given a candy. When it performed an action leading to a salient environment change (e.g. first successful bell ringing), the agent also gains intrinsic reward. The results show that introducing intrinsic motivation into this scenario improves the learning capabilities of the tested agent.



Figure 2: A depiction of the simulated playground from (Chentanez, Barto, & Singh, 2004)

All these implementations show different approaches for integrating motivation into autonomous systems – one purely extrinsic with a rule based system for action selection, one representing intrinsic drives as emotions and using these for reward computation and one most resembling an attempt for a “flow” state – agent was motivated both by extrinsic reward and by intrinsic. The last two agents both introduced reinforcement learning techniques to

achieve learning and an abstraction of how learning is represented in real-world entities.

1.5 Chapter summary

In this chapter we reviewed the elemental theories behind motivation, explained the concepts of intrinsic and extrinsic motivation and explored techniques describing how motivation can be integrated in autonomous systems. Most sophisticated systems used reinforcement learning for optimizing behavior over time and intrinsic motivation for enabling exploration and introducing more complex behavior.

2 Proposed model

In this chapter we formalize a description of the architecture of the proposed model (Pileckyte & Takáč, 2013). First we describe the representation of input state space. We follow up by explaining how motivation is integrated in the proposed model and show that it matches the definitions of motivation delineated in the first chapter. We finalize this chapter by describing proposed algorithms and parameters for achieving self-optimizing behavior – namely actor critic for reinforcement learning, multilayer perceptron as a building block for artificial neural networks and softmax action selection with Boltzmann distribution in order to balance agents exploration/exploitation behavior.

2.1 General architecture

An agent is embodied in a static, partially observable environment and its internal physiological state is connected to this environment by performing actions in a continuous interaction loop. The agent interacts with the environment in a self-optimizing manner based on an internal state of the agent and a state of the environment (represented as objects surrounding the agent). Self-optimization is handled by a learning and action selection block.

Each available action is represented as a semi-Markov decision process (Russel & Norvig, 1995) and optimized in a manner described in the computational reinforcement learning framework (Sutton & Barto, 1998). After each time-step (one performed action) – the agent autonomously determines the reward from changes in its internal state system.

The internal state system is represented as a set of needs – both intrinsic and extrinsic. These needs can be represented as a multidimensional vector where each dimension represents one need. The agent (like organisms) is in a healthy, content state when the needs are in homeostasis – meaning the value of each need is close to zero. The motivation system is based on a deficit of one

or multiple needs. When a need is in deficit, the agent is driven to satiating it – when the agent performs an action that lowers the deficit of a particular need, it is rewarded, while when it performs an action that would lower the deficit of a need that is already satiated, the reward is minimal, or none. The reward system will be explained in depth in following sections.

2.2 Internal state system

As mentioned above, the internal state system is represented through the use of physiological (for purposes of this model *extrinsic*) and subjective (for purposes of this model *intrinsic*) needs – motives, motivational variables. The needs change in time in response to outcomes of performed actions. These needs form a multidimensional vector that represents a point in the need space. Distance of this point in the need space from the homeostatic equilibrium describes a need deprivation of the agent. The point in the need space can be positioned in various predefined regions (see Figure 4 for an illustration):

- a) *Optimal region* – A need is close to equilibrium point (point where all the needs are set to zero). In this region the need is satisfied, thus performing an action that would otherwise satiate the need only awards the agent with a small reward – or none at all.
- b) *Operational region* – One or multiple needs is in a point between an optimal region and a deficit region. When performing an action that satisfies this need, the agent is rewarded normally.
- c) *Deficit region* – When a need value is positioned very far from the equilibrium, satiating the need is most rewarding (see figure 3 for illustration) – simulating a stressful behavior (White 1957 in Outdeyer & Kaplan, 2008). This region affects only physiological (extrinsic) needs.

A depiction of an internal state space represented by an agent with 2 needs (hunger and tiredness) with regions for dynamic rewarding. In this example

the agent is well rested, but extremely hungry and will be highly rewarded for feeding (if it is already learnt, it will be highly motivated to feed).

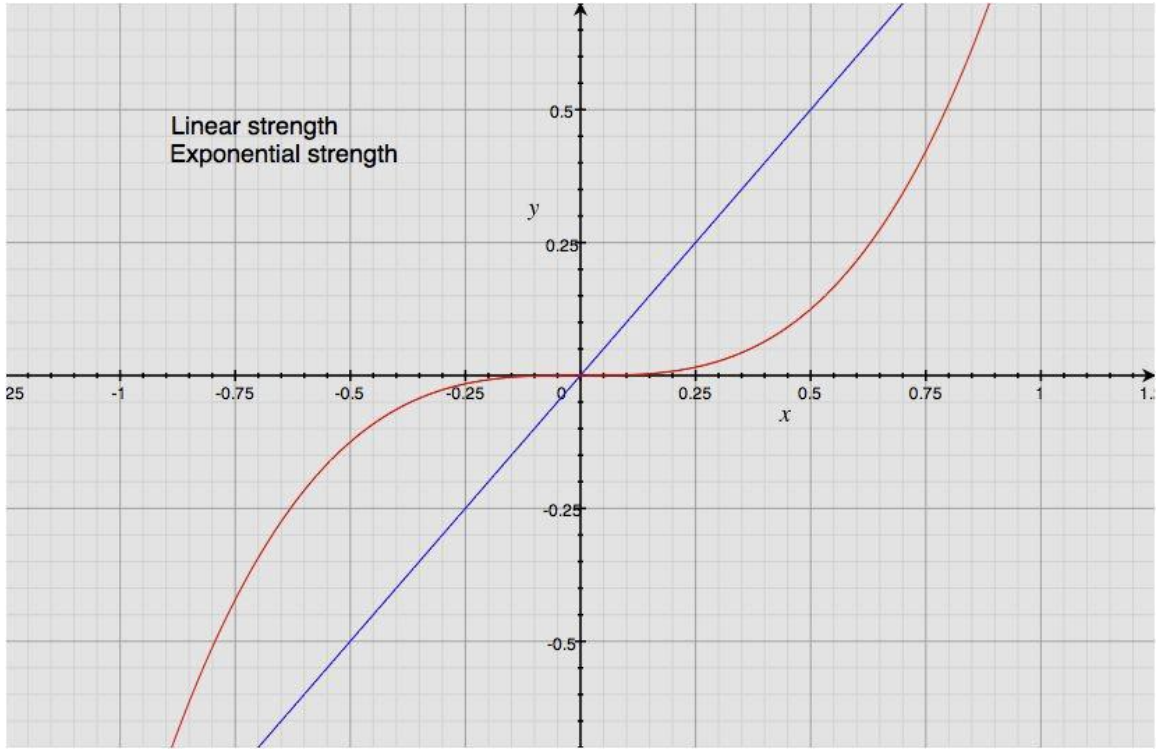


Figure 3: After trespassing deficit region threshold for a need, the motivation to satiate the need starts to grow exponentially (Pileckyte & Takáč, 2013).

Other than in the deficit region, there is no direct distinction between intrinsic and extrinsic need values, rather in the modelled behavior. When the point in internal state space is moved closer to the equilibrium, the agent is rewarded. The needs system resembles the theory of drives (Hull, 1943 in Outdeyer & Kaplan, 2008) with added intrinsic needs – in the model there is reward for experiencing novel situations but if intrinsic needs reach what would be a deficit region, no stress behavior is being simulated (White 1957 in Outdeyer & Kaplan, 2008). In the following sub-sections we will describe the **extrinsically** and **intrinsically** motivating needs and **actions** available to the agent in this model.

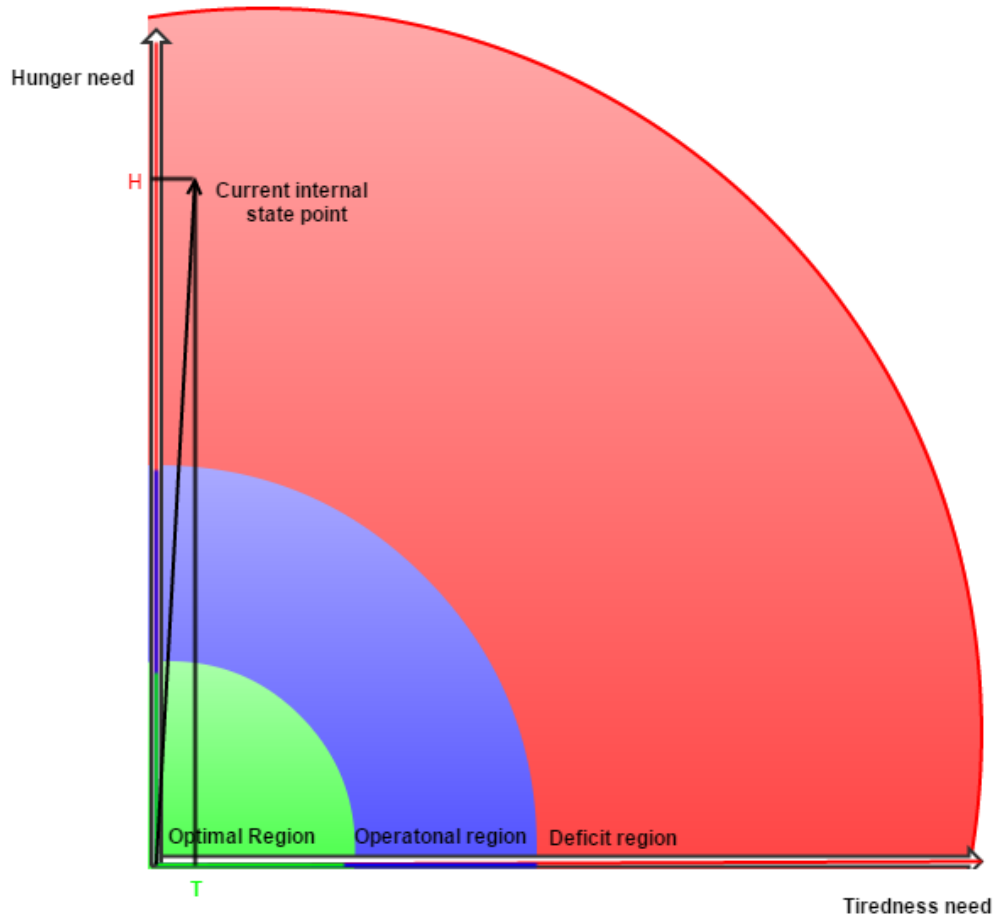


Figure 4: A depiction of an internal state space represented by an agent with 2 needs (hunger and tiredness) with regions for dynamic rewarding. In this example the agent is well rested, but extremely hungry and will be highly rewarded for feeding (if it is already learnt, it will be highly motivated to eat)

2.2.1 Extrinsically motivating needs

Hunger: With each action took (other than being fed, or eating), the hunger need increases. Hunger can be satiated by being fed (upon crying). A motivation for eating/being fed should increase in time proportionally to the value of hunger.

Tiredness: Similarly to hunger, the tiredness need increases over time (with every action took other than sleeping) and the desire to sleep should increase proportionally to the value of the need. Additionally, various actions can increase the tiredness by a different value – moving is less tiresome than playing.

Pain (Physical integrity): Upon encountering a dangerous object, this need will drastically increase and can only be reduced when escaping from the scope of force of the said object.

2.2.2 Intrinsically motivating needs

Curiosity (Boredom): Represents the agent’s desire to experience novel situations. In action selection process, the need is used to lower the strength of priority weights of strongly recommended actions and increase the rates of less recommended actions causing the agent to select an untrained action in order to explore novel situations. The need increases when the agent performs actions, outcome of which it is certain about. This will be described in detail in the learning and action selection section. The agent can also be partially driven by novel situations by default – even with the curiosity need in satiated level.

Competence: This need is not directly represented in the internal state space, rather embedded in the learning system. In direct contradiction with curiosity, the agent strives to master known ways of interacting with the environment – thus causing a constant variation between **exploration** and **exploitation**. Both needs for curiosity and competence can be weighted – enabling researchers to simulate highly curious, or perfectionist behavior.

Playfulness: Similarly to extrinsically motivating needs, the need for playing increases over time – but the desire to play does not strengthen with deficit regions. The need becomes satiated when the agent is playing. The decrease of the need depends on the intensity of the playing action – agent needs to learn how to play with a toy to gain optimal intrinsic reward.

2.2.3 Actions

In the proposed model most of the actions that an agent can perform serve for satiating one intrinsic, or extrinsic need (hunger by an eat action, tiredness by a sleep action, playfulness by a play action), while increase one or several others. Some of the actions (playing, crying) feature an *intensity*

parameter – even though the action has been selected, the agent has to learn how to perform it correctly in different scenarios. Some of the actions can be performed in any scenario, other actions can only be performed on an object (e.g. eat action can only be performed on a food object) – the agent implicitly knows if it can, or cannot perform an action.

Finally, the agent has a *move* action that – other than curiosity in specific scenarios – does not satiate any need, rather provides means to explore the environment and learn to perform actions on various objects at the cost of increasing a multitude of other needs.

2.3 Motivation

In this model, motivation simulating behavior directly emerges from the predicted reward for various agent states. In a naïve agent, the priority of each action for a given state is usually sub-optimal. A self-optimizing agent continuously acts in the environment, the learning block changes weights so in a given state it will (with increasing probability) perform the most-rewarding action. As mentioned above, the action is selected based on the state of the agent in each time-step. The state of the agent consist of two sub-states:

- a) *Objective state* – agent perceives the surrounding environment (either full environment or part of it).
- b) *Subjective state* – settings of the internal state system in the current time-step.

The states are being combined and passed to the learning & action-selection module, which selects actions to be performed. The probability of an action being performed depends on past action outcomes in the current state. Thus, in this model, motivation for performing an action is determined by the presence of external stimulators (motives, motivational variables) and internal state of needs that provide a goal – being content – resulting in a behavior that passes the definition of motivation (Nevid, 2013).

2.4 Learning & Action-selection

As stated in the previous section, action-selection is built around a system that – given a particular state – weights importance of all available actions and with increasing probabilities chooses those with higher priorities. These weights are calculated using a self-optimizing system built around a *computational reinforcement learning framework* based on the *actor/critic* algorithm (Sutton & Barto, 1998).

Neural networks in the actor/critic algorithm are represented using multiple *multi-layer perceptron* instances, one of which features a *softmax layer* for learning individual action weights (Marsland, 2009). Depending on agent’s innate curiosity and current boredom state, the calculated action weights are adjusted using *Softmax action selection with Boltzmann distribution* (Goldberg, 1990) (Sutton & Barto, 1998). In the following paragraphs we give a brief explanation of these AI techniques and algorithms.

2.4.1 Multilayer perceptron

Multilayer perceptron is a building block of our behavior emulating system. It can be used for implementing customizable, high-level abstractions of animal/human neural networks. In order to understand a multilayer perceptron, we first need to understand how we a single neuron can be modelled.

In biology, very plainly speaking, a neuron is connected to other neurons through synapses. Through these synapses, information is propagated by using chemicals called neurotransmitters. Action potential of a neuron is being summed up with every firing neuron connected to this neuron – the action potential of a firing neuron is being propagated through synapses. If action potential of one neuron connected to another neuron reaches a certain value – trespassing a threshold of approximately -55 mV – information is spread to connected neurons. If the neurons it is connected to get activated, activation

spreads through synapses to all other connected neurons (with inhibition mechanisms that prevent all neurons in the brain from being activated) (Marsland, 2009). Strength of these connections is determined by the frequency of connected neurons firing together – Hebb’s rule (Sejnowski & Tesauro, 1989). Different bulks of neurons activate for different stimuli of an organism causing the organism to react in a different manner to various situations.

A mathematical model of a neuron consists of a set of *weighted inputs*, an *adder*, and an *activation function – action potential threshold* (McCulloch & Pitts, 1943). Main idea behind this model is, that each weighted input represents a synaptic connection to other neurons and the connection strength determines the power of a given input – hence weighted inputs. Once the strength of inputs is multiplied by weights, the neuron sums input values ($\sum w_i * x_i$) and either fires (1), or not (0) depending, if summed values surpassed the threshold given by the activation function.

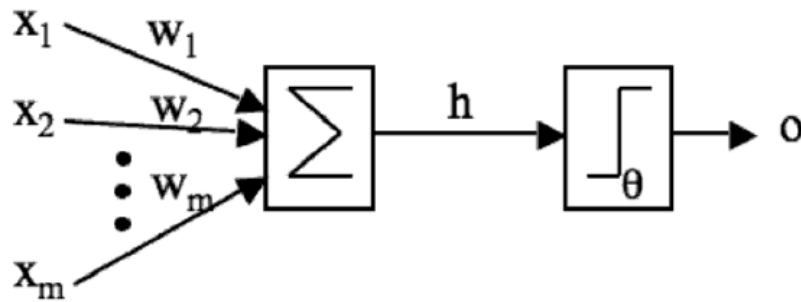


Figure 5: A depiction of a mathematical model of a neuron (McCulloch & Pitts, 1943). The inputs (x_1 - x_m) each represent one connected neuron, while the weights w_1 - w_m represent each synaptic connection for a given neuron. The stimulus are summed and compared to the threshold θ . If the threshold is surpassed, the neuron fires(1), otherwise it doesn't (0). Image from (Marsland, 2009).

Based on the mathematical model of a neuron, the simple perceptron neural network abstraction technique has been proposed and implemented (Marsland, 2009). A perceptron is an abstraction of a single-layer neural network (see Figure 6) that – using supervised learning (Fritzke, 1994) – is able to adjust weights for input neurons so the perceptron is able to categorize a set of elements or learn to predict values when given a vector of features by

propagating errors from the output layer to the weights and comparing desired output values to values that are being propagated by the neural network:

$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$

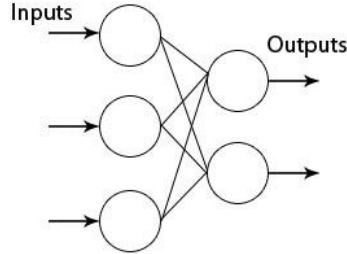


Figure 6: An example of a simple perceptron architecture with three input neurons and two output neurons in a softmax output layer. Image from (neuroph.sourceforge, 2015).

The learning capabilities of a perceptron depend on being supervised (data from the training set are presented to the algorithm until it learns them through recognizing errors in predictions) and the data-set being linearly separable (see Figure 7).

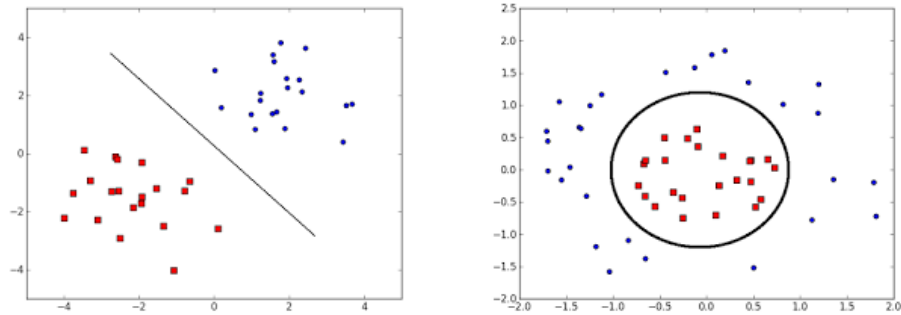


Figure 7: An example of linearly separable data – left and data that are not linearly separable (right). Image from (Sai, 2015).

These categorized elements can be instances of any objects that are described by a set of features. They can describe points in space characterized as a vector defined by x and y , but they can also describe flowers defined by their sepal length, sepal width, petal length and petal width (IRIS dataset), vehicles characterized by colors, highest speed possible or a lack of engine. For the use-case in our model, a perceptron is also obviously capable of learning to

respond to a state of an environment described in a multidimensional vector with a predicted reward value and also respond to this state of environment with a set of weighted actions.

For our model, we are likely to feature high dimension input vectors. We cannot guarantee that the input data will be linearly separable, so we will be featuring a multilayer perceptron. A difference between a simple perceptron and a multilayer perceptron lies in the number of layers. A multilayer perceptron contains one or more additional – hidden – layers of neurons – nodes that are activated nonlinearly.

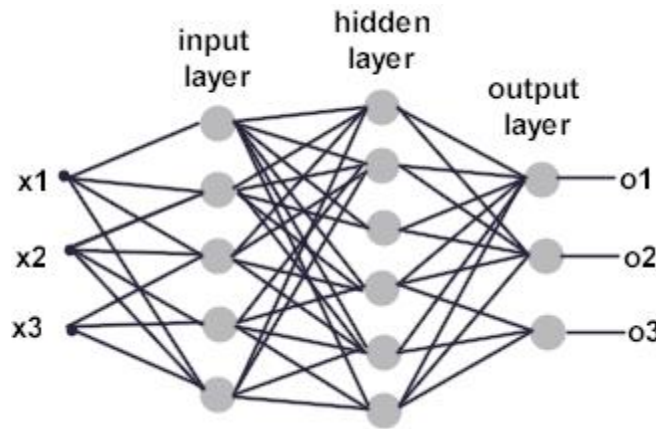


Figure 8: An example of a multilayer perceptron architecture with a softmax output layer (a softmax layer enables integration of multiple output neurons)

Each of the nodes in a layer connects through an abstraction of a synaptic weight to each node in a following layer (see Figure 8). For some time it was unclear how errors could be propagated through hidden layers (since it is impossible to recognize which weights connected to hidden layers should be trained). This problem was solved by the backpropagation algorithm (Marsland, 2009).

Learning capabilities of a multilayer perceptron are highly dependent on parameter configurations. Researchers can choose various *activation functions* (sigmoid, and hyperbolic tangent are most common), *weight initializations* (small and around point of origin), *learning rate* (constant, or

decreasing) and *number of hidden neurons in hidden layer* (deeplearning.net, 2015). There are several rules of thumb that can be used when choosing the right configuration. We will list some of these rules in the following chapters.

2.4.2 Actor Critic algorithm

Self-optimizing behavior of the agent in the proposed model is handled using the *computational reinforcement learning framework* – namely the actor-critic architecture (Sutton & Barto, 1998). Reinforcement learning is based on learning to map situations to actions that yield maximal numerical reward signal. A learner is embodied in an environment with no *a-priori* knowledge about its capabilities and environment features. The learner partially or fully perceives the surrounding environment and performs actions in a continuous interaction loop while discovering which actions receive maximum reward (see Figure 9).

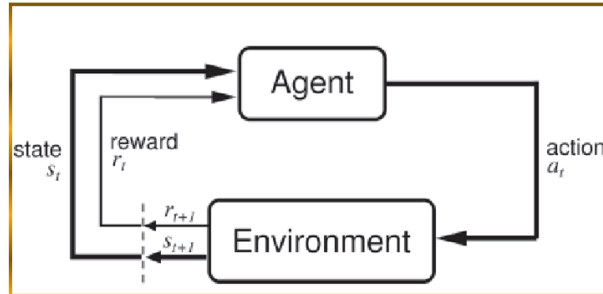


Figure 9: A depiction of the agent-environment interaction loop as specified by the computational reinforcement learning framework

As indicated in the previous chapter, unlike *supervised learning*, where the learner learns through examples provided by a knowledgeable supervisor, the agent has to *exploit* discovered knowledge to gain reward, but also *explore* unexperienced situations in order to increase its performance in time. In our model, the agent has an explicit goal of remaining in homeostasis. Integration of extrinsically motivating needs with deficit regions should drive the agent to exploit the knowledge of gaining higher reward when e.g. hungry. The integrated intrinsic motivation should kick in while in a relatively healthy state enabling the agent to explore different scenarios.

Besides the agent and the environment, implementations of a computational model of reinforcement learning involve a number of sub-elements: a *policy*, a *reward function* and a *value function*.

In psychology, a *policy* could be known as a set of stimulus-response rules. Policy represents the implementation methods applied for action-selection. In our actor critic architecture it is represented as an actor – a multilayer perceptron with softmax output layer, where each neuron in the layer represents weights of one action in a specific state.

A *reward function* defines the goal of the CRL problem. The purpose of the reward function is to map each action to a numerical reward signal – value of which is dependent on the current state of the environment. In our model, the reward is calculated as the difference of the distance of the internal state point from equilibrium in two consecutive time-steps (see Figure 10).

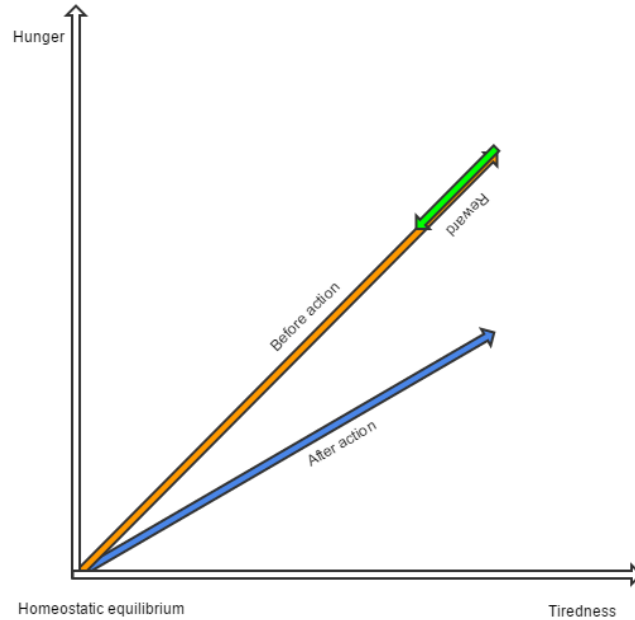


Figure 10: A depiction of how reward is calculated in our proposed model. In the previous state (S_t) the agent was hungry and performed an eat action. As the need was satisfied, the point representing agent's content in the internal space system moved closer to homeostatic equilibrium (S_{t+1}). The calculated reward $R_{(t+1)}$ is the difference of the length of vectors before action (S_t) and after action (S_{t+1}).

A *value function* defines the utility of the performed action in relation to reward in the long run. An action can have low or none immediate reward, but can lead to a potentially high future reward. A well-defined value function should provide motivation for the agent for performing move actions providing a “promise” of future reward.

Some reinforcement learning methods also integrate a *model of environment*. Models, as abstractions of the environment, are used for planning. In each step an agent is able to simulate a path consisting of its future states. Models enable more exact reward predictions and higher control over data that is being fed to the agent – but also requires pre-implementing abstractions of the world which means that intelligent behavior does not directly emerge from the agent interacting with the environment. In our model we are using a *temporal-difference (TD)* learning technique – *actor-critic* architecture.

TD learning relies on self-optimizing directly via the use of raw experience, without requiring an integration of a model of the environment. Contrary to some other reinforcement learning methods, where behavior optimization often occurs after episodes (e.g. in a tic-tac-toe game, the learning would occur after one match), TD learning methods optimize their behavior after each time-step – using a *bootstrapping* method of learning by basing future predictions on past estimates. Value $V(s_t)$ of an action in a particular time-step is computed using estimated reward $R_{(t+1)}$ of the consecutive action $V(s_{t+1})$ altered with a discount factor γ . Using reward computed from this value function (see Figure 11) the agent optimizes its decision making after each time-step.

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right].$$

Figure 11: Formula of a simple TD-learning value function (figure from (Sutton & Barto, 1998))

In our model we implement actor-critic architecture as our *TD learning technique*. As the name of the algorithm states, the main components of the architecture are an *actor* which is a name of a policy which selects actions and a *critic* – value function that criticizes actions performed by the actor. In our model, both actor and critic are represented as a multilayered perceptrons.

Both networks are fed the current state of the agent merged with the perceived state of the environment. In critic, the reward is being predicted – propagated through the network into a single output neuron.

Actor computes weights representing priorities of the actions into a softmax output layer. Actions are then being picked by softmax action selection (discussed in the next section). The performed action is criticized by the critic. The critic optimizes its weights in order to predict the reward for the current state more accurately. If the received reward was greater than the reward expected by the critic, actor modifies its weights in order to select the performed action more often (see Figure 12 for a depiction of the actor/critic architecture).

Steps of the actor-critic algorithm for our proposed model:

1. Actor propagates the priority weights for each action (in our case it propagated 80% for the top node in output layer) $A_t(S_t)$.
2. Action is selected using Softmax action selection with Boltzmann distribution a_t .
3. Critic predicts the expected cumulative reward $V_t(S_t)$ for the current state S_t .
4. The agent performs the selected action a_t – arrives in a new state $S(t+1)$.
5. Compute reward based on the performed action – reward is computed without including boredom.

6. Dependent, whether expected cumulative reward is higher than a hardcoded surprise threshold, increase/decrease boredom.
7. Actual reward with boredom inclusion is being computed $r(t+1)$.
8. Critic propagates expected reward in the new state $V_t(S_{t+1})$.
9. Target for critic is being computed: $V_{t+1}(S_t) = r_{t+1} + \gamma * V_t(S_{t+1})$.
10. Critic is being trained on an input that is a vector that represents the state of the agent and the environment before the action was performed.
11. If the target for critic is bigger than the expected cumulative reward, train actor on the same input as in step 10. – in order to perform the rewarding function more often.

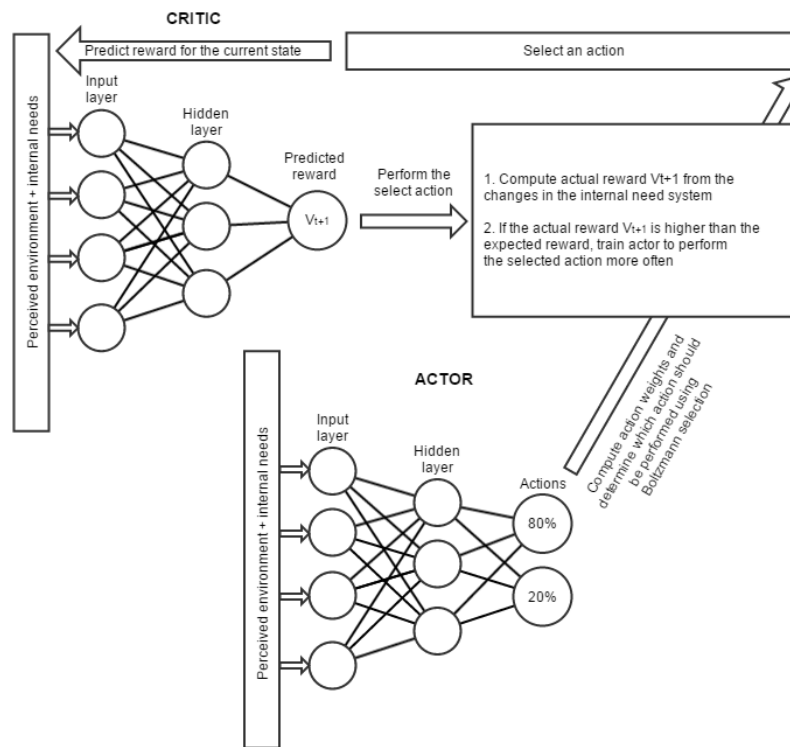


Figure 12: A Depiction of the neural network architecture of actor critic as proposed for our model

2.4.3 Softmax action selection with Boltzmann distribution

Actor learns to assign priority weights to different actions in various scenarios. If the agent is in a state of extrinsic need deprivation, it would be suitable for the agent to select an action that will satisfy the need by exploiting the knowledge of being rewarded for satiating this need when its value is in deficit region.

However, if the agent is in a homeostatic equilibrium it might be suitable for it to explore other possible actions. Similarly to a baby crawling around the house after being fed. Instead of selecting the action with the maximum potential reward, we would like to be able to dynamically decrease the priority of these actions by integrating intrinsic motivation in the process. We can achieve this by applying the boredom need as a sort of a temperature increasing mechanism. The higher the temperature, the more randomized priorities for each actions (see Figure 13 for an illustrating graph).

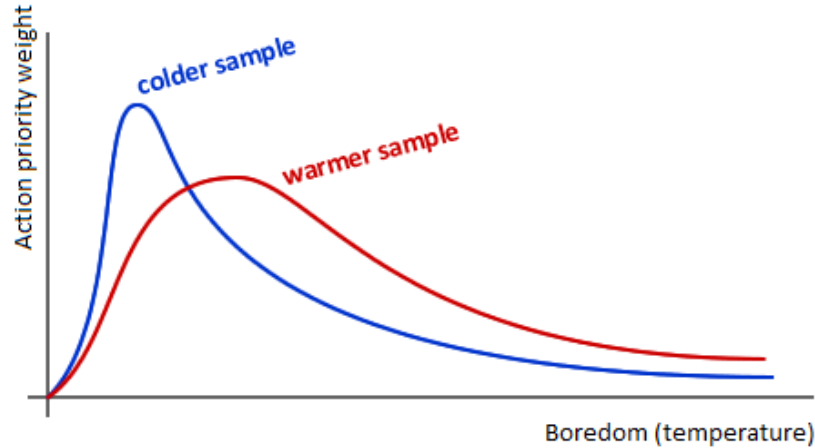


Figure 13: Increasing boredom randomizes the distribution of action priorities – image inspired by (Tarnopolsky, 2014)

In the proposed model – as mentioned in the previous paragraph – the temperature τ for altering distribution of action priorities is dependent on the value of the intrinsic boredom need. The strength of the boredom can be

customized by using a hardcoded p_τ constant. Notion of implicit curiosity can be applied by using a q_τ : $\tau = \text{boredom_need_value} * p_\tau + q_\tau$. This temperature is being applied in a Boltzmann distribution (Goldberg, 1990) (Sutton & Barto, 1998). Priority weights $P(w_a)$ of actions are being adjusted using the temperature and summed forming a distribution ($D = \sum P(w_a)^\tau$) of potentially selected actions that can be represented as a wheel resembling a roulette game (see Figure 14 for an explanation).

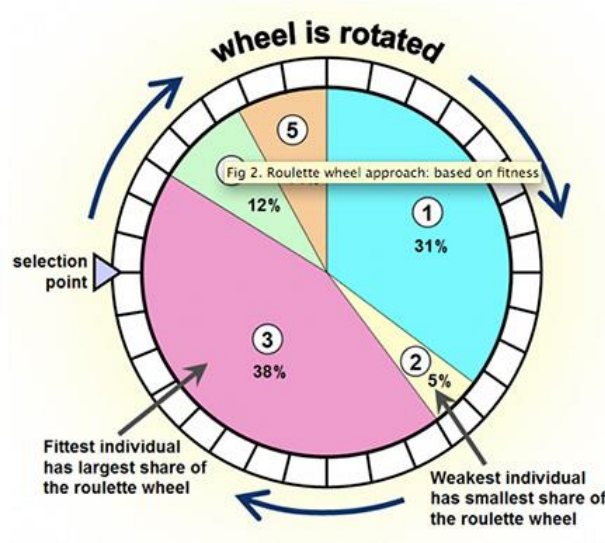


Figure 14: A delineation of how the Boltzmann distribution can be imagined. The action selection logic reminds spinning a roulette – the action that is selected to be performed is the one that owns the share in which the ball landed after spinning the roulette wheel. Image from (Reddiford, 2015)

2.5 Chapter summary

In this chapter we formalized the important components of the implemented model. First, we described proposed motivation mechanisms for both intrinsic and extrinsic needs. Later in the chapter we introduced methods used to represent the self-learning mechanisms integrated in the agent, such as multilayer perceptron, reinforcement learning (actor critic algorithm and the architecture proposed for the model), we described how reward is being computed after performing an action and perhaps most importantly we discussed using Boltzmann distribution for action selection with temperature based on intrinsic need values.

3 Implementation

In this chapter we describe the implementation details of the proposed model. First we provide a general overview of an application we built for experimenting with the model. Detailed user guide for working with the implemented application is provided in appendix. Later in the chapter we describe the environment in which the agent is embodied. Later on we also provide a description of the implementation of the skeleton of our agent and finalize with small portions of source code in order to showcase the implementation of artificial intelligence components that represent the “brain” of the implemented model. The full source code is available on our GitHub repository (Lacik, 2015).

3.1 Application overview

The application for performing experiments on the model is built as a Rich Internet Application (Piero, Rossi, & Sanchez-Figueroa, 2010) using the CakePHP framework powered by MySQL on server side for customizing agent parameters and vanilla JavaScript on client side for handling visualizations and the raw implementation of the model. Such an implementation enables team members located in different parts of the planet to observe changes and perform tests of the application without requiring users to install new versions of the application. Currently, the application resides on www.actorcritic.sk.

Similarly to product testing in commercial companies, we believe that only an application which enables detailed customization in a user-friendly manner can enable complex experimenting with projects similar to the proposed model. Our model already contains a multitude of various parameters (such as the size of perceived area, or the number of neurons in a hidden layer of a multilayer perceptron) that could be hardcoded, but it is practical for non-technical researchers, or researchers without access to the source code that they are able to update any of the parameters.

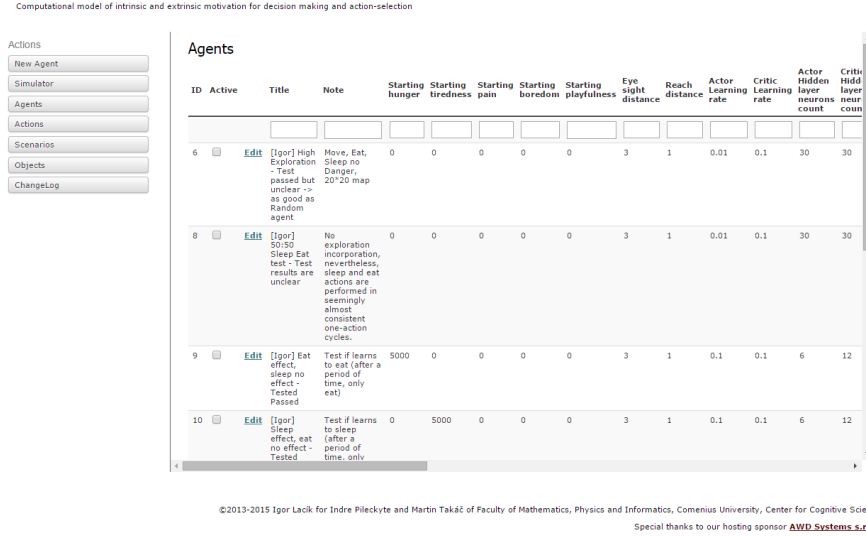


Figure 12: Example of a view of the application featuring a number of customized configurations of the agent part of the model.

The application we implemented features a number of subpages that enable researchers to manipulate parameters of various agent configurations, or agent actions and also enable building different scenarios and insert relations between these configurations (e.g. it is possible to add a relation to one action configuration from multiple agent configurations). In this way, as programmers, we can prepare environments for testing the functionality of individual model features (such as action outcomes), and also, as researchers, we can prepare environments featuring a vast number of different scenarios for testing various behavior patterns.

In summary, in the application, there is a possibility of unlimited number of various configuration entries for agents, actions and scenarios (see Figure 15 for an example). By managing relations between these entries, researchers can thoroughly test the functionality and behavior that emerges in the implemented model.

The simulation itself takes place on a subpage with an HTML5 canvas which serves for representing the configured environment. The simulation can be controlled using a set of buttons. The agents learning progress can be monitored in graphs that are being updated in real-time. Each time-step is

being saved into the web browser memory and the collection of these time-steps can be stored on the server at any moment after pausing the simulation.

In the following sections we provide a description of the environment, the agent, actions the agent can take and the implementation of artificial intelligence elements. Most of these contain parameters that determine impact of the objects on a large number of different components of the model. These parameters are fully customizable. We provide information on how these structures were intended to be used, nevertheless, researchers are encouraged to experiment with these parameters in ways we may not have predicted.

3.2 Environment overview

The agent acts on a two-dimensional spherical lattice. This means that there are no walls. If the agent is positioned on the far right on the far right position $A [length-1, 3]$, after moving right it will appear on the far left of the map $B [0, 3]$. Similar functionality applies for vertical positions and the area perceived by the agent. There are several objects that can be placed on the environment that the agent can encounter, or manipulate (see Figure 16 for a depiction of the environment):

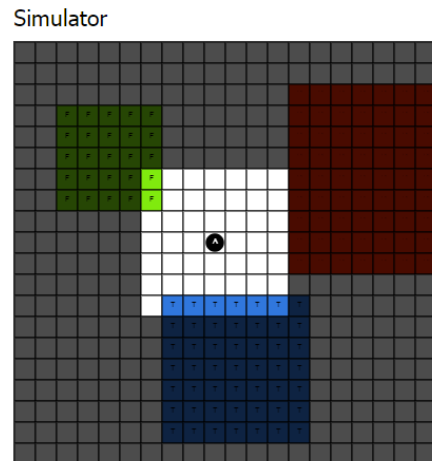


Figure 13: Depiction of the environment with agent in the center (black circle) Toys (blue fields), Food (green fields) and dangerous objects (red fields)

Fields filled with white color are *empty* and do not contain any object. Fields filled with green color with an *F* character in the center represent *food* sources. It is impossible to deplete a food source. The agent can only perform an eat action when it is near enough to the object (particular distance depends on the *reach distance* parameter configured in the agent).

Fields filled with blue color with a *T* character in the center represent a *toy* object. Similarly to the food source object, an agent can only perform a play action once it is positioned near enough to the toy object. It is impossible to move the object to a different location.

Finally, fields filled with red color with a *D* character in the center represent a *dangerous* object. When the agent moves to a field containing a danger object, its *pain* (physical integrity) need rapidly increases (the concrete value by which the pain increases each time-step can be configured in the objects subpage as explained in the attached user guide).

We can notice, that some of the objects in Figure 16 are brighter than others. This is how we illustrate which objects are being perceived by the agent and which are not.

3.3 Agent overview

Each time-step, the agent performs a set of configurable actions in the environment that are being selected by AI mechanisms in an *act* method. As stated in Chapter 2, these actions have altering effects on the agents internal need state. By default, *eating* satisfies hunger, *sleeping* satisfies tiredness, *playing* satisfies playfulness, moving increases each of these, but enables exploration of new possibilities and *crying* satisfies both hunger and rescues the agent from a dangerous situation. As stated in the previous section, some of these can be performed only when the agent is in proximity of an action-enabling object (such as a food source, or a toy).

The reward signal retrieved for performing a *play* action is dependent on the intensity with which it was performed. We can imagine this as a way of adding a requirement for learning an action in order to perform it correctly. A child starts giggling only once it manages to throw the spoon down from a table, or have it bang loudly after hitting a plate with it. If it only holds the spoon in its hand, or moves it with no sound responses it is not as content.

The implementation of a play action features the intensity parameter (intensity is being learned by a separate multilayer perceptron) and if intensity value reaches over a certain configurable threshold, the agent is rewarded with a constant reward signal. If the intensity value reaches over yet another (higher) threshold, the agent is rewarded with a randomized – yet still configurable – reward signal (see Figure 17 for a source code example).

```

111 PlayAction.prototype.affectWithIntensity = function( intensity ) {
112   if ( ( intensity > this.play_thr_const ) && ( intensity < this.play_thr_rand ) ) {
113     this.agent.playfulness -= this.const_play_dec;
114     this.title = "Play constant: " + intensity;
115   }
116   else if ( intensity > this.play_thr_rand ) {
117     this.agent.playfulness -= Math.floor( ( Math.random() * this.max_random_play_dec ) + 1 );
118     this.title = "Play random: " + intensity;
119   }
120   else {
121     this.title = "Play not enough intensity: " + intensity;
122   }
123 }; // affectWithIntensity

```

Figure 17: Source code implementation of how playing intensity affects playfulness need satiation

Similarly, intensity for crying can be described as a metaphor of how loudly the child has to scream until its mother starts being worried and tries to satisfy it. Until the child doesn't cry too loudly, the mother does not help. When the child starts to cry with a certain intensity, its mother thinks it is hungry and feeds it.

When the intensity reaches a certain relatively high threshold, the mother immediately comes thinking the child is in danger (and when it is, mother places it to a secure location). All of these actions are configurable in the *edit action* subpage as specified in the user-guide attached to this thesis. The intensity *ranges* can be configured in the *edit agent* subpage.

For understanding how the agent is embedded in the environment and how it can be configured we provide a detailed overview of parameters that can be configured in the edit agent subpage (see attached user-guide). Most of the parameters concern AI components and their impact on the emerging behavior is only outlined here (often using information from unofficial sources such as (Wikibooks, 2015) as well as official ones (Marsland, 2009)) and will be described in detail with some general rules of thumb in the next chapter:

- The *starting hunger*, *tiredness*, *pain* and *playfulness* are pretty self-explanatory – if users want to test if an agent is able to learn to eat, specifying a high *starting hunger* need can optimize the testing process.
- The *deficit regions* for *hunger*, *tiredness* and *pain* specify, when an action for satiating a need that is in deficit will multiply the reward by *deficit multiplier*, respectively for *hunger*, *tiredness* or *pain*
- *Eye sight range* enables users to specify the size of the area that surrounds the agent. The agent is able to perceive objects located in this area. Eye sight 0 means it only perceives the field it is positioned on, eye sight 1 means it is positioned in the center of a 3*3 square which it is able to perceive. The agent from Figure 16 is set with an eye sight range equal to 3.
- *Reach distance* enables users to configure the distance in which the agent is able to manipulate objects (namely perform object-dependent actions such as eating). The reach distance is not illustrated by any graphics, but the area is computed in the same manner as *eye sight range*.
- *Actor*, *Critic*, *Play* and *Cry* learning rate specifies the learning rate of a multilayer perceptron – learning rate controls the size of weight changes during training. Lower configured learning rate has a higher probability of finding local minimums of the error function, but weights are changed

in a slow manner which can cause the resulting learning process to be slower.

- *Actor, Critic, Play and Cry hidden layer count* determines the number of neurons in the hidden layer of a multilayer perceptron. Neurons in hidden layer are used to simplify the complexity of learned data (if the data are linearly separable, no hidden layer is necessary). Using additional neurons provides greater processing power and system flexibility, while adding too many neurons can cause the MLP to be incapable of generalization.
- *Actor, Critic, Play, Cry hidden layer activation function* is an abstraction representing the rate of action potential firing in a neuron. In practice, it determines the output. It is possible to configure the activation function as softmax, linear, or a hyperbolic tangent. After empirical tests we observed that it is best to use softmax in our model.
- *P Beta* and *Q Beta* in actor critic section define the temperature in Boltzmann distribution for softmax action selection from the previous chapter. Increasing *Q Beta* makes the agent become more curious even if the boredom need is set to 0.
- *Surprise threshold* with *Boredom increment* and *Boredom decrement* define how boredom need is being increased or decreased. If the surprise (difference between the predicted cumulative reward and actual reward) is higher than threshold, boredom is being decreased by the value specified in the *Boredom increment* parameter. Otherwise it is increased by the value specified in the *Boredom decrement* parameter.
- *Discount factor* is used for preparing the target for critic – also taking into account future predictions in TD learning.
- *Play threshold constant* and *Play threshold random* specify the thresholds for constant/random playfulness need satiation (value by which the need is being satiated is dependent on the configurations set in the actual play action)

- Similarly to the previous bullet-point, *Cry threshold eat* and *Cry threshold danger* allows the user to configure the threshold. Crying with intensity above the threshold causes the virtual “mother” to feed the agent/move it to a safe location. The actual amount by which the hunger need is being satisfied after being fed during crying is specified in the actual cry action.
- *P Beta* and *Q Beta* parameters in MLPs, which are used for learning optimal intensity (play and cry), both form a *Beta* parameter using the same formula as the temperature in Boltzmann distribution for this model. This Beta parameter provides a method we use to change the applied intensity in order to learn an optimal value (see Figure 18 for source code).

```

421 Agent.prototype.getIntensity = function( picked_action, input, p_beta, q_beta ) {
422     if ( !p_beta ) {
423         return undefined;
424     }
425     var beta = this.computeBeta( p_beta, q_beta );
426     function beta_intensity( intensity, beta ) {
427         var res = Number( intensity ) + ( beta * ( Math.random() * 2 ) - 1 );
428         if ( res < 0 ) return 0;
429         return res;
430     }
431     switch( picked_action ) {
432         case "Cry" :
433             return beta_intensity( this.cry_mlp.propagate( input ), beta );
434         case "Play" :
435             return beta_intensity( this.play_mlp.propagate( input ), beta );
436         default:
437             return undefined;
438     }
439 }; // getIntensity

```

Figure 14: Source code for computing intensity

3.4 A JavaScript implementation of a Multilayer perceptron

Our multilayer perceptron implementation is based on an implementation technique described in (Béhounek, et al., 2009). The described implementation technique enables us to write a source code for the multilayer perceptron without a predefined neural network architecture. The resulting source code we provide is in a form of a separate module and can be reused outside of the implementation of our model.

The neural network based on the above mentioned technique is based on indexing neurons and connecting them with other neurons in an iterative manner – a neuron with a higher index is only connected to neurons with lower index neurons, which is true for all feedforward neural networks. The resulting neural network (depicted on Figure 19) is constructed using a generator code that is located in the GitHub repository as a part of our library containing the MLP JavaScript implementation (Lacik, 2015).

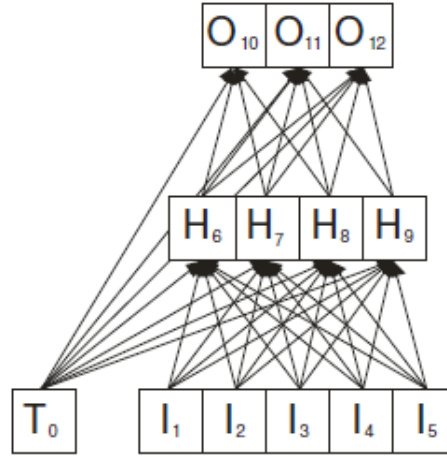


Figure 19: A depiction of a generated neural network architecture using a method with indexed neurons (Image from (Běhounek, a iní, 2009)). The image represents a feedforward neural network with 5 input neurons that serve as placeholders for input patterns fed to the network, 4 hidden neurons, 3 output neurons and one special neuron T_0 that always fires with a value equal to 1.

3.4.1 Forward pass

We illustrate the idea behind the neural network implementation using the above mentioned technique on a forward pass function (see Figure 20 for source code). Forward pass function serves for propagating errors to the output layer and retrieving trained values from the neural network. The algorithm is based on determining the types of neurons with index ui saved in $uType$ collection.

The first neuron (ui equals to 0) is a threshold neuron and always fires with an activation equal to 1. Activations of the input neurons depend on the pattern that is presently displayed to the network. Activations of the neurons

on the hidden and the output layer are calculated by summing up multiplications of source weights w_i with values on connected neurons $wValue$ $[w_i]$ and the strength of their connections $wSource$ $[w_i]$.

```

103 MLP.prototype.forwardPass = function( input )
104 {
105     var Output = [];
106     for ( var ui = 0; ui < this.NeuronCount; ui++ )
107     {
108         if ( this.uType[ui] === THRESHOLD )
109         {
110             this.ACT[ui] = 1.0;
111         }
112         else if ( this.uType[ui] === INPUT )
113         {
114             this.ACT[ui] = getInputValue( input, ui );
115         }
116         else
117         {
118             var iact = 0.0;
119             for ( var wi = this.uFirstWeight[ui]; wi <= this.uLastWeight[ui]; wi++ )
120             {
121                 iact += this.wValue[wi] * this.ACT[this.wSource[wi]];
122             }
123             this.ACT[ui] = activationFunction( iact, this.uAftype[ui] );
124             this.ACTD[ui] = derivateActivationFunction( this.ACT[ui], this.uAftype[ui] );
125         }
126         if ( this.uType[ui] === OUTPUT )
127         {
128             Output[ui] = this.ACT[ui];
129         }
130     }
131     return Output;
132 }; // forwardPass

```

Figure 20: Algorithm for forward pass

3.4.2 Backpropagation algorithm

In the backpropagation algorithm (Figure 21) we first set the derivations DE_DNA to 0. Afterwards the derivations are computed from the last output neuron to the first neuron in the hidden layer. The computation stops once it reaches input layers.

If the current neuron is an output neuron, the delta between actual and desired output is being computed into the derivation DE_DNA $[ui] += (getTarget(ui, targets) - ACT[ui])$; Since the relevant error signals have been propagated from neurons with higher indexes, the value in the current derivation DE_DNA $[ui]$ is multiplied by the a derivative of the activation function.

The above described method proved to be a useful and effective method for implementing a multilayer perceptron, nevertheless when we performed a mistake while programming the algorithm using a pseudo-code guide, it was

extremely difficult to repair our mistakes. In the next section, we provide a brief introduction to a small MLP library we built that can be used by future researchers.

```

203 MLP.prototype.train = function( input, targets )
204 {
205     for ( var ui = this.NeuronCount - 1; ui >= 0; ui-- )
206     {
207         DE_DNA[ui] = 0.0;
208     }
209     for ( var ui = this.NeuronCount - 1; ui >= 0; ui-- )
210     {
211         if ( this.uType[ui] === INPUT )
212         {
213             break;
214         }
215
216         if ( this.uType[ui] === OUTPUT )
217         {
218             DE_DNA[ui] += (this.getTarget( ui, targets ) - this.ACT[ui]);
219         }
220         DE_DNA[ui] *= this.ACTD[ui];
221         for ( var wi = this.uLastWeight[ui]; wi >= this.uFirstWeight[ui]; wi-- )
222         {
223             if ( wi === -1 )
224             {
225                 continue;
226             }
227             if ( ( this.uType[this.wSource[wi]] !== INPUT ) &&
228                 ( this.uType[this.wSource[wi]] !== THRESHOLD ) )
229             {
230                 DE_DNA[this.wSource[wi]] += this.wValue[wi] * DE_DNA[ui];
231             }
232             DLT_W[wi] += this.alpha * DE_DNA[ui] * this.ACT[this.wSource[wi]];
233         }
234     }
235     for ( var wi = 0; wi < this.WeightCount; wi++ )
236     {
237         this.wValue[wi] += DLT_W[wi];
238         DLT_W[wi] *= this.beta;
239     }
240     return result;
241 }; // backPropagate

```

Figure 21: The backpropagation algorithm based on Černansky's technique

3.4.3 Using our MLP Library

We will be glad if future researchers save time and use our implementation of the multilayer perceptron. A programmers-guide can be found in (Lacik, 2015). This small library enables programmers to integrate a multilayer perceptron only by using our *constructor*, *train* method and our *propagate* method.

The *train* method has two arguments. The first argument is an input *array*, where each *element* of the array represents an input feature for one input neuron. The second parameter is an array of targets, where each element is a target (desired value) for one output neuron in specific order (the order of

targets must be the same as the order of neurons, otherwise neurons can be trained on values that are meant to be targets for different neurons).

To *propagate* results of an input, use the *propagate* function with one input parameter – input array. Here we provide an example of how our library could be used when learning to categorize a flower from IRIS dataset:

```
mlp = new MLP (
    _inputNeuronCount,
    _hiddenNeuronCount,
    _outputNeuronCount,
    _hidden_Actvt_Function,
    _output_Actvt_Function,
    _defined_weights, // use undefined for randomized weights
    _learning_rate,
    _beta,
    is_softmax // Boolean value
); // multilayer perceptron was constructed with desired configurations

// A train method with 4 features as inputs and 3 targets - 3 categories (number
// of targets must // be the same as number of output neurons)

mlp.train ([0.224000, 0.624000, 0.067000, 0.043000], [1.0, 0.0, 0.0]);

// When we propagate the same flower, if it is trained we expect an output similar
// to [0.9, 0.03, 0.07] (meaning it is 90% sure that the flower belongs to the 1st
// category of flowers)

mlp.propagate ([0.224000, 0.624000, 0.067000, 0.043000]);
```

3.5 A JavaScript implementation of Actor Critic

In this section we describe the implementation of the logic behind the architecture consisting of multilayer perceptron instances. The individual neural networks are not connected to each other through abstractions of synapses, but through source code logic. As hinted in previous chapter, actor and critic are connected through conditions specifying whether the critic predicted a lower reward or not.

The source code itself can be divided into 3 main methods – an *Act* method contained in the agent skeleton implementation, a *Choose Action* method and a *Reward and Train* method.

3.5.1 Act

Systems implementing autonomous agents usually encapsulate the logic of AI in one method that is invoked in each time-step (in video games it can be after a number of frames). All the AI logic of the agent in our model is realized in this method. Agent first perceives the surrounding environment, prepares an input pattern for neural networks, and only then is the actor critic described in the previous chapter actually started.

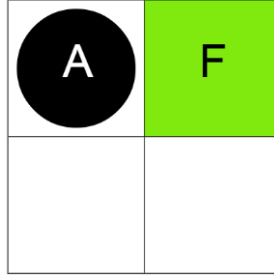


Figure 22: An image of environment fully perceived by the agent. Agent internally stores empty fields with the value 0. Fields containing a food object are represented with the value equal to 5. For the purposes of this example we can say, that the agent has a hunger value set to 25, tiredness set to 20, pain set to 0, boredom set to 0 and playfulness set to 0.

The inputs are translated into a one dimensional array containing both needs and perceived nodes. Once the perceived nodes are translated (perceived nodes from Figure 22 could form an array – e.g. [0, 5, 0, 0]), they are being concatenated with the internal need states (e.g. [25, 20, 0, 0, 0]) and form one input array such as [0, 5, 0, 0, 25, 20, 0, 0, 0]. This input will serve as input for all MLPs in the current time-step. In the constructor, all MLPs are predefined to have a number of input neurons equal to the *number of perceivable objects + 5 for the number of needs*.

Once the inputs are prepared, the functionality implemented in an actor critic object picks actions (details are explained in the following section). If the picked action requires intensity parameter, the intensity is being propagated regarding to the picked action and dependent on the level of curiosity (if cry action has been picked, intensity for crying is being computed).

When all required parameters have been computed, the agent performs the picked action. After the action has been performed, the internal state changes are computed and the agent is rewarded (with either a negative, or a positive reward signal). Finally, the computed signals are fed back to the *actor critic* object to perform training (see Figure 23 for full source code).

```

375 Agent.prototype.act = function () {
376   // AI PICK AND PERFORM ACTION //;
377   var tmp_distance = this.cur_dist;
378   var tmp_input = this.prepareInputData(); // get St
379   var picked_action = this.actor_critic.chooseAction( tmp_input );
380   picked_action = this.getActorActionByID( picked_action ); // action At
381   var p_beta = undefined;
382   var q_beta = undefined;
383   if ( picked_action == "Cry" ) {
384     p_beta = this.cry_p_beta;
385     q_beta = this.cry_q_beta;
386   }
387   if ( picked_action == "Play" ) {
388     p_beta = this.play_p_beta;
389     q_beta = this.play_q_beta;
390   }
391   }
392
393   var intensity = this.getIntensity( picked_action, tmp_input, p_beta, q_beta );
394   this.performAction( picked_action, intensity ); // perform At, is in St+1
395   // END PICK AND PERFORM ACTION //
396   this.map.drawMap();
397   this.pain += Number( this.map.getNode( this.x, this.y ).pain );
398   // TRAIN //
399   var reward_wo_boredom = tmp_distance - this.cur_dist; // compute reward without boredom
400   var reward_diff = Math.abs( reward_wo_boredom - this.actor_critic.expected_reward );
401   this.updateBoredom( reward_diff );
402   var actual_reward = tmp_distance - this.cur_dist; // Rt+1
403   this.actor_critic.computeRewardAndTrain(
404     actual_reward,
405     tmp_input,
406     this.prepareInputData(),
407     picked_action,
408     intensity
409   );
410   // END TRAIN //
411   this.finishAction();
412 }; // act

```

Figure 23: Implementation of the act method that merges all the AI components

3.5.2 Choosing action

As hinted in the previous chapter, when the actor is being fed with an array of input data, it responds with an array of probabilities. Each probability represents the probability of performing a single action. If the computed *Beta* (temperature *invTemp* for Boltzmann distribution – see Figure 24 for implementation details) is equal to 1, the probability remains the same, otherwise it is distributed – the scale of distribution depends on the curiosity level. In some situations, an untrained actor may compute a probability for actions that are not permitted in the current state (eat with no source of food in the near vicinity). Probability for performing these actions is set to 0 before the logic of Boltzmann selection kicks in.

```

189 ActorCritic.prototype.boltzmannSelection = function(
190     softmax_outputs,
191     invTemp
192 ) {
193     if ( invTemp === 0 ) {
194         invTemp = 1;
195     }
196     var powered = initArrayZeros( softmax_outputs.length );
197     softmax_outputs = this.nullUnpermittedActions( softmax_outputs );
198     var sum = 0.0;
199     for ( var i = 0; i < softmax_outputs.length; i++ ) {
200         powered[i] = Math.pow( softmax_outputs[i], invTemp );
201         sum += powered[i];
202     }
203     var p = sum * Math.random();
204     var sumPending = 0.0;
205     var i;
206     for ( i = 0; i < softmax_outputs.length; i++ ) {
207         sumPending += powered[i];
208         if ( sumPending > p ) {
209             break;
210         }
211     }
212     return i;
213 }; // boltzmannSelection

```

Figure 24: Boltzmann distribution for softmax action selection

From the source code point of view, computing an action is extremely simple (mostly thanks to the functionality provided in our MLP mini-library). First, critic propagates the expected cumulative reward (*expected_reward*). Next, the actor propagates an array of probabilities for performing each action that is being fed to the Boltzmann selection. Here the action probabilities are distributed and finally the actual action is picked (see Figure 25 for the actual implementation).

```

72 ActorCritic.prototype.chooseAction = function( input ) {
73     this.expected_reward = this.critic.propagate( input )[0];
74     this.computed_actions = this.actor.propagate( input );
75     this.picked_action = this.boltzmannSelection( this.computed_actions, this.agent.beta );
76     return this.picked_action;
77 }; // chooseAction

```

Figure 25: Source code of action picking behavior

3.5.3 Reward and Train

Once the action has been selected and performed, and the consequences of the action have been computed into the internal state system, the algorithm has to determine whether the action should be given a higher priority and correct the estimations for the expected reward in similar future scenarios.

Confirming what we described in TD learning in the previous chapter, the critic is being trained on *actual reward* as well as predicted reward of the *new state* (the state the agent has moved into after performing the selected action) multiplied by the *gamma discount factor* parameter.

It is important to note, that training in both actor and critic is performed on the input states from before the action has been performed. Actor is only trained when the difference between the expected reward and critics target is positive. Additionally, intensity is also being trained only if the respective action has been trained (see Figure 26 for implementation details).

```

99 ActorCritic.prototype.computeRewardAndTrain = function(
100   actual_reward,
101   old_input,
102   new_input,
103   picked_action,
104   intensity
105 ) {
106   this.actual_reward = actual_reward;
107   var new_state_reward = this.critic.propagate( new_input ); // Vt(St+1)
108   this.target_for_critic[0] = this.getCriticTrainingTarget(
109     this.actual_reward, // Rt+1
110     this.agent.gama,
111     new_state_reward // Vt(St+1)
112   ); // will get target Vt+1(St)
113   this.critic.train( old_input, this.target_for_critic );
114   if ( this.expected_reward < this.target_for_critic[0] ) {
115     this.actor.train( old_input, this.getActorTargets( picked_action ) );
116     if ( picked_action == "Cry" ) {
117       this.agent.cry_mlp.train( old_input, [intensity] );
118       this.cry_intensity = intensity;
119     }
120     if ( picked_action == "Play" ) {
121       this.agent.play_mlp.train( old_input, [intensity] );
122       this.play_intensity = intensity;
123     }
124   }
125 }; // compute reward

```

Figure 26: Implementation of self-optimizing capabilities of the agent

3.6 Chapter summary

In this chapter we explained how we implemented the agent, environment and AI in a configurable application. The agent is embedded in the environment through connections between a perception functionality and a learning & decision making component based on multilayer perceptron algorithm and actor critic architecture.

For the building block of our AI system we released a mini-library based on the programming technique specified in (Běhounek, et al., 2009). Our library enables full multilayer perceptron configuration while remains extremely simple to use. We demonstrated this on a short example and in the source code examples from our project.

4 Simulation & Results

In this chapter we present a number of scenarios which were used for performing experiments on the implemented model. The scenarios are presented in an iterative fashion – starting with simple scenarios and continuing with more complex ones. Surely, we do not provide a complete list of all possible scenarios, nor do we regard the descriptions of our observations as providing all the conclusions this model (or a given scenario) can offer in terms of developmental psychology. Interested researchers might be able to come up with more relevant conclusions.

During testing we not only observed, whether the agent did or did not optimize its actions, but also if it stayed in a proximity of the homeostatic equilibrium and if performed actions have been driven by motivation (e.g. if an extrinsic need would be in deficit state we would expect it to be satiated).

4.1 AI Parameter Discussion

The application we implemented (as described in the previous chapter and the manual) is meant to be used by future cognitive science or psychology researchers – not necessarily computer scientists or even people with programming skills. Even though it provides user-friendly functionality for configuring agents embedded in custom scenarios, it is useless unless the user understands even the more complex parameters. Unfortunately, this is such a wide topic that we will only provide brief guidelines and rules of thumb for these parameters.

Most of the parameters have been described in the previous chapter in a sufficient manner (for researchers who read the preliminary paper, or this diploma), but *learning rate* and the *number of neurons in a hidden layer* require further description. Following subsections serve as a guide for choosing parameter values manually, nevertheless future researchers might consider thinking about implementing a *random*, or *grid* search (Bergstra & Bengio,

2012) for optimizing hyper-parameters. This might be challenging – automated configuration could result in a loss of complex behavior.

4.1.1 Learning rate

In neural networks, a learning rate is a constant or a variable (e.g. lowering in time) determining the size of the weight changes after one training (Marsland, 2009). Choosing too low a learning rate can result in a very slow learning curve, while choosing too high a learning rate can result in suboptimal training. In some implementations, learning rate is being dynamically adjusted (e.g. in time, or after the network has been trained well enough).

A metaphor of hill traversing can be illustrated using Figure 27. The graph displays the learning progress of an algorithm. The lower the $f(x)$ the higher the ability to perform the trained task. A small learning rate can take a lot of time to get to the bottom of the graph (a global or a local minima, depending on where the starting point is), or get stuck in a suboptimal local minima, while a high learning rate might jump from one edge of the graph to another – thus never reaching optimal performance (Champandard, 2003).

During testing of our model, it seemed useful to set the learning rate for actor/critic networks to a relatively low number (ranging from 0.01 to 0.1) and the learning rate for intensity learning to higher number (ranging from 0.1 to 0.2).

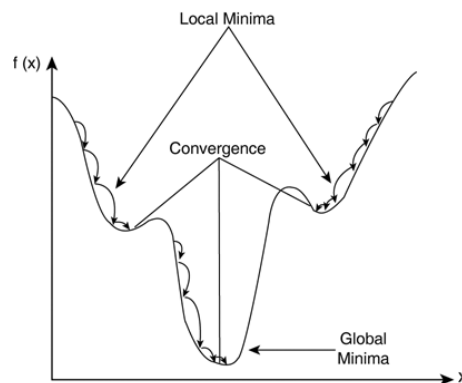


Figure 27: Illustration of local and global minima in a learning process of a neural network self-optimizing process (depiction from (Champandard, 2003))

4.1.2 Number of neurons in a hidden layer

For resolving linearly separable data, there is no need for a hidden layer. After adding a hidden layer with hidden neurons, it is still possible for the neural network to categorize the simple data. Adding hidden neurons into the neural network reduces the space complexity of the trained dataset. Determining the number of hidden neurons in a hidden layer is a difficult process that can be best illustrated on empirical examples.

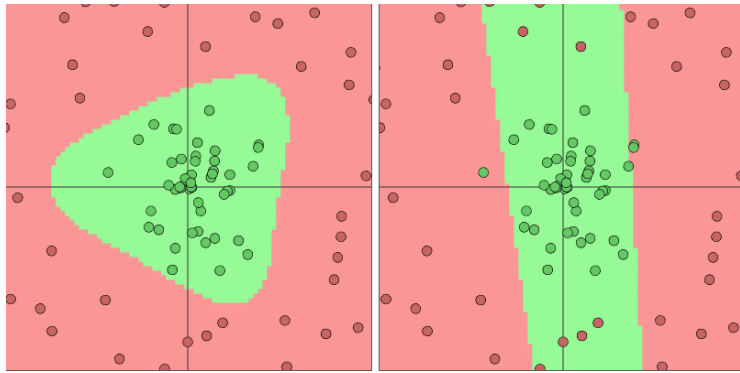


Figure 28: An illustration of a network with a good enough number – 3 – of hidden neurons (left) and one with too few – 2 – hidden neurons (right). In this task, green inputs should be categorized in the green set and the red neurons should be categorized in the red set. Illustration was performed on a playground from the ConvNetJS library (Karpathy, 2015).

The example depicted in Figure 28 illustrates the difference between a well specified number of hidden neurons in comparison to a suboptimal hidden neuron count. However, adding a large number of hidden neurons increases the complexity of the matrix computations and can result in slower learning – and performance issues of the implementation. There are some rules of thumbs we can apply even though thoughts on their effectivity vary (NN FAQ, 2015):

1. Hidden layer size is somewhere between the input layer size and the output layer size (Blum, 1992)
2. $(\text{Input size} + \text{Output size}) * 2/3$
3. Number of hidden nodes = number of principal components needed to capture 70-90% of the variance of the input data set (Boger & Guterman, 1997)

4.2 Scenario experiments

In this section we describe how the model can be tested on a number of scenarios. First we start with simple scenarios, in which we test the agent's ability to remain or converge to a proximity of a homeostatic equilibrium, while only having to care about a subset of all possible needs with a subset of all possible actions (starting with purely extrinsic motivation factors and later continuing with intrinsic motivation). In more complex scenarios we examine the behavior of an agent with various action/need combinations. For evaluating results we are using data generated by our environment in the Logs section (see manual).

For representing scenarios we will be using a similar method to the one proposed in the *Scenarios* part of (Pileckyte & Takáč, 2013). Each scenario will be described by *active types of motivation*, *admissible actions*, *parameter configuration* (we will only include the parameters relevant to the scenario), a *depiction of the environment*, *intuitive results* and *actual results*.

4.2.1 Extrinsic motivation with one active need

Active types of motivation:

Purely extrinsic motivation for satisfying hunger.

Admissible actions:

Eating satisfies hunger by 5 (until hunger < 5).

Sleeping has no impact on agents internal need system.

Parameter configuration:

Starting hunger	1000
Eating hunger increment	-5
Sleeping	No change of internal needs
Deficit region	None

Actor/Critic hidden neuron count	8/8
Actor/Critic learning rate	0.1/0.1
Discount factor	0.1

Environment:

A 2*2 map with – agent has constant access to food (Figure 29).

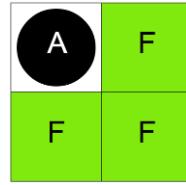


Figure 29: Environment depiction

Intuitive results:

In summary, this scenario is supposed to demonstrate the agent’s ability to learn. After trying both actions, the agent should quickly prioritize an eat action – until it converges close to a > 90% probability of performing eat action.

After learning to perform an eat action with a high probability, critic should estimate its target as relatively high (based on the settings of the discount factor). The critic should be able to learn to predict values close to the computed targets – no surprise is involved.

Actual results:

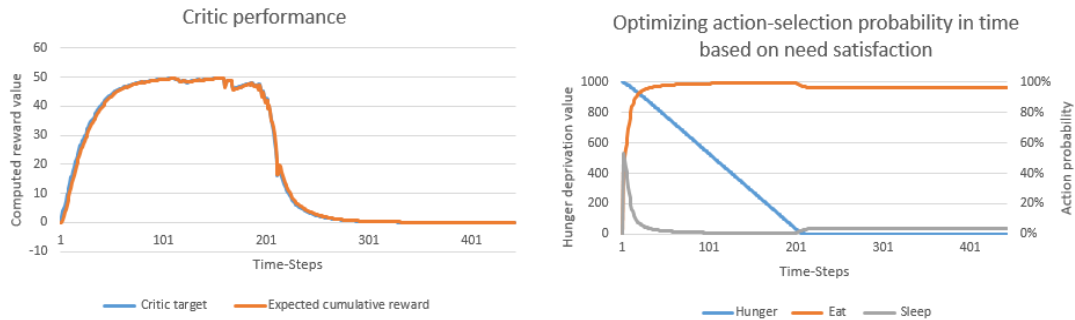


Figure 30: Graphs depicting average agent behavior over 10 runs. The plot on the left shows that the critic had no trouble learning fed targets (also showing effect of discount factor with rising targets), and after

the agent reached optimal region (around step 200) the result estimation converged to 0. The plot on the right clearly shows that the actor was able to learn to eat with a close to 100% probability very quickly – lowering the probability of sleeping. A small error in the learning curve is visible in both plots causing the agent to slightly increase sleeping after being fully satisfied.

4.2.2 Extrinsic motivation – two dependent motives

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Tiredness satiation with a less powerful drive when in deficit region.

Admissible actions:

Eating satisfies hunger by 5, but increases tiredness by 5 after being performed. Deficit region for hunger is set to 50 and reward for satisfying hunger when in deficit region is doubled.

Sleeping satisfies tiredness by 5, but increases hunger 5 after being performed. Deficit region for tiredness is 20 and reward for satisfying tiredness when in deficit region is multiplied by 1.5.

Parameter configuration:

Starting needs	0
Eating hunger increment	-5 hunger (+5 tiredness)
Sleeping	-5 tiredness (+5 hunger)
Deficit region (hunger/tiredness)	50: 2 * reward / 20: 1.5 * reward
Actor/Critic hidden neuron count	8/8
Actor/Critic learning rate	0.01/0.1
Discount factor	0.1

Environment:

A 2*2 map with – agent has constant access to food (as in Figure 29).

Intuitive results:

Agent is expected to cycle between actions eat and sleep. The need values in these cycles should be in a proximity of deficit regions – the agent should learn that the reward signal lowers near the homeostatic equilibrium.

Actual result:

The actual number of actions shows alternation between both actions. The alternation is more frequent in time (Figure 31), showing that the agent first cycled between highly preferring one action. The distribution of rewards for hunger show a higher probability of low rewards in optimal and also deficit region for hunger – which is why we can see that the trained agent remains in deficit region of tiredness while acting in operational region of hunger.



Figure 31: Number of performed actions in time (top plot), a distribution of reward values for hunger states (bottom left) and distribution of action probabilities for hunger values in a trained agent (after approx. 200 time-steps). For smoother plots, data are averaged from a time-frame of 50 time-steps.

4.2.3 One extrinsic motive with moving without curiosity

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Admissible actions:

Eating satisfies hunger by 5

Moving increases hunger by 0.5

Parameter configuration:

Starting needs	0
Eating hunger increment	-5 hunger
Moving	+0.5 hunger
Deficit region (hunger)	50: 2 * reward
Actor/Critic hidden neuron count	14/14
Actor/Critic learning rate	0.01/0.1
Discount factor	0.1

Environment:

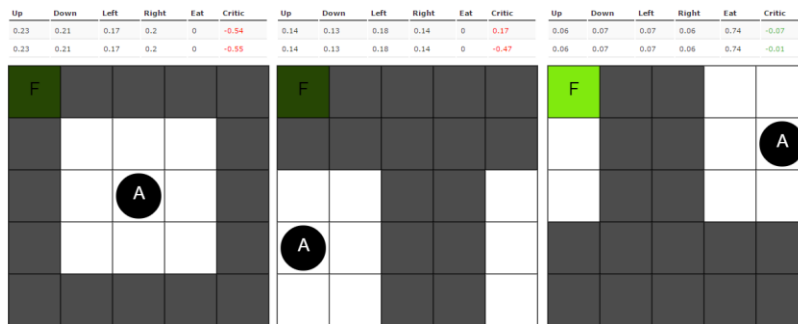


Figure 32: Environment depiction

A 5*5 map with one food source – the agent is required to move near to the food source in order to be able to perform an eat action. We may notice, how the distribution of actions is being changed in time (in the table above the map

depiction – upper row represents actor outputs before the Boltzmann distribution is applied, the bottom row represents actor outputs after the Boltzmann distribution is applied). Upper critic value represents expected cumulative reward while the bottom value represents critic target.

Intuitive results:

The agent should perform move actions until encountering the food source and performing an eat action. Each time after an eat action has been performed, the number of performed eat actions in the last 50 steps should increase close to linearly.

Actual results:

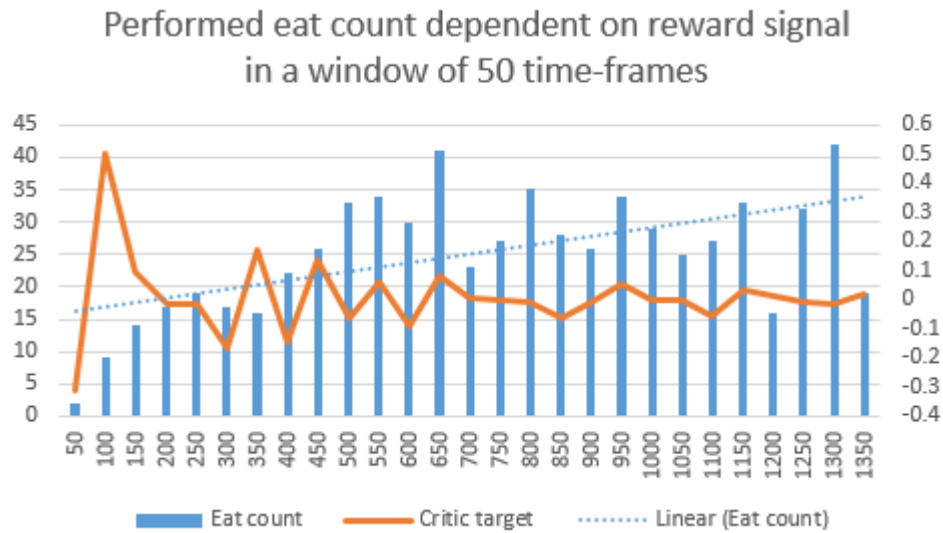


Figure 33: While the eat action is not performed too often (causing a rapid reward reduction), the number of actions increases. Once the agent remains in optimal region for a long time (reward is close to 0), the frequency of eat actions is being reduced – and after increasing hunger deprivation it is increased again. Hunger need never reaches the deficit region.

4.2.4 Two extrinsic motives with moving without curiosity

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Tiredness satiation with a drive when in deficit region.

Admissible actions:

Eating satisfies hunger by 5 and increases tiredness by 0.1.

Sleeping satisfies tiredness by 5 and increases hunger by 0.1.

Moving increases hunger by 0.5 and increases tiredness by 0.1.

Parameter configuration:

Starting needs	0
Eating hunger increment	-5 hunger (+5 tiredness)
Moving	+0.5 hunger
Deficit region (hunger/tiredness)	50: 2 * reward / 100: 1.5 * reward
Actor/Critic hidden neuron count	14/14
Actor/Critic learning rate	0.01/0.1
Discount factor	0.5

Environment:

Identical to the one described in the previous scenario.

Intuitive results:

While satisfying tiredness when in it is in operational region, the agent should perform move actions until encountering the food source and performing an eat action. After performing a series of eat actions, the agent should be able to learn to alternate between sleep and eat actions in order to remain in proximity of the homeostatic equilibrium.

Actual results:

At first, the agent seems to learn to alternate between eating, sleeping and moving, but after a relatively low number of time-steps actor is over-trained and prefers the sleep actions even when it could be highly rewarded by performing eat when hunger reaches past the borders of deficit region (see Figure 34).

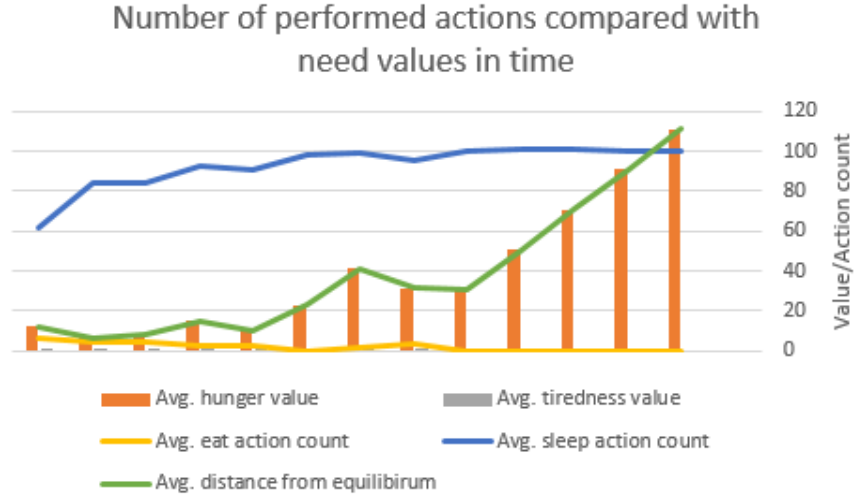


Figure 34: A depiction of agent behavior during this scenario. Horizontal axis represents the number of time-steps (2700 in total). The timeframe in which the sleep actions have been count and averaged is 100.

The observed cause for this behavior is a lack of opportunities for performing eat action – agent does not move often enough to encounter the food source. During testing, high discount factor configuration enabled a presence of initial sleep/eat action repetition, nevertheless, in the long run sleep action would dominate. From the model architecture we can predict that introducing boredom to this scenario should resolve this problem.

4.2.5 Two extrinsic motives with moving with curiosity

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Tiredness satiation with a drive when in deficit region.

Boredom after receiving no surprise from performing an action.

Admissible actions:

Eating satisfies hunger by 5 and increases tiredness by 0.1.

Sleeping satisfies tiredness by 5 and increases hunger by 0.1.

Moving increases hunger by 0.5 and increases tiredness by 0.1.

Parameter configuration:

Starting needs	0
Eating hunger increment	-5 hunger (+5 tiredness)
Moving	+0.5 hunger
Deficit region (hunger/tiredness)	50: 2 * reward / 100: 1.5 * reward
Actor/Critic hidden neuron count	14/14
Actor/Critic learning rate	0.01/0.1
Discount factor	0.1
Surprise threshold	3
Boredom increment/decrement	0.05/5
Actor PBeta/QBeta	0.05/0

Environment:

Identical to the one described in the previous scenario.

Intuitive results:

While satisfying tiredness when in it is in operational region, the agent should perform move actions until encountering the food source and performing an eat action. After performing a series of eat actions, the agent should be able to learn to alternate between sleep and eat actions in order to remain in proximity of the homeostatic equilibrium.

Actual results:

After running the program we were unsure whether the agent is learning, or acting randomly (even after qualitatively observing a dramatic decrease of hunger deprivation when the hunger value exceeded hunger deficit region). When setting the curiosity parameters we have to be careful not to set these too high – the behavior can be very random, or contrary, with high Beta,

the action probability distribution can be even smaller than before applying Boltzmann distribution formula – high priority actions remain dominant.

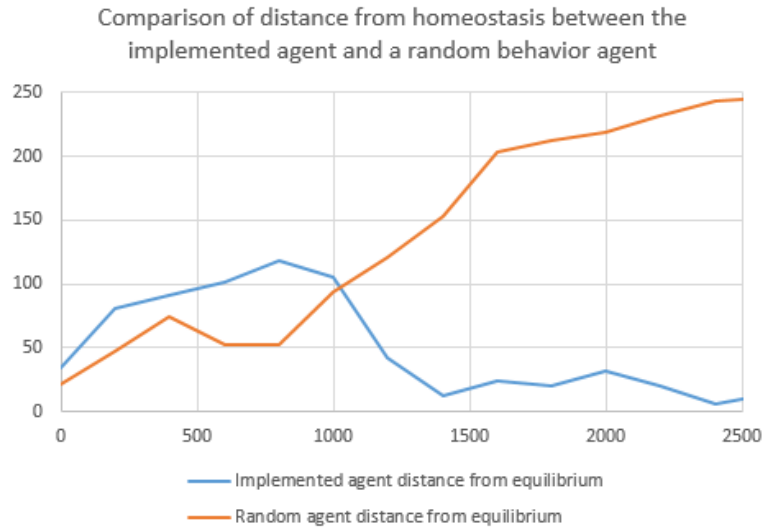


Figure 35: Performance difference between the implemented AI and random behavior agent averaged in a window of 100 time-frames.

In order to ensure, that the agent is not being successful (close to homeostasis) only by chance, we implemented a randomly behaving agent. This agent has evenly distributed chance to perform all available actions in every time-step (it cannot perform an eat action in the same states as the implemented agent). Data comparison from runs of random agent and the implemented agent show significant difference in performance of these agents (see Figure 35).

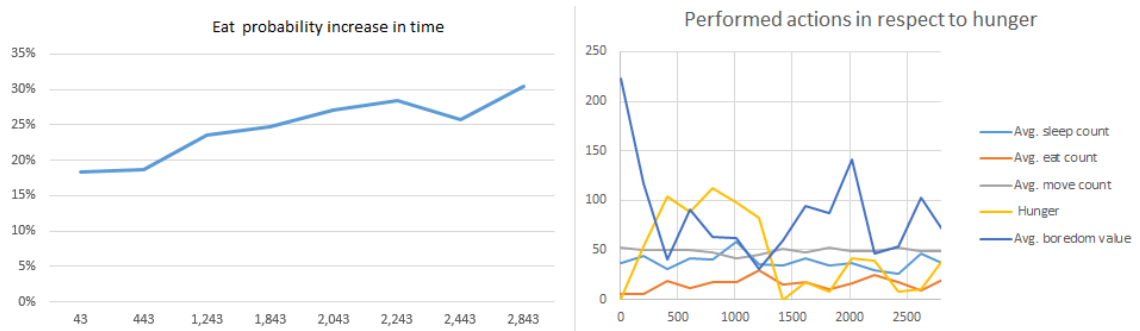


Figure 36: Action performance in time comparable with hunger values (problematic in previous scenario). At first, hunger need increased dramatically, but after the eat action has been tried in deficit region, the probability for performing the action increases and around time-step = 1400 only the eat action is being

performed. Never again does the hunger need exceed deficit region boundary – both sleep and eat actions are performed often enough to remain in operational region while curiosity drives the agent to explore when the needs are satiated (to enable need comparison, boredom value has been multiplied by 100 in each point). All data have been subsequently averaged from a window of 100 time-steps.

4.2.6 Combination of four motivating needs

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Tiredness satiation with a drive when in deficit region.

Slightly increasing drive for playfulness.

Boredom after receiving no surprise from performing an action.

Admissible actions:

Eating satisfies hunger by 5 and increases tiredness by 0.5 and playfulness by 0.1.

Sleeping satisfies tiredness by 5 and increases hunger by 0.5 and playfulness by 0.1.

Playing increases hunger and tiredness by 0.5, and decreases playfulness by intensity dependent constant value 10, or random value from range 0 to 8.

Moving increases hunger by 0.5 and tiredness by 0.5.

Parameter configuration:

Starting needs	0
Deficit region (hunger/tiredness)	50: 2 * reward / 100: 1.5 * reward
Actor/Critic hidden neuron count	14/14
Actor/Critic learning rate	0.01/0.05
Discount factor	0.1
Surprise threshold	3
Boredom increment/decrement	0.05/5

Actor PBeta/QBeta	0.05/0
Play MLP hidden neuron count	15
Play MLP learning rate	0.1
Play MLP P/Q Beta	0.1/0.05
Play threshold const/random	1/5

Environment:

Similar to last scenario. A toy object is added to the bottom left corner (Figure 37).

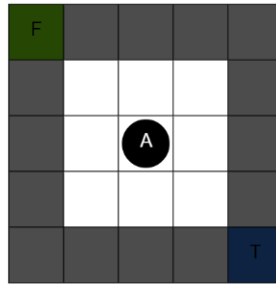


Figure 37: Environment depiction

Intuitive results:

Even by adding one new object and action it can prove increasingly difficult to predict the agent's behavior. There is a possibility that the agent will become unable to learn – requiring manual parameter adjustments – or there will be no change at all – in its attempt to remain in the proximity of homeostasis, the agent will perform well. The addition of playing – generally speaking an extrinsic action that is more difficult to learn and does not even have the motivational benefit of a deficit region – might help lower boredom, if it gets too high, or it might even be ignored.

Actual results:

Actual results are also hard to evaluate. First, in several runs boredom has risen too high causing a constant repetition of either sleep, or eat action.

Indeed, we resulted in very slight parameter changes in order to be able to observe interesting behavior. Mainly, the intensity boundaries of constant reward for play action has been set to 1 so it is easier to learn.

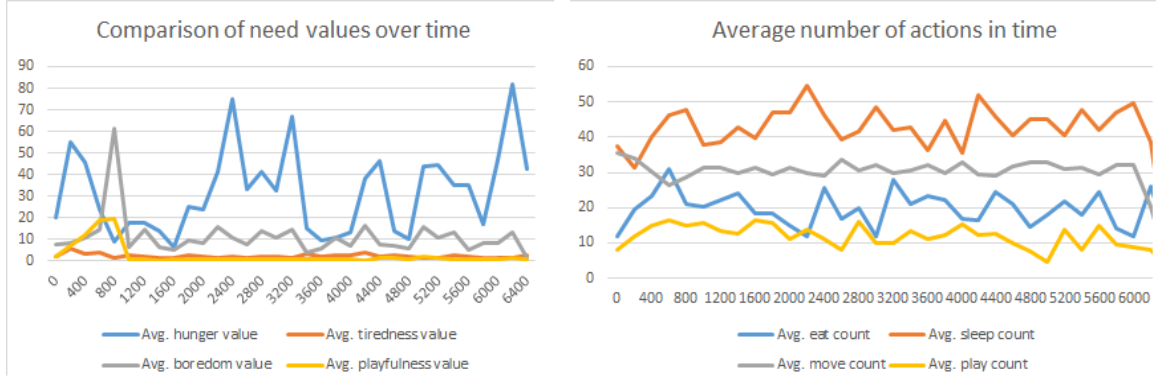


Figure 38: In the left graph spikes in boredom are followed by lowering the deficit of a currently high need (mostly hunger). Nevertheless, quite often and for longer periods, hunger is in deficit region – this is slightly different in the previous scenario, and may suggest that the play action intervenes with drives for satiating extrinsic needs. Again, we can observe a rapid increase of eat actions once the agent performs an eat action when hunger is in deficit region – complying with the theories described in chapter 1. All in all adding this many actions and needs causes complex behavior (again, for better graph clarity, boredom values have been multiplied by 100 and the data have been subsequently averaged in a time-frame window of 200 time-steps).

4.2.7 Action with vastly different outcomes based on intensity

Active types of motivation:

Hunger satiation with a strong drive when in deficit region.

Strong drive to avoid pain.

Boredom after receiving no surprise from performing an action.

Admissible actions:

Crying with intensity exceeding 4 satisfies hunger by 5 and unless on dangerous object, crying always satisfies pain by 30. Crying with intensity exceeding 7 relocates the agent on a secure node. Moving increases hunger by 0.5 and can place the agent on a dangerous – rapid pain increasing – object.

Parameter configuration:

Starting needs	0
Deficit region (hunger/pain)	50: 2 * reward / 100: 4 * reward
Actor/Critic hidden neuron count	14/14
Actor/Critic learning rate	0.01/0.05
Discount factor	0.1
Surprise threshold	3
Boredom increment/decrement	0.05/5
Actor PBeta/QBeta	0.05/0
Cry MLP hidden neuron count	15
Cry MLP learning rate	0.1
Cry MLP P/Q Beta	0.1/0.05
Cry threshold eat/run away	1/5

Environment:

One half of the map is covered by dangerous, pain inflicting objects, while the other one is empty (see Figure 39).

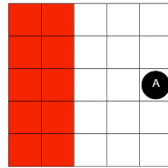


Figure 39: A depiction of the environment for the current scenario.

Intuitive results:

After repeatedly encountering the dangerous objects, the agent should be able to learn not to step on these as often. Action that would be performed most often should be crying on an empty field.

Actual results:

The agent was unable to remain constantly in a close proximity of the homeostatic equilibrium. From pure observation we were not sure, if the agent

is acting randomly, or adopts a semi-deterministic behavior, we again compared the distance from homeostasis of the implemented agent and a random behavior agent. The difference between these systems is enormous (see Figure 40) suggesting a large complexity of this scenario.

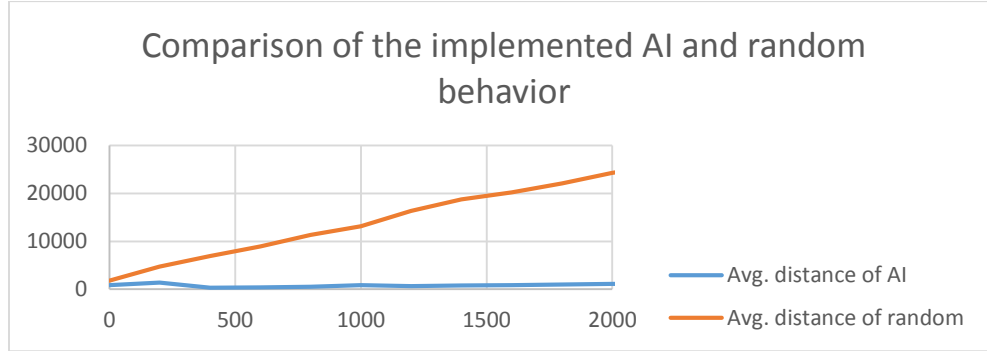


Figure 40: Comparison of the distance from homeostasis between the implemented AI (fluctuating from 0 to 1000) and a random behavior agent (linearly increasing from 0 to 30000 in around 2000 steps). Again, data have been averaged in a window of 200 time-steps.

The intensity parameter in crying took a long time to train – mostly because boredom took a long time to navigate the intensity to exceed the configured boundaries (see Figure 41). Once the agent learn how to cry for food, it was instantly driven to lower the hunger need and managed to remain near homeostasis. When the hunger has been satiated, the value the beta parameter configured in the agent drives the neural network to for different behavior – again lowering the intensity.

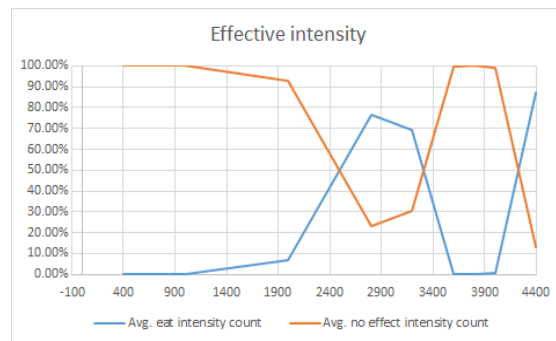


Figure 41: Percentage of effective and ineffective intensity in time. Displayed data is averaged from a 200 time-step window.

Even though the agent managed to near to a value of pain equal to 1000, in the long run it managed to satisfy the need by crying in safe locations, slowly lowering the amount of time it stepped on the dangerous object (see Figure 42 – chart on the left).

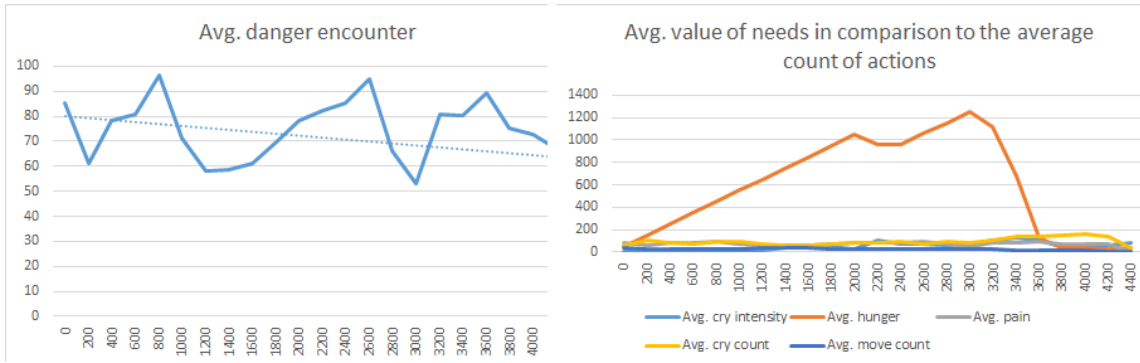


Figure 42: Average number of time the agent stepped on a dangerous objects (left) in timeframes of the size 200 and the average number of times an action has been performed in regards with extrinsic needs. Cry intensity has been multiplied by 100 in order to see results.

5 Summary & Discussion

Not a hundred years after first digital computers were built we are trying to use them to understand the development of cognitive abilities by implementing an abstraction of a child.

When working on such a project, we have to appreciate the amazing complexity of our minds. While writing the source code, understanding what it is that we are trying to implement or analyzing the data, we are using a combination of cognitive abilities that are light years ahead of what we are currently able to implement.

In our project we did not focus on the development of all cognitive abilities of a child, merely decision making. We have built an application that enables us to put a specifically configured agent into a specific scenario to observe its behavior – much like a chemist adds different substances into a test tube to observe reactions. In this chapter we showed a subset of a series of test scenarios we built for experimenting with the model.

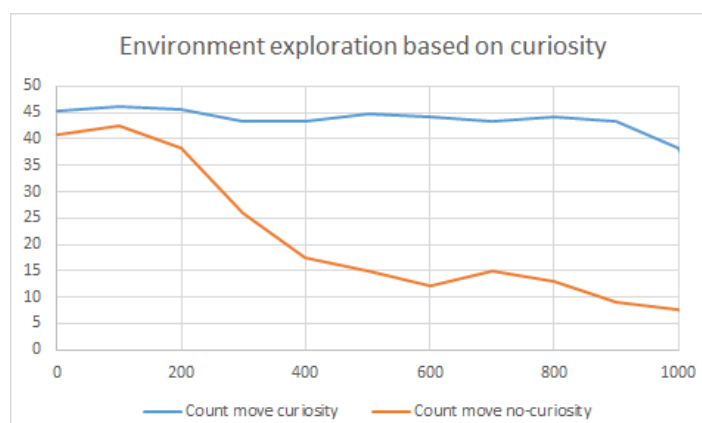


Figure 43: Even after learning to satisfy hunger, intrinsically motivated agents continue exploring the environment, while agents motivated purely by extrinsic needs mostly stay near the food source and perform an eat action. Data averaged from windows of 100 time-steps.

Despite their simplicity, on some of these scenarios we observed a relatively complex behavior. The main outcome of these scenarios was the fact that without intrinsic motivation, the agent would be unable to overcome an

overlearning problem. Intrinsic motivation in the form of curiosity enabled the agent to explore scenario-specific available possibilities until it was driven by extrinsic motivation to satisfy basic needs such as hunger, or tiredness – similarly to the theories described in Chapter 1. Our results suggest, that binding exploration with boredom as a form of intrinsic motivation could be an effective tool

Corresponding to our assumptions, in trained agents, a drive of an extrinsic need was mostly strongest in between optimal and operational regions. This is counter-intuitive, and we do not have a clear explanation for this behavior. Our assumption is that the configured agent is unable to delay gratification and wait until the reward reaches deficit region. For untrained agents, deficit region proved to optimize the learning curve of a given action and also proved well with learning to master this action (as with the intensity parameter for crying or playing in the last scenarios).

References

- Arkin, R. C., Ali, K., Weitzenfeld, A., & Cervantes-Perez, F. (1999). Behavioral models of the praying mantis as a basis for robotic behavior. *Robotic and Autonomous Systems*, 39-60.
- Barnard, C. J. (1983). *Animal Behaviour - Ecology and Evolution*. Springer US. doi:10.1007/978-1-4615-9781-0_2
- Bergstra, J., & Bengio, Y. (2012). Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13, 281-305.
- Bernard, L. C., Mills, M., Swenson, L., & Walsh, R. P. (2015). An Evolutionary Theory of Human Motivation. *Genetic, Social, and General Psychology Monographs*, 129-184.
- Blum, A. (1992). *Neural Networks in C++*. NY: Wiley.
- Boger, Z., & Guterman, H. (1997). Knowledge extraction from artificial neural network models. *IEEE Systems, Man, and Cybernetics Conference*. Orlando, FL.
- Champanhard, A. J. (2003). *AI Game Development: Synthetic Creatures with Learning and Reactive Behaviors*. Introduction to optimization: New Riders.
- Chentanez, N., Barto, A. G., & Singh, S. P. (2004). Intrinsically motivated reinforcement learning. *Advances in neural information processing*, 1281-1288.
- Čerňanský, M. (2009): Rekurentné neurónové siete. In: V. Kvasnička, J. Pospíchal, Š. Kozák, P. Návrát, & P. Paroulek, Eds. *Umelá inteligencia a kognitívna veda I*. Bratislava: STU.
- deeplearning.net. (2015, April 19). *MultiLayer Perceptron*. Retrieved from deeplearning.net: <http://deeplearning.net/tutorial/mlp.html>

- Feinstein, J. S., Adolphs, R., Damasio, A., & Tranel, D. (2011). The human amygdala and the induction and experience of fear. *Current biology* 21(1), 34-38.
- Friston, K. J., & Klaas, S. E. (2007). Free-energy and the brain. *Synthese*, 417-458.
- Fritzke, B. (1994). Growing cell structures—a self-organizing network for unsupervised and supervised learning. *Neural Networks*, 7(9), 1441-1460.
- Goldberg, D. E. (1990). *A Note on Boltzmann Tournament Selection for Genetic Algorithms and Population-Oriented Simulated Annealing*. University of Illinois at Urbana-Champaign, Urbana IL 61801 USA.
- Hull, C. L. (1943). Principles of behavior: an introduction. *New-York: Appleton-Century-Croft*.
- Karpathy, A. (2015, 05 01). *ConvNetJS*. Retrieved from <http://cs.stanford.edu/people/karpathy/convnetjs/started.html>
- Lacik, I. (2015, 04 31). *Github*. Retrieved from MultiLayer Perceptron JavaScript: <https://github.com/LacikIgor/MultiLayer-Perceptron-JavaScript>
- Malfaz, M., & Salichs, M. A. (2006). Emotion-based learning of intrinsically motivated autonomous agents living in a social world. *Proceeding of the ICDL 5: The Fifth International Conference on Development and Learning*. Bloomington, Indiana.
- Marsland, S. (2009). *Machine Learning An Algorithmic Perspective*. CRC Press.
- Maslow, A. H., Frager, R., & Cox, R. (1970). *Motivation and personality* (Vol. 2). (J. Fadiman, & C. McReynolds, Eds.) New York: Harper & Row.

- McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5.4, 115-133.
- Montgomery, K. (1954). The role of exploratory drive in learning. *Journal of Comparative and Physiological Psychology*, 60-64.
- neuroph.sourceforge. (2015, April 19). *Perceptron*. Retrieved from Perceptron: <http://neuroph.sourceforge.net/tutorials/Perceptron.html>
- Nevid, J. (2013). *Psychology: Concepts and applications* (4th ed.). Belmont.
- NN FAQ. (2015, 05 01). *Neural Network FAQ*. Retrieved from <ftp://ftp.sas.com/pub/neural/FAQ3.html>
- Oudeyer, P. Y., & Kaplan, F. (2007b). What is Intrinsic Motivation? A Typology of Computational Approaches. *Frontiers in Neurorobotics* 1:6.
- Oudeyer, P. Y., & Kaplan, F. (2008a). How can we define intrinsic motivation. *proceedings of the 8th international conference on epigenetic robotics: modelling cognitive development in robotic systems*. lund university cognitive science.
- Oudeyer, P., Kaplan, F., & Hafner, V. (2007a). Intrinsic Motivation Systems for Autonomous Mental Development. *IEEE Transactions on evolutionary computation*, Vol. 11, No. 2.
- Piero, F., Rossi, G., & Sanchez-Figueroa, F. (2010). Rich internet applications. *Internet computing, IEEE*, 9-12.
- Pileckyte, I., & Takáč, M. (2013). *Computational model of intrinsic and extrinsic motivation for decision making and action-selection*. Technical report TR-2013-038. Faculty of Mathematics, Physics and Informatics Comenius University, Bratislava.

- Reddiford. (2015, 04 25). *Algorithm for roulette wheel selection*. Retrieved from reddiford.co.nz/Algorithm-for-roulette-wheel-selection
- Richard, R. M., Rigby, S. C., & Przybylski, A. (2006). The motivational pull of video games: A self-determination theory approach. *Motivation and Emotion*, 30(4), 344-360.
- Russel, S., & Norvig, P. (1995). *A modern approach*. Prentice-Hall, Egnlewood Cliffs 25: Artificial Intelligence.
- Ryan, R., & Deci, E. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary educational psychology* 25(1), 54-67.
- Sai, R. (2015, April 19). *Linear Separability*. Retrieved from Blog Sai Rahul: <http://blog.sairahul.com/2014/01/linear-separability.html>
- Sejnowski, T. J., & Tesauro, G. (1989). The Hebb Rule for Synaptic Plasticity: Algorithms and Implementations. *Neural Models of Plasicity*, NY: Academic Press, 94-102.
- Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An Introduction*. Cambridge: MA, MIT Press.
- Tarnopolsky, Y. (2014, 25 04). *Essays*. Retrieved from The chemistry of money: <http://spirospero.net/Essay55.html>
- Wikibooks. (2015, 04 26). *Artificial Neural Networks*. Retrieved from Neural Network Basics: http://en.wikibooks.org/wiki/Artificial_Neural_Networks/Neural_Network_Basics

Appendix – User Manual

The resulting rich internet application is located on www.actorcritic.sk. It was implemented to provide the functionality enabling users view the simulation of the proposed model as well as enabling students and researchers to explore the divergence in the model behavior based on customizable algorithm parameters, various model scenarios, agent actuators and sensors. In the following subsections we provide a detailed user-guide that will hopefully prove useful for any future users.

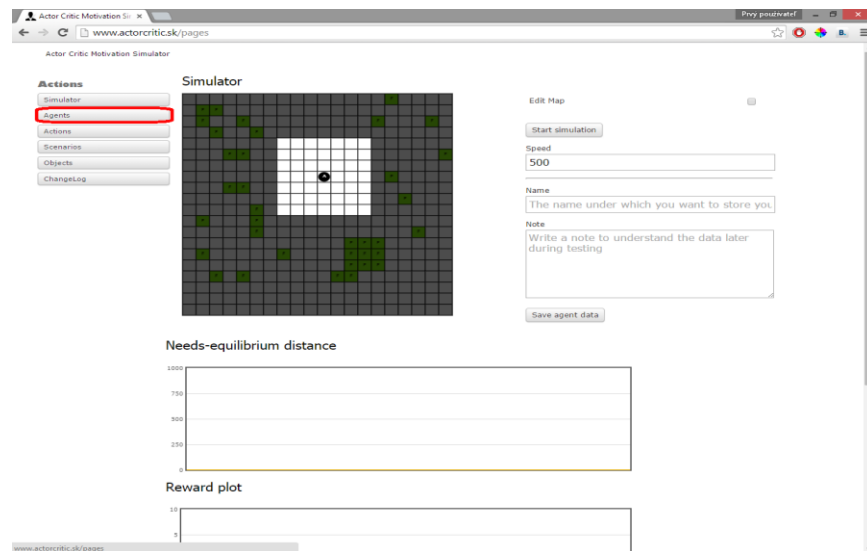
The overall structure of the given functionality is based on specifying parameters for abstract concepts of the model. These abstract concepts are *Agent*, *Action*, *Scenario* and *Object*. There is always only a single *Agent* that can perform multiple *Actions* in a single *Scenario* that might contain multiple *Objects*. There is also an entity that only serves as a research tool for keeping simulation logs – *Agent Log*. There are multiple microsites enabling a user to perform common functions over these entities – *list all entity instances*, *create a new entity*, *update an existing entity* and *delete an entity*. *Scenario* and *agent* entities also contain a *set to active* function – determining which scenario and agent is set to be active in the simulator. We provide a basic overview of the entities following with a detailed user-guide with screenshots:

- An *Agent* entity is described by its default intrinsic settings, the quality of its sensors, influence distance and a number of parameters used in the implemented AI algorithms.
- An *Action* entity can be viewed as an agent actuator. We implemented 4 types of specifiable actions (*eat*, *sleep*, *move*, *play*). These need to be added to an agent entity so that the agent can make use of an action. A user can specify multiple instances of any action type. These instances may differ in the way they affect the agent's internal state (e.g. resulting in one *eat* action that only affects the agent positively satisfying hunger, while a different *eat* action might also increase tiredness).
- A *Scenario* entity represents the world the agent exists in. It is represented by a map determining the starting location of the agent and a static location of the objects. Similarly to an *Agent* entity a *Scenario*

needs to have a relationship with an *Object* entity specified for it to be able to contain a given *Object*.

- An *Object* entity represents an object in the environment. There are 4 types of objects, that are implemented (*sleep*, *danger*, *food*, *toy*). Currently, the *sleep* object is unnecessary, and even though the *Object* entities can be described by specifying agent altering parameters, the only one that matters is the *pain increment* and the *range* of the object (specifying the influence of the object – implemented only for *danger*). We advise to keep matters simple and use *Action* entities to define how various actions alter the agent's internal state.

Create a new agent entity



15. Navigate to the agents section

ID	Title	Note	Starting hunger	Starting tiredness	Starting path	Starting boredom	Starting playfulness	Eye sight distance	Reach distance	Actor Learning rate	Critic Learning rate	Actor Hidden layer neurons count	Critic Hidden layer neurons count	Actor Hidden Activation Function
6	Boredom 0.1 - Threshold 2		0	0	0	0	0	3	0	0.01	0.1	30	30	SGH
8	Move, Sleep, Eat		0	0	0	0	0	3	0	0.01	0.1	30	30	SGH
9	Eat effect, sleep no effect	Test if learns to eat (after a period of time, only eat)	1e+09	0	0	0	0	3	1	0.1	0.1	6	12	SGH
10	Sleep effect, eat no effect	Test if learns to sleep (after a period of time, only sleep) -> determine sleep	100000	100000	0	0	0	3	1	0.1	0.1	6	12	SGH

16. This microsite lists all agent entities. You can filter entities based on various properties and also set an entity as active for the simulator. Click on the New Agent button to navigate to the microsite containing the functionality enabling New Agent entity creation.

Actor-Critic Motivation Simulator

www.actorcritic.sk/Agents/add

Actions

- Simulator
- Agents
- Actions
- Scenarios
- Objects
- ChangeLog

Add Agent

Name:

Note:

Starting hunger:

Starting tiredness:

Starting pain:

Starting boredom:

Starting playfulness:

Eye sight range:

Reach distance:

17. Fill in all the fields and click on the green „Submit” button on the bottom of the page to create a new Agent entity.

Give actions to the agent

Actor-Critic Motivation Simulator

www.actorcritic.sk/agents

Actions

- New Agent
- Simulator
- Agents
- Actions
- Scenarios
- Objects
- ChangeLog

Agents

ID	Title	Note	Starting hunger	Starting tiredness	Starting pain	Starting boredom	Starting playfulness	Eye sight distance	Reach distance	Actor Learning rate	Critic Learning rate	Actor Hidden layer neurons Count	Critic Hidden layer neurons Count	Actor Hidden Activation Function
6	Boredom o.l. Threshold 2		0	0	0	0	0	3	1	0.01	0.1	30	30	SGH
8	Move, Sleep, Eat		0	0	0	0	0	3	0	0.01	0.1	30	30	SGH
9	Eat	Test if learns to eat (after a period of time, only eat)	1e+09	0	0	0	0	3	1	0.1	0.1	6	12	SGH
10	Sleep	Test if learns to sleep (after a period of time, only eat)	100000	100000	0	0	0	3	1	0.1	0.1	6	12	SGH

1. To attach an action relationship to an agent entity navigate to the "edit" microsite of the entity. To do this, either doubleclick the row containing the set agent or click on the edit button.

Surprise threshold

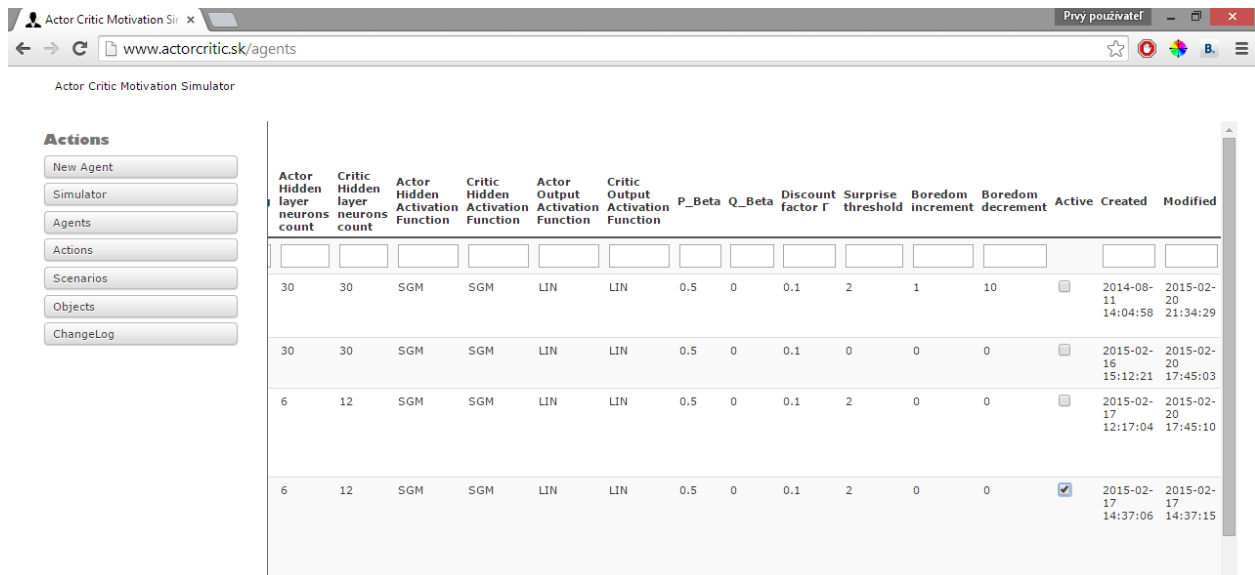
Boredom inc.

Boredom dec.

ID	Title	Hunger inc.	Tiredness inc.	Pain inc.	Boredom inc.	Playfulness inc.	Active
<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>	<input type="checkbox"/>
1	Move	1.1	1.2	0	0	0	<input type="checkbox"/>
2	Eat	-5	0.1	0	0	0	<input type="checkbox"/>
3	Sleep	0.1	-5	0	0	0	<input type="checkbox"/>
4	Sleep	0	0	0	0	0	<input checked="" type="checkbox"/>

2. In the bottom of the "edit" microsite we located a table with all available actions. The user can set those that should be active for the current agent by checking the checkboxes in the active column. In this screenshot, the last sleep action is active.

Set agent as currently active

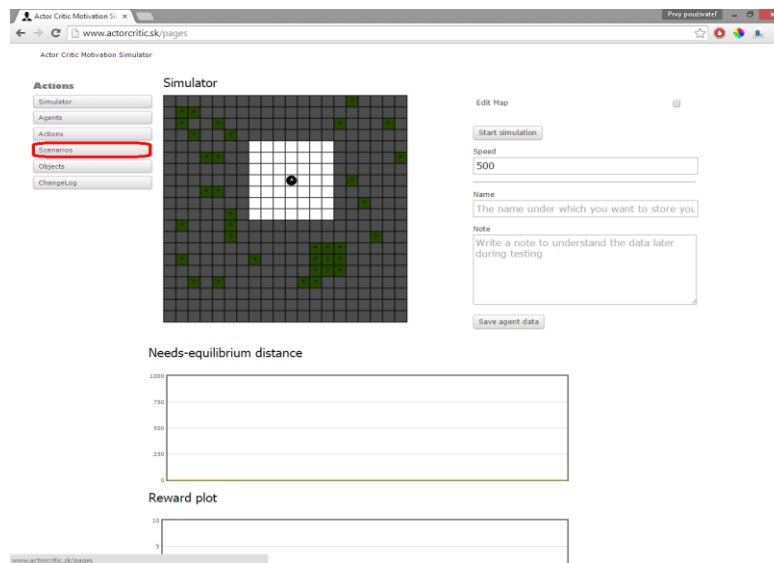


The screenshot shows the 'Actor Critic Motivation Simulator' web application. On the left, there is a sidebar with buttons for 'New Agent', 'Simulator', 'Agents', 'Actions', 'Scenarios', 'Objects', and 'ChangeLog'. The main area displays a table of agents. The table has columns for 'Actor Hidden layer neurons count', 'Critic Hidden layer neurons count', 'Actor Hidden Activation Function', 'Critic Hidden Activation Function', 'Actor Output Activation Function', 'Critic Output Activation Function', 'P_Beta', 'Q_Beta', 'Discount factor γ ', 'Surprise threshold', 'Boredom increment', 'Boredom decrement', 'Active', 'Created', and 'Modified'.

Actor Hidden layer neurons count	Critic Hidden layer neurons count	Actor Hidden Activation Function	Critic Hidden Activation Function	Actor Output Activation Function	Critic Output Activation Function	P_Beta	Q_Beta	Discount factor γ	Surprise threshold	Boredom increment	Boredom decrement	Active	Created	Modified
30	30	SGM	SGM	LIN	LIN	0.5	0	0.1	2	1	10	<input type="checkbox"/>	2014-08-11 14:04:58	2015-02-20 21:34:29
30	30	SGM	SGM	LIN	LIN	0.5	0	0.1	0	0	0	<input type="checkbox"/>	2015-02-16 15:12:21	2015-02-20 17:45:03
6	12	SGM	SGM	LIN	LIN	0.5	0	0.1	2	0	0	<input type="checkbox"/>	2015-02-17 12:17:04	2015-02-20 17:45:10
6	12	SGM	SGM	LIN	LIN	0.5	0	0.1	2	0	0	<input checked="" type="checkbox"/>	2015-02-17 14:37:06	2015-02-17 14:37:15

To set an agent as currently active for the simulation, check the checkbox of the desired agent. The checkbox is located in the active column and the user might need to use the horizontal slider to access it.

Create a new scenario



The screenshot shows the 'Actor Critic Motivation Simulator' web application with the 'Scenarios' section selected. The sidebar on the left has the 'Scenarios' button highlighted with a red box. The main area is divided into three sections: 'Simulator', 'Edit Map', and 'Needs-equilibrium distance'.

Simulator

The 'Simulator' section displays a grid map with green squares representing obstacles and a black dot representing the agent. The grid is 10x10 units.

Edit Map

The 'Edit Map' section contains a 'Start simulation' button, a 'Speed' input field set to 500, a 'Name' input field with the placeholder text 'The name under which you want to store you', and a 'Note' input field with the placeholder text 'Write a note to understand the data later during testing'. There is also a 'Save agent data' button.

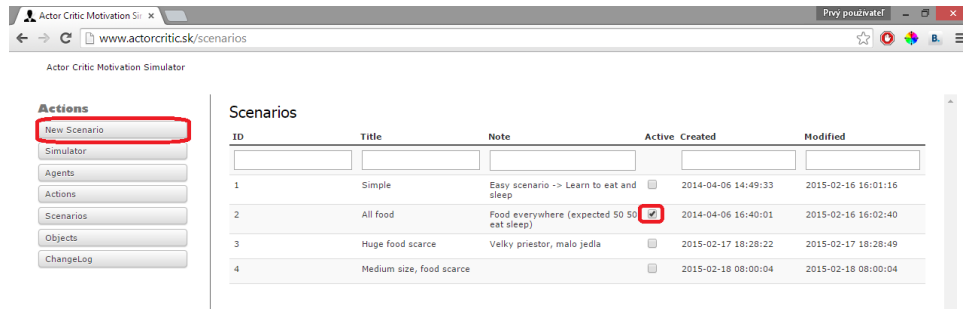
Needs-equilibrium distance

The 'Needs-equilibrium distance' section shows a line graph with the y-axis ranging from 0 to 1000. The graph shows a single horizontal line at y=0.

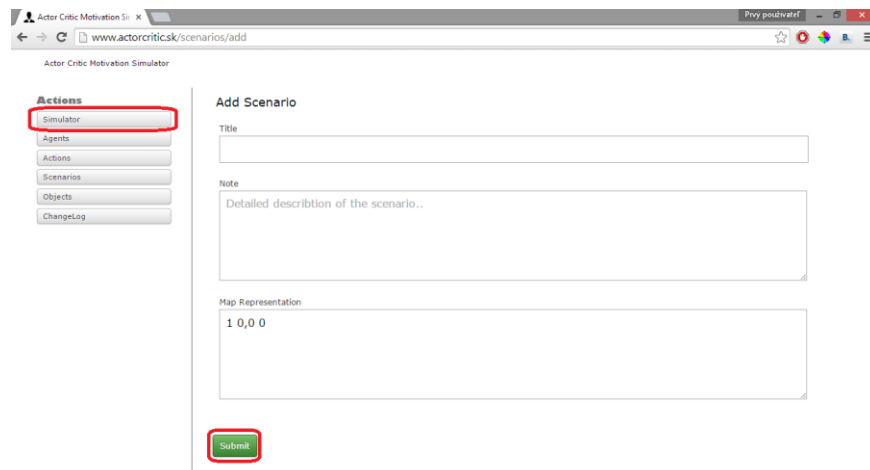
Reward plot

The 'Reward plot' section shows a line graph with the y-axis ranging from 0 to 10. The graph shows a single horizontal line at y=0.

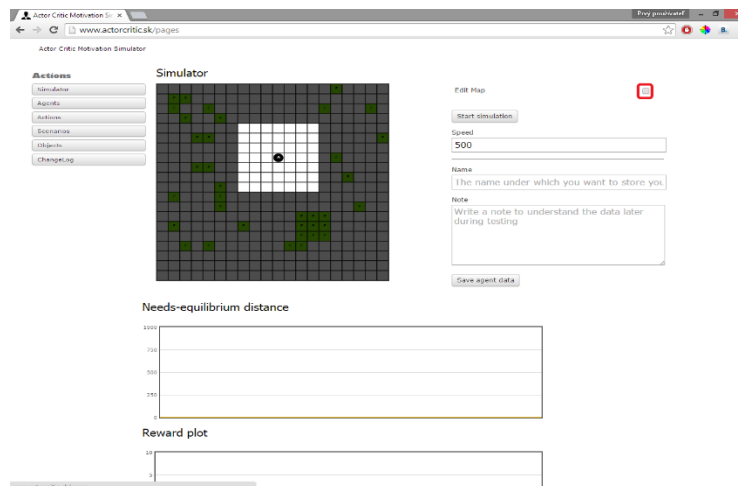
1. Navigate to the scenarios section.



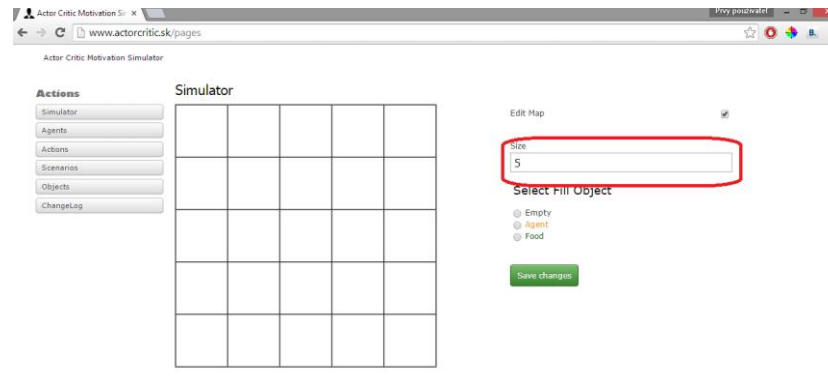
2. Click on the New Scenario button to navigate to the microsite enabling the creation of a New Scenario entity. An **active** scenario is set similarly to the microsite containing the list of agent entities.



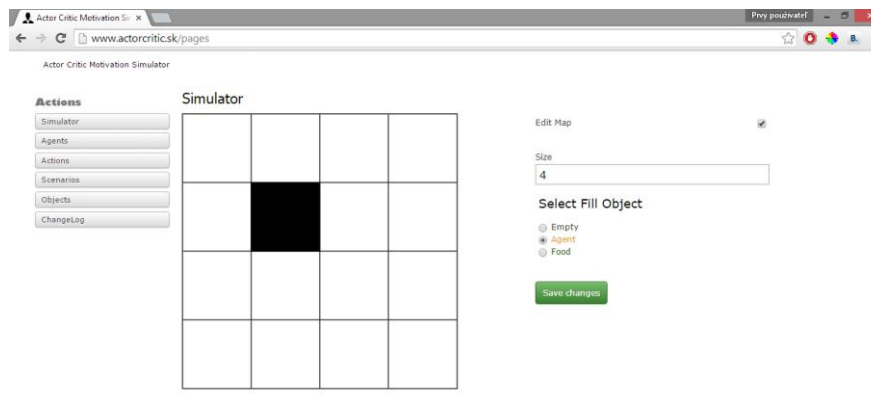
3. Fill in the Title and the Note, the Map Representation describes the scenario map in a literal way, it can be ignored for the time being. Save the Scenario entity by click on the green „Submit” button. Click on the „Simulator” button from any microsite in the application for a direct access to the main page containing the Simulator functionality (along with the map layout editing functionality).



4. To edit a map layout, check the Edit Map checkbox in the „Simulator” microsite.

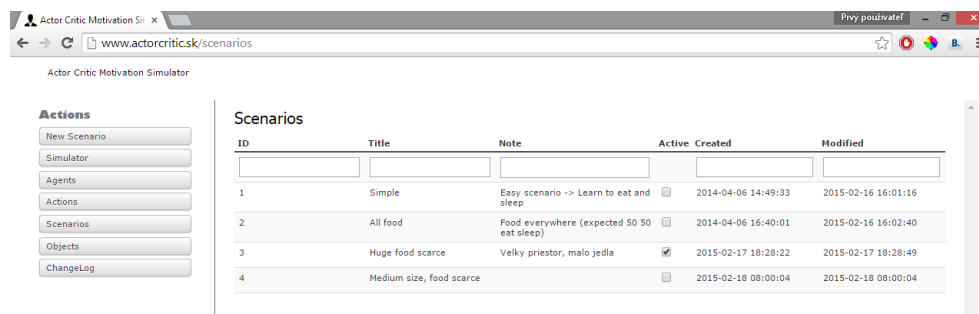


5. Select the desired size. The map will always be a square containing **size** squared identical squares. Each square represent a node on the map and can contain either any type of allowed object or be empty.

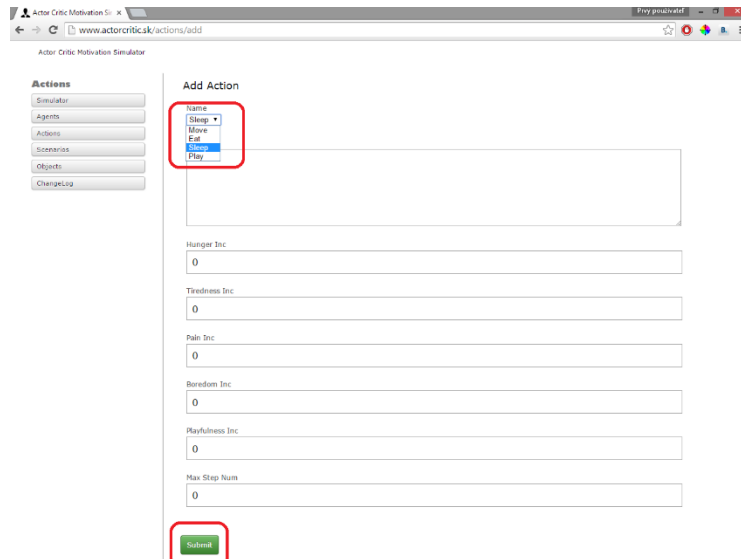


6. To assign an object to a node, first click on the radio button next to an item listed under the Select Fill Object header. Then click on the desired node. This action can be repeated until a different object is selected (with an exception of Agent object). Once content with the layout of the map, click Save Changes to apply the changes for this Scenario entity. Notice, that there is no option for selecting Danger, or Toy objects into the map. The reason for this is that a scenario has to be explicitly assigned any of the objects it might contain.

Add objects to a scenario

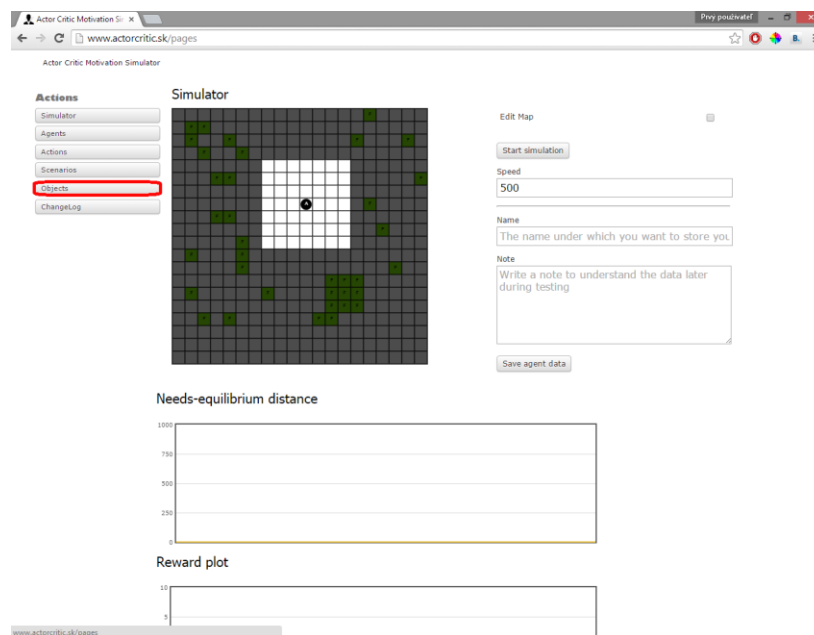


1. **Double-click** on a row containing the Scenario you want to update (and in the process assign objects to).

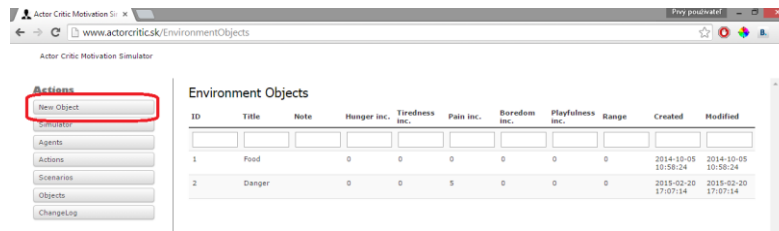


3. From the dropdown menu located under the "Name" label select the desired type of the action, set the agent internal state changing parameters of the action and submit it by clicking the green "Submit" button.

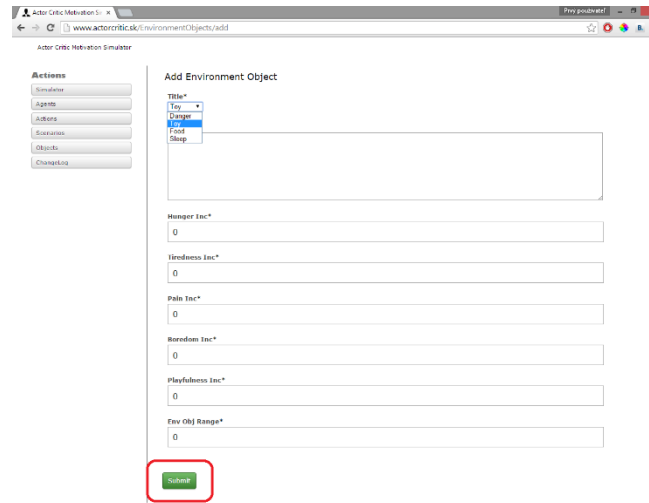
Create a new object



1. Navigate to the objects section.

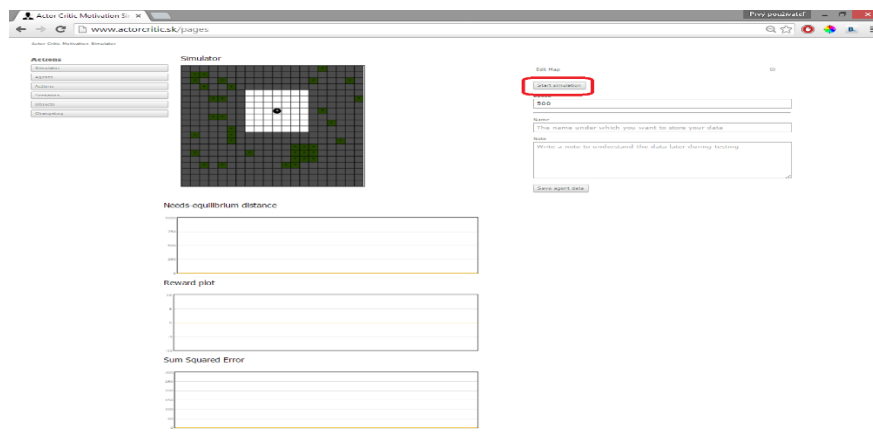


2. Click on the "New Object" button to navigate to the microsite containing functionality enabling new object entity creation



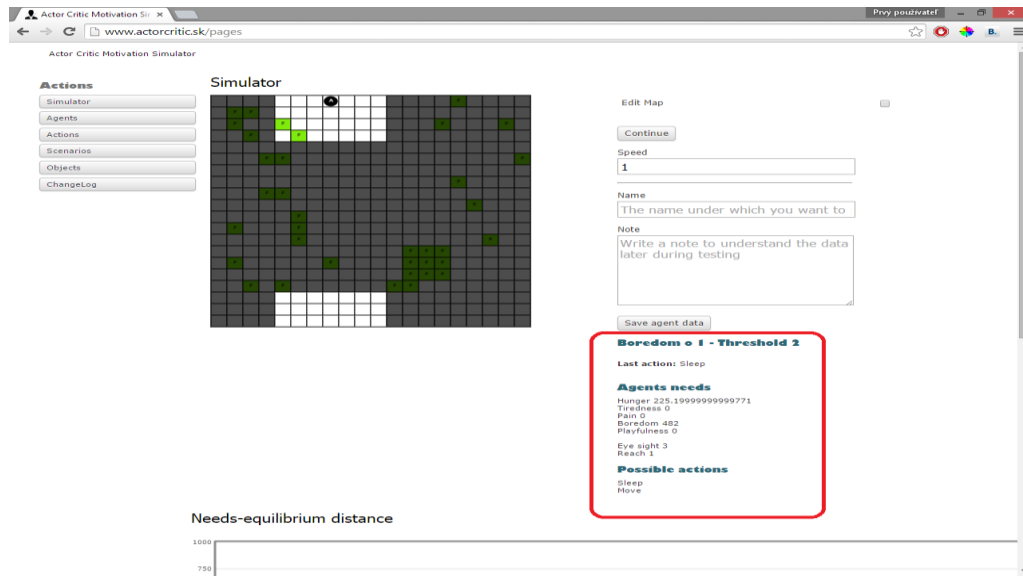
3. From the dropdown menu located under the "Title" label select the desired type of the Object. It is advised not to change any other parameters, but "Env Obj Range", which can serve for a range of danger objects affect. Save by clicking on the "Submit" button.

Run agent in the simulator

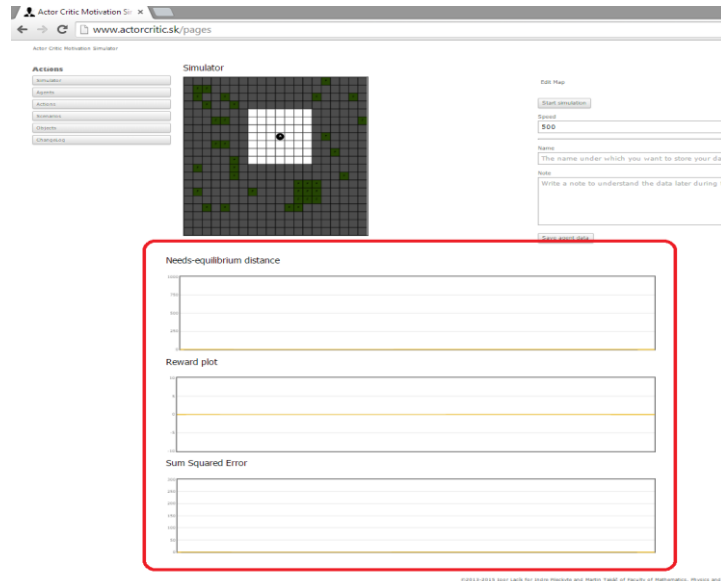


1. Once all the representing entities are set to the desired values, press "Start simulation". The simulation can be paused by clicking on the same button (which will say „Pause“). The simulation is started with a

default speed of 500 milliseconds between actions. To get the fastest possible speed, set it to 1 millisecond, to get a slower speed, set it to a higher value.

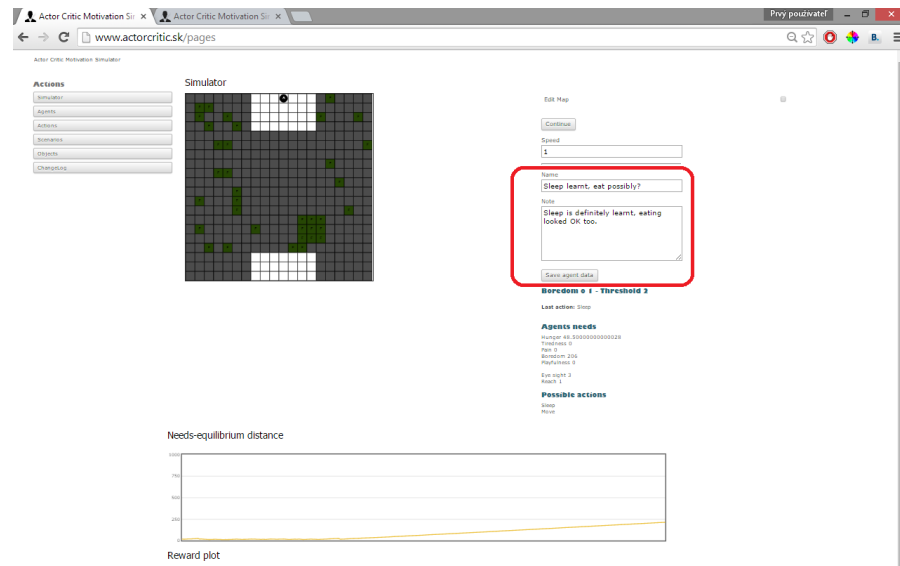


2. The agent's visual field is displayed adding lighter hues to the surrounding nodes. In the panel highlighted by the red rectangle the agent's current internal state can be explored along with the actions that can be performed to get to the next state as well as the last action performed.



3. There are 3 real-time plots located below the simulator map: **Needs-equilibrium distance**, **Reward plot** and **Sum Squared Error**. The first serves as a help to explain the expected model behavior. It represents the distance from a perfectly satisfying internal state. The reward plot shows the average reward obtained in the last 20 steps and the Sum Squared Error helps determine whether the **critic** is learning (if it stops oscilating, it is).

Save simulation logs



All the performed actions and the internal state is being logged during the simulation. To save the logs for future analysis, enter the log title into the input located under the label „Name” and a note about the simulation (e.g. qualitatively observed behavior). Then click “Save agent data” button.

Browse agent logs

Actor-Critic Motivation Simulator

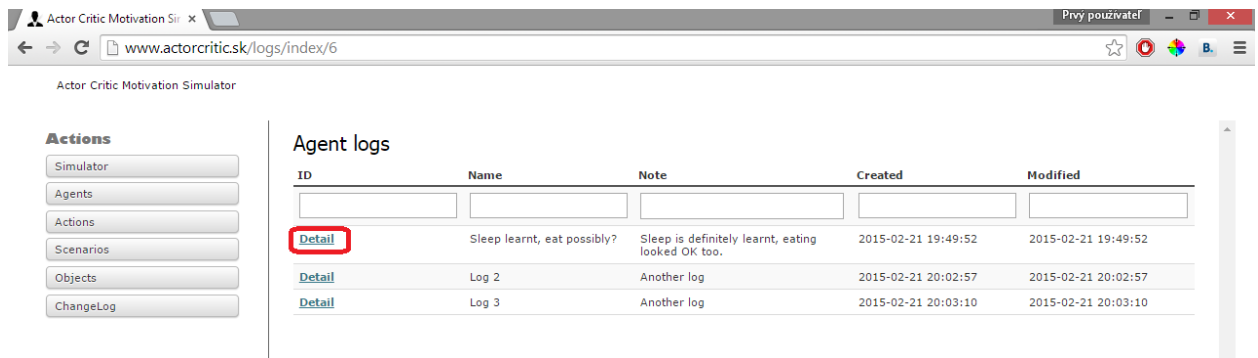
Agents

ID	Title	Note	Starting hunger	Starting tiredness	Starting pain	Starting boredom	Starting playfulness	Eye sight distance	Reach distance	Actor Learning rate	Critic Learning rate	Actor Hidden layer neurons count	Critic Hidden layer neurons count	Actor Hidden Activation Function
6	Edit Boredom 0 1 - Threshold 2		0	0	0	0	0	3	1	0.01	0.1	30	30	SGM
8	Edit Move, Sleep, Eat		0	0	0	0	0	3	0	0.01	0.1	30	30	SGM
9	Edit Test if learns to eat (after a period of time, only eat)	Test if learns to eat (after a period of time, only eat)	1e+09	0	0	0	0	3	1	0.1	0.1	6	12	SGM
10	Edit Sleep effect, eat no effect	Test if learns to sleep (after a period of	100000	100000	0	0	0	3	1	0.1	0.1	6	12	SGM

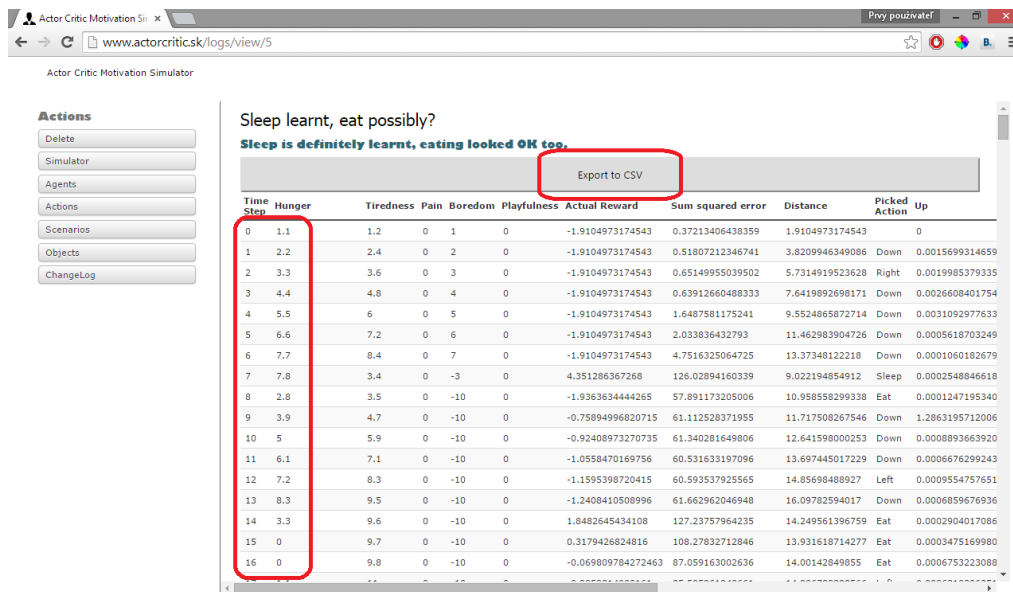
1. The logs for each agent are located in the respective agent's „Edit” section. There can be a potentially unlimited number of logs for each agent.



2. In the left panel, click on the button titled “Agent Logs”.



3. All the logs saved for the agents simulation scenarios can be listed in this section. To explore detailed data from a concrete log, click on the “Detail” button in the row summarizing the log.



©2013-2015 Igor Lacić for Indre Pileckyte and Martin Takáč of Faculty of Mathematics, Physics and Informatics, Comenius University, Center for Cognitive Science

4. All the data from the simulation are displayed in a table (starting from top with the first state). It is possible to export the data to a CSV file which enables researchers to explore the data using a 3rd party program such as Microsoft Excel, or Open Office Calc.