Esmeralda Brkic, BSc

Deep Reinforcement Learning with slow feature analysis

Master's Thesis
to achieve the university degree of
Master of Science
Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor
Univ.-Prof. Dipl-Ing. Dr.techn. Robert Legenstein

Institute of Theoretical Computer Science
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Robert Legenstein

Graz, October 2020

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis.

Date

Signature

Abstract

Deep reinforcement learning has been shown to perform well on tasks such as video games. In deep Q-learning, deep convolutional networks are often trained end-to-end. In this thesis we propose to use slow feature analysis (SFA) as an unsupervised learning principle to extract a low-dimensional feature space from input images. A convolutional SFA network is first trained in an unsupervised manner. A simple deep Q-learning network can then be trained on the extracted features. We test this principle on a simple navigation problem and on the Atari game Pong. We find that SFA is able to extract useful features in these cases and that these features can be used to train a well-performing agent.

Acknowledgments

It was a great honor to work with my mentor Prof. Dr. Robert Legenstein and I would like to thank him for his understanding and unconditional help during the process of making this master thesis. He is the one who lead me through the whole process of research. I am very grateful for all the advices, suggestions, trust, patience and time for numerous questions. Thanks for everything.

Contents

List of Figures

List of Tables

1 Introduction

Artificial Intelligence is one of the most important topics in today's world, as evidenced by the fact that the number of papers on this topic is dramatically increasing. Among the many interesting topics, Reinforcement Learning (RL) stands out, which is a learning setting that is also the most natural for humans. Just as in real life, where we get praise or criticism for every action, so does our agent in RL. In the case of a well-done series of steps, he gets a positive reward, and a negative reward is experienced after bad decisions.

Based on that, we can say that the executor of the action is an **agent**. **Action** represents the set of all possible moves, the agent can make. The world in which the agent acts is called an **environment**. A **state** is a concrete and immediate situation in which the agent finds itself. Finally, **reward** is the feedback by which we measure the success or failure of an agent's actions in a given state.

Video games have often been used as a benchmark for RL algorithms. In particular, the Atari 2600 games were used to evaluate deep RL methods [1, 2, 3, 4]. Usually, these methods use deep neural networks that represent a Q-function as a policy, which are trained end-to-end. Unsupervised learning is an intriguing alternative [5, 6]. The idea is to learn useful features without supervision or reinforcement, and then train an RL-agent on these features. This should lead to more efficient use of rewards, because they are not used to learn the state representation, but only the appropriate actions. We focus in this thesis on unsupervised learning with Slow Feature Analysis (SFA) as this method has been proven successful in a related context previously [7, 8, 9].

Since Atari games need a huge number of iterations for an RL algorithm to create a stable trained model, we decided to create an auxiliary game, which is simpler but still requires the same learning treatment as Atari games. We called it the "Simple Navigation Game", represented by a single agent, whose position can be anywhere on a board, and whose task is to find the closest path to the goal, which can be the upper left corner, upper right corner, lower left corner or lower right corner.

To prove the advantage of different algorithms, we use Reinforcement Learning on images, Reinforcement Learning on coordinates and Reinforcement Learning performed on Slow Features (SFA). We performed experiments first on the Simple Navigation Game, to determine all the metaparameters for RL algorithms such as Q-learning and Deep Q-learning. We used the same parameters later for the Atari Pong Game. Our goal is for the algorithms to behave the same on both games, where more epochs are expected in the Pong game, because of its complexity (large number of possible states).

The thesis is structured as follows. In Section 2, we present Slow Feature Analysis (SFA). Accordingly, we present its mathematically definitions, how it can

be used in terms of hierarchical networks, and discuss it in a simple example. Section 3 contains the general description of Reinforcement Learning, as well as Q-Learning, Q-Learning with linear function approximation and Deep Q-Learning. Then, in Section 4, we discuss related work. Numerous experiments, are described in in Section 5. At the end of the thesis, in Section 6, we give a conclusion as well as ideas for future work. A graphical user interface has been developed in order to simplity the setup of experiments. This GUI is described in Appendix A.

2 Slow Feature Analysis

Slow Feature Analysis is an unsupervised learning method, developed by Laurenz Wiskott, which extracts slowly varying features from a quickly varying input signal. This method is used for self organization of complex-cell receptive fields, the recognition of whole objects invariant to spatial transformations, the self-organization of place-cells, extraction of driving forces, nonlinear blind source separation, etc. [10].

2.1 Algorithm

In order to understand the slowness principle let us describe image representation on the screen displays where an image is made up of $m \times n$ pixels which represent colored points. The frequent change of individual pixels compared to the time-consistency of objects and attributes leads to the conclusion that relevant features in sensory data vary on a relatively slow time-scale. The slowness principle is based on the difference in time-scales, where the extraction of slowly changing features of sensory signals is supposed to restore the external causes of sensory input. Humans recognize the objects on the screen, as well as their movements and they can conclude the next movement of some object according to its previous behaviour. Similarly, the principle of slowness is a natural hypothesis for the functional organization of the visual cortex and the processing of visual information.

This non-linear optimization problem is demonstrated in Figure 1. Mathemat-

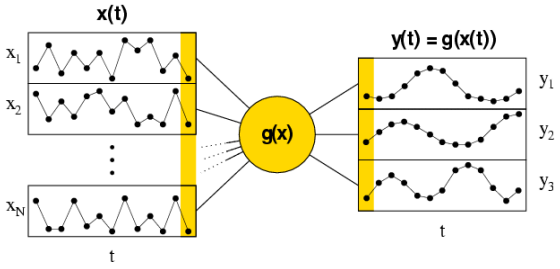

Figure 1: Schematics of the optimization problem solved by Slow Feature Analysis. On the given input signals, represented over time, a non-linear function is used which calculates output signals y(t). For g, the global optimum is calculated.

ically, the Slow Feature Analysis algorithm has as its input a multivariate time series X and an integer n indicating the number of features to be extracted from the series, where n should be lower than the dimension of the time series. The algorithm determines n functions as it is shown in the equation (1).

$$y_j(t) := g_j(\mathbf{x}(t)) \tag{1}$$

In order to obtain slow features, the average of the quadratic time derivative of each y_j is minimized. Accordingly, maximization of the slowness of the features when given a input signal $\mathbf{x}(t)$, finds functions $g_j(\mathbf{x})$ that minimize

$$\Delta(y_j) := \langle \dot{y}_j^2 \rangle_t \tag{2}$$

under the constraints represented in equations 3, 4 and 5

$$\langle y_j \rangle_t = 0 \tag{3}$$

$$\langle y_j^2 \rangle_t = 1 \tag{4}$$

$$\forall i < j : \langle y_i, y_j \rangle_t = 0 \tag{5}$$

where equations (3), (4) and (5) represent zero mean, unit variance and decorrelation with order, respectively.

The objective function (2) measures the slowness of the feature. The zero-mean constraint (3) makes the second moment and variance of the features equivalent and simplifies the notation. The unit variance constraint (4) discards constant solutions. The final constraint (5) decorrelates the features and induces an ordering on their slowness. This means that first comes the slowest feature, then the next less slow feature that is orthogonal to the one before it and so on [10].

2.2 Input expansion

Hence, SFA requires an optimization over functions g_j, rather than over a set of parameters. By restricting the functions g_j to a finite dimensional space, such as all polynomials of degree two, the problem of variation can be transformed into more conventional optimization via the coefficients of the basis of the functional space (e.g., all monomials of degrees 1 and 2). This makes the problem easier to solve where the usage of algebraic methods is a basis of SFA algorithm, as shown below.

Let us assume that the two dimensional input signal is given:

$$x_1(t) := 3\sin(t) + \cos(6t)^2 \tag{6}$$

and

$$x_2(t) := \cos(6t) \tag{7}$$

where slow varying "feature" is hidden:

$$y(t) = x_1(t) - x_2(t)^2 = 3\sin(t) + \cos(6t)^2 - \cos(6t)^2 = 3\sin(t) \tag{8}$$

and can be extracted with a polynomial of degree two, which leads to

$$g(\mathbf{x}) = x_1 - x_2^2. \tag{9}$$

Figure 2 shows all input signals described above.

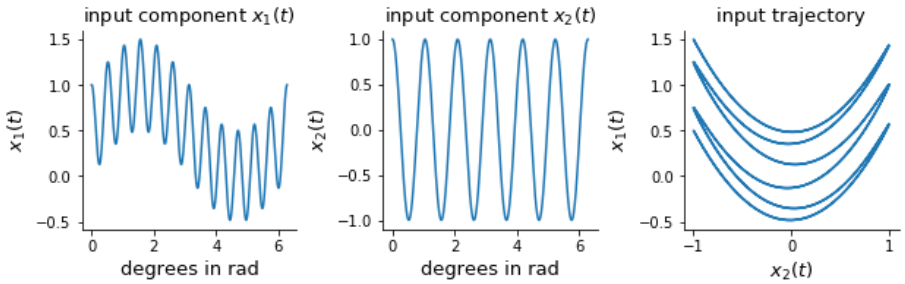

Figure 2: Input signal composed of two separates signals, which are described in the text above. Left panel shows $x_1(t)$, in the middle $x_2(t)$ and on the right side the 2D input trajectory $(x_1(t), x_2(t))$ is plotted.

Finding the function $g(\mathbf{x})$, which extracts slow features S, assumes the following steps:

1. **Transformation of the non-linear problem to a linear one.** Let us consider the example from above where the slow signal can be obtained by a polynomial of degree two. Now consider a polynomial expansion of input signal: $z_1 := x_1, z_2 := x_2, z_3 := x_1^2, z_4 := x_1 x_2$ and $z_5 := x_2$. According to this signal representation, every polynomial of degree two, can be described as a linear problem using the new mentioned signals (z).

2. **Normalization of expanded signal.** It is calculated by mean subtraction and application of a linear transformation in a way, to have unit variance in all directions.

3. **Measurement of temporal variation in the normalized space.** Calculation of time derivative $\dot{z}(t)$

4. **Extraction of slowest-varying features.** The direction, in which the signal varies most slowly, is the one which contains least variance of the time derivative. If more outputs are present, then orthogonal directions with the smallest variance should be taken.

The resulting function $y(x)$ is shown in Figure 3, also known as extraction function.

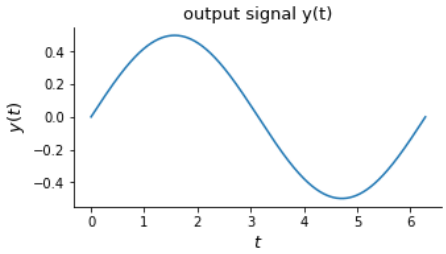

Figure 3: Output signal $y(t)$ which is a result of subtraction of x_1 and x_2^2.

2.3 The hierarchical network

Theoretically, Slow Feature Analysis supports an unlimited number of input channels. This is not possible in practice because it leads to a problem of dimensionality due the limited computational resources. For example, just a quadratic expansion of an input image of 100 by 100 pixels leads to a dimension of about 50 million, which is a huge number and too complex for the calculation and data manipulation.

In order to avoid the dimensionality problem, the input data should be separated into several subsets. Accordingly, the slowest-varying feature will be extracted per subset, where its outputs are used as inputs for the another iteration of SFA in the next layer. In this way, through every iteration a larger fraction of data is integrated in the new solution. According to that, the curse of dimensionality is avoided.

This type of hierarchy and separation of input data leads to a new representation of the visual system. Accordingly, the feed-forward neural network was the first and simplest type of artificial neural network in which information moves only in one direction, forward, from input nodes, through hidden nodes (if any) to output nodes.

One example of this type of networks is shown in Figure 4, where it is demonstrated how the nodes are separated. The computational complexity of the SFA algorithm is of order $O(NI^2 + I^3)$, with N denoting the number of samples and I the input dimensionality, as written by Wiskott and Escalante [11].

2.4 Application example

In this section the usage of Slow Feature Analysis in the MDP (Modular Toolkit for Data Processing) software suite is shown.

Let us assume that input signals are chaotic time series obtained from a non-stationary logistics map in the sense that the underlying parameter is not static

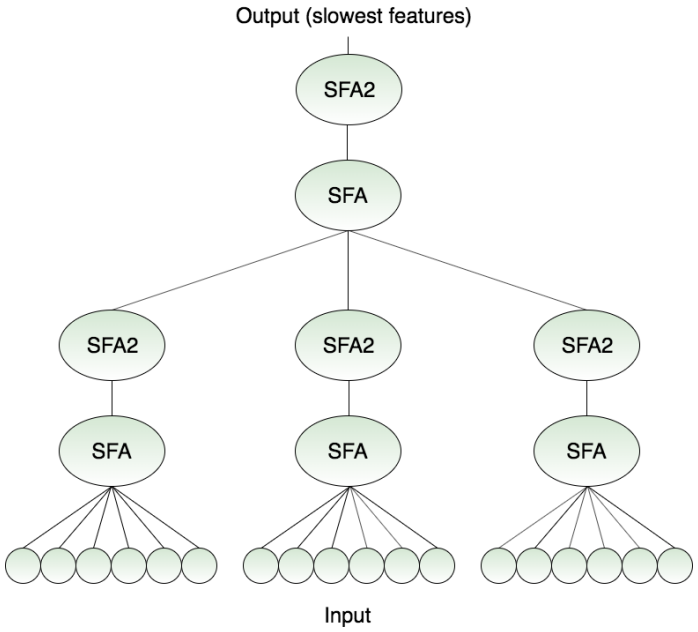

Figure 4: Hierarchical Slow Feature Analysis Network with two layers where on every input node a linear SFA for dimensionality reduction is performed (SFA). Then on its output a quadratic SFA is applied (SFA2) .

and it changes uniformly over time. The goal is to extract a slowly varying variable which is hidden in the observed time series.

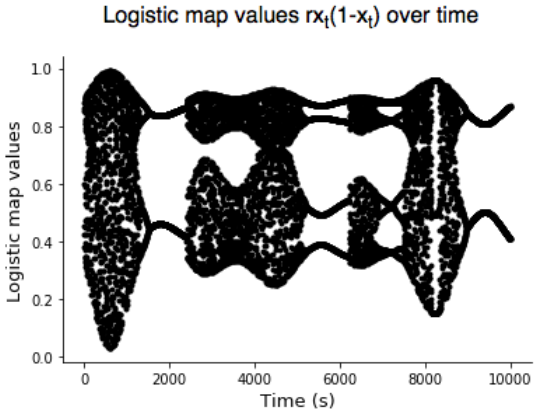

Figure 5: Time series calculated by the logistic equation $rx_t(1 - x_t)$, where t represents time and r is a slowly varying driving force generated as a sum of three sine functions: $sin(6\pi t)$, $sin(8\pi t)$ and $sin(10\pi t)$ [12].

A slowly varying driving force is generated by combination of three sine functions $sin(6\pi t)$, $sin(8\pi t)$ and $sin(10\pi t)$.

Input signals are shown in Figure 5. On this input signal (Figure 5) SFA is performed with polynomials of degree 3. First, a node is used that embeds

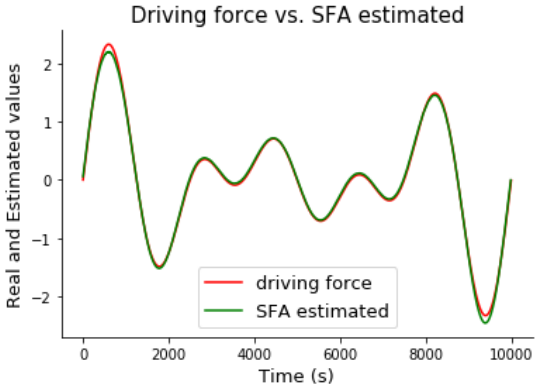

Figure 6: Real input signal of driving force vs. estimated driving force by SFA. Both signals are approximately same, which shows that SFA was able to extract the slow driving force.

the 1-dimensional time series in a 10-dimensional space. Then, the signal is expanded in the space of polynomials of degree 3 on which an output SFA node with output dimension size 1 is added.

To enable direct comparison, the driving force is rescaled to have zero mean and unit variance. According to that, the real driving force is plotted together with the driving force estimated by SFA in Figure 6.

Figure 6 shows the well learned model from which any input signal can be extracted with minimal deviations.

3 Reinforcement Learning

Reinforcement learning (RL) is one of the most important learning paradigms together with supervised and unsupervised learning. For example, humans use reinforcement learning to learn through interaction with the environment and their own experience.

An example is how a child learns. Let us assume that the child is outside and feels cold. Somewhere in the distance, it sees a fire and goes towards the fire. When it gets too close, the child feels warm. In this way, the child learned from its own experience that it is comfortable to stand by the fire (positive reward). In the other case, the child gets too close to the fire and tries, for example, to touch it, automatically its sensory receptors inform the brain that its hand is hurting (negative reward). Then the child instinctively removes its hand from the fire and learns that the fire can only be used from a sufficient distance, because it produces heat, otherwise it can hurt itself.

The same learning methods is used by reinforcement learning. It is not a type of supervised learning, where the prediction model is determined according to given inputs and outputs, for example images and its descriptions. Here the agent learns through its behaviour, rewards and previous knowledge. For example, the agent does some action and waits for information of its influence on the environment. According to the result and its reward, it learns that the used action was correct or wrong for the given situation.

Basic examples of reinforcement learning are solving different games, for instance Atari games which are used in the experimental part of this work.

3.1 General Algorithm Explanation

Typically, a Reinforcement Learning Setup includes two components, an agent and an environment, as can be seen in Figure 7. We describe here the RL problem according to [7] and [13].
The environment starts by sending a state to the agent, which chooses an action according to its knowledge about the given state. Based on the agent's choice, the new state will be sent by the environment which represents the result-state for a given action. With the result-state comes also the new reward and information if the game is finished. Then the agent's knowledge will be updated and all this steps will be repeated, until the given number of iteration is reached or the game is over.

According to that, the **agent** is the learner and decision maker. He interacts continually with an environment, getting reward for every done step/action. The **environment** also gives rise to rewards, which the agent seeks to maximize over time through its choice of actions. The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$.

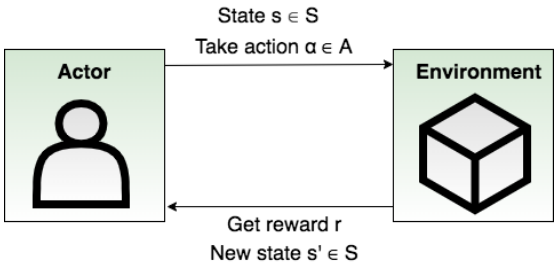

Figure 7: Reinforcement Learning Representation: Agent, according to the action which he has performed on the current state, gets the new state and a reward, which are used as learning signals for the RL algorithm.

a set of possible states S. For each state $s \in S$ there is a set of possible actions $A(s)$.

With every time step the agent receives the **state** $s_t \in S$ for which he should choose an **action** $a_t \in A(s_t)$. As a consequence for chosen action, the agent gets the **reward** $r_{t+1} \in \mathbb{R}$, which can be seen as a measure of the desirability of performing action a_t in state s_t. This type of interaction results in a trajectory: $s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, ...$

In this case, the random variables r_t and s_t have well defined discrete probability distributions dependent only on the preceding state and action. That can be described by

$$p(s', r \mid s, a) = Pr\{s_t = s', r_t = r \mid s_{t-1} = s, a_{t-1} = a\}, \tag{10}$$

where $s', s \in S$, $r \in \mathbb{R}$ and $a \in A(s)$.

The expected reward in state s when performing action a can be computed as

$$r(s, a) = E[r_t \mid s_{t-1} = s, a_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r \mid s, a). \tag{11}$$

In the same way expected rewards can be determined for the current state, reward and next state tuple by

$$r(s, a, s') = E[r_t \mid s_{t-1} = s, a_{t-1} = a, s_t = s'] = \sum_{r \in R} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)}. \tag{12}$$

The behaviour of the agent time t is determined by its policy π_t. Mathematically, $\pi_t(s, a)$ gives the probability that action a is chosen if the agent is in state s.

The expected discounted return R_t at time t is defined as

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... \\ &= \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \end{aligned} \tag{13}$$

where $0 \leq \gamma \leq 1$ is called the discount rate.

The goal of reinforcement learning is to find a policy π such that the expected discounted reward is maximized.

3.2 Q-Learning

Q-learning is a value-based reinforcement learning algorithm. Its main task is to find the best action to take on the given current state according to its own/previous knowledge. It is off-policy because the policy used to generate behaviour doesn't need to be the same as the one that is optimized. It uses a Q-table which stores for each state-action pair (s, a) the expected action a and then following the agent's policy π:

$$Q^{\pi}(s, a) = E[R_t \mid s_t = s, a_t = a]. \tag{14}$$

The Q-table is initialized randomly and the updated while the agent performs actions in the environment. Update of the Q table is represented by the equation (15)

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_{\{a_{t+1}\}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]. \tag{15}$$

where the following variables are used:

1. s_t - state at time t
2. a_t - action at time t
3. r_t - reward at time t
4. γ - discount factor
5. α - learning rate

Pseudo code is shown in Algorithm 1.
It remains to be defined how the agent takes actions based on its Q-table, i.e. how policy is defined. The policy has to be stochastic because the agent has to try out various actions in each state in order to find out which one performs best. There are several possibilities available. We describe here the ϵ-Greedy Policy.

Algorithm 1 Q-Learning

Initialize $Q(s, a)$ arbitrarily
for each episode: **do**
    Initialize s
    for each step of episode: **do**
        Choose a from s using policy derived from Q (e.g., ϵ-greedy)
        Take action a, observe r and s'
        Update $Q(s, a)$ with $Q(s, a) + \alpha \left[r + \gamma \max_{\{a'\}} Q(s', a') - Q(s, a) \right]$
        Update s with s'
    end for
    Until s is terminal
end for

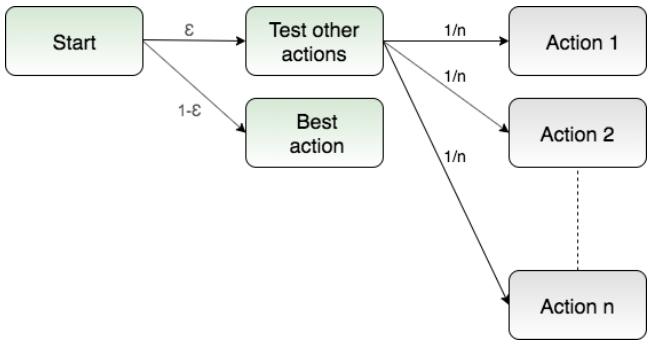

Figure 8: $\epsilon-$Greedy Policy Example of usage. This algorithm at any moment exploits/chooses the best action with a probability of 1-Epsilon whereas, explores with probability of Epsilon. There is one negative side of this algorithm, when exploring, it chooses equally among all possible actions which means that it is equally likely to choose the worst and the best action.

3.2.1 ϵ-Greedy Policy

This is one of the simplest possible algorithms for trading off exploration and exploitation. There is a decision made in the current step where one of two possibilities can be used. First possibility is a random action and second one is a deterministic action. The use of each one depends on the ϵ value which should be in a range from 0 to 1. At the start of training it is often 1, which leads to huge number of random actions. More specifically, the agent has a better chance of learning how to react in a situation by trying out a number of different actions and getting rewards for them. According to this, the agent learns much faster and builds a more stable model. Then ϵ will get lower and lower, depending on the algorithm type.

For example one can use ϵ value 1 for a first 20^4 iterations and then ϵ multiplied by 0.999 in every iteration.

The probability of taking random action is known as Epsilon (ϵ) in this algorithm and 1-Epsilon ($1-\epsilon$) as the probability of exploiting the recommended action, which is the action with the maximum Q-value in the current state.
If $\epsilon = 1$, the agent should always explore and never act greedily with respect to the action-value function. In comparison to that, using $\epsilon < 1$ causes a good balance between exploration and exploitation.

Exploitation is based on making the best decision according to given the current information and exploration should gather more information and explore possible situations.

3.3 Q-Learning with linear function approximation

In linear function approximation, the Q-value is approximated by a linear function of the state $\mathbf{s}$. Here, $\mathbf{s}_t \in \mathbb{R}^d$ is a d-dimensional vector of features that describes the current state. For l actions, we represent the action at time t as an integer $a_t \in \{1, ..., l\}$. The Q-values $\mathbf{q} \in \mathbb{R}^l$ of all possible actions in a given state $\mathbf{s}_t$ is then given by a linear function:

$$\mathbf{q} = \mathbf{W} \cdot \mathbf{s}_t \tag{16}$$

where $\mathbf{W} \in \mathbb{R}^{l \times d}$ is a weight matrix.

We obtain the value of action a in state $\mathbf{s}_t$ as the a-th component of q:

$$Q(\mathbf{s}_t, a) = \mathbf{q}_a \tag{17}$$

Consider that action a_t has been taken in state $\mathbf{s}_t$ and reward r_t has been received. We update the a_t-th row $\mathbf{w}_{a_t}$ of W as follows:

$$\mathbf{w}_{a_t} = \mathbf{w}_{a_t} + \alpha \left[r_t + \gamma \max \{\mathbf{W} \cdot \mathbf{s}_{t+1}\} - \mathbf{w}_a \cdot \mathbf{s}_t) \right] \cdot \mathbf{s}_t^T \tag{18}$$

where $\mathbf{s}_{t+1}$ is the new state and the max operator returns the maximum element of the vector $\mathbf{W} \cdot \mathbf{s}_{t+1}$.

3.4 Deep Q-Learning

Simple Q-Learning creates a Q-table with entries for every state and action pair. For environments with large state spaces this algorithm is ineffective.

Let us take an example environment with 10^4 states and 10^3 actions per state. This situation leads to the creation of Q-table with 10^7 (10 million) cells, which causes a performance problem, even for modern computers.

This represents two problems:

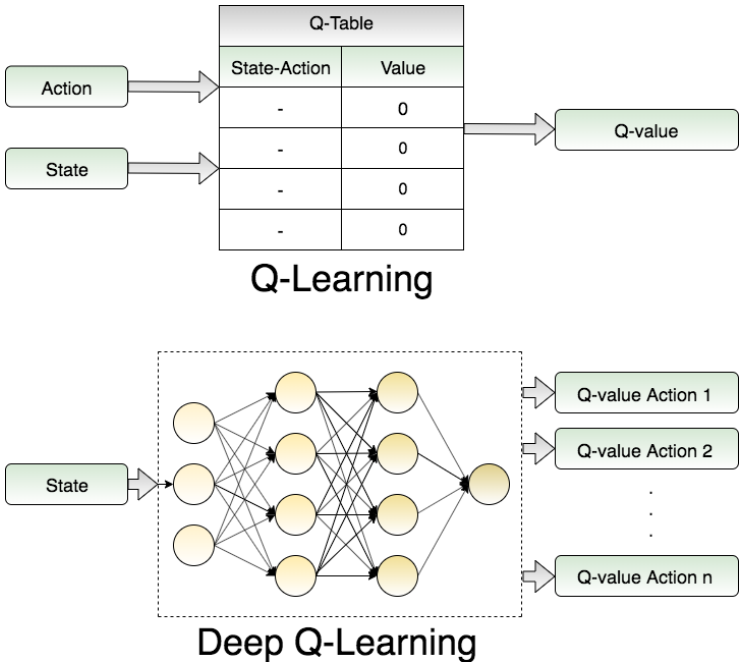

Figure 9: Q-Learning and Deep Q-Learning logic where Q-Learning Algorithm contains a Q-table which is a reflection of state-action combinations. Deep Q-Learning has a neural network with at least one hidden layer and it is used for more complex problem (more states and actions).

First, the amount of memory required to save and update that table would increase as the number of states increases.
Second, the amount of time required to explore each state to create the required Q-table would be hard to realize.

The main idea of deep learning is the approximation of Q-values with machine learning models such as neural network. Then, the created neural network with input, output and hidden layers is used for deep learning, as is shown in Figure 9.

3.4.1 Experience Replay

The following pseudo-algorithm (Algorithm 2) implements Deep-Q Learning with Experience Replay.

Instead of learning Q values for state and action pairs as they occur during the simulation or actual experience, the Experience Replay algorithm stores the collected data [state, action, reward, next state] in some table/array/list.

Algorithm 2 Deep Q-Learning with Experience Replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for each episode=1,M **do**
    Initialize state s_t
    for each t=1,T: **do**
        With probability ϵ select a random action a_t
        otherwise select $a_t = \max_{\{a\}} Q^*(s_t, a; \theta)$
        Execute action a_t and observe reward r_t and state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Set $s_{t+1} = s_t$
        Sample random minibatch of transitions (s_t, a_t, r_t, s_{t+1}) from D
        $y_j = r_j$ if terminal s_{t+1}, otherwise $y_j = r_j + \gamma \max_{\{a'\}} Q(s_{t+1}, a'; \theta)$
        Perform a gradient descent step on $(y_j - Q(s_t, a_j; \theta))^2$
    end for
end for

As an initialization is used a memory with 500.000 such tuples, which can grow during learning up to 1.000.000 (one million)[14].

The following variables are used in the pseudo code:

1. ϵ - exploration
2. r - reward
3. s_t - current state
4. a_t - current action
5. γ - discount factor

4 Related work

This work and its experiments are based on a number of papers which are related to Reinforcement Learning, Feature Extraction and their combination for complex problems. In this work, we are considering Atari games which contain a significant number of possible states.

4.1 Reinforcement Learning on Slow Features of High-Dimensional Input Streams

In [7], slow features are used as an input for the Q-Learning algorithm where the authors used two phases:

1. Creation of Slow Feature model and
2. Reinforcement Learning based on Feature Extraction.

For the Slow Feature Analysis a 4-layer hierarchical network is used whose input is a 155x155 pixels image. In the first three layers the dimensionality is reduced to 32 outputs and the last one, output SFA layer, is reduced to one array of 64 elements, which is later used as an input for a Reinforcement Learning algorithm, to be more precise, Q-Learning [7]. A similar principle was used in

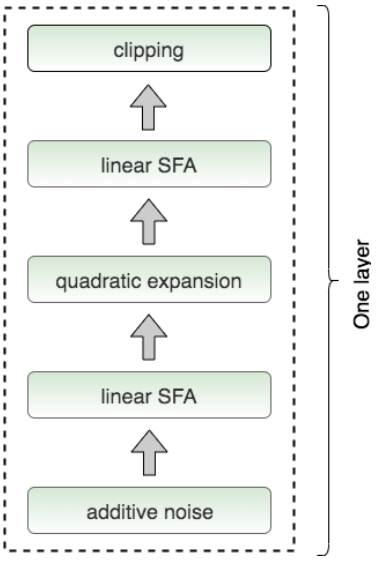

Figure 10: One Slow Feature Analysis Layer where the additive noise is always added on input in order to avoid possible singularities. Then a linear SFA is performed, after that quadratic SFA and once again linear SFA. Clipping is done to the last calculated SFA in the layer, in this case from -4 to 4. The same structure is used for all layers in hierarchical network.

this work, where the image was used for input but in this case reduced to every other pixel, so instead of 160x160 pixels, we have 80x80 pixels. The hierarchical network is similar but with 3 layers, since the input is smaller. Accordingly, output is a single array with 64 elements.

Every layer has the same model architecture which is shown in Figure 10 and it is also used in this work.

4.2 Unsupervised State Representation Learning in Atari

In [5] a new representation learning technique that maximizes mutual information about representations across spatial and temporal axes is introduced. Moreover, a new state benchmark learning representation for the state based on the Atari 2600 game package is proposed to highlight learning more generative factors.

It is shown that the proposed method excellently captures the underlying latent state factors even for small objects or when there are a large number of objects that are difficult for generative or other contrastive techniques. Their proposed benchmark can be used to examine the qualitative and quantitative differences between representation learning techniques and hopefully encourage more research into state representation learning [5].

Three main phases of the work are shown:

1. proposing a new technique of self-controlled learning for state representation that uses the spatial-temporal nature of visual observations in strengthening the learning environment;

2. proposing a new state representation learning benchmark with 22 Atari 2600 games based on the Arcade Learning Environment (ALE);

3. making extensive evaluations of existing teaching techniques for representation based on the proposed benchmark and compare them with the proposed method;

4.3 Combined Reinforcement Learning via Abstract Representations

Two main approaches to learn how to perform sequential decision-making tasks from experience in reinforcement learning are model-based and model-free approaches. The first one, model-based, has as a main task to learn a model of the environment, to build its own planning algorithm and to choose appropriate action for every time step. On the other side, model-free approaches use directly a policy or an action-value function. Which one is better to use, depends on the complexity of the task. Both have their advantages and disadvantages, what leads to the main point of this paper. What can we get if we use both of them is the main question in [6]. Their combination in Reinforcement Learning is

referred to as CRAR (Combined Reinforcement via Abstract Representations).

This type of learning has the following advantages [6]:

1. features in the abstract state provide a good generalization, since they are used for model-based and model-free approaches;
2. computationally efficient planning within the model-based module is enabled since planning is done over the abstract state space;
3. it facilitates interpretation of the decisions taken by the agent by expressing dynamics and rewards over the abstract state;
4. it allows developing new exploration strategies based on this low-dimensional representation of the environment;

Experiments are done on the two games: the Labyrinth task and the Catcher, where the Catcher is the similar to our "Atari Pong" game.

The main idea of building the abstract representation is to reduce the dimensionality of input features, whose crucial information are integrated in the new environment representation. Lower input means directly lower computation time and faster performance of the well learned model. The most important advantage here is to allow improved generalization. According to that, in low-dimensional abstract state space both model-based and model-free components can be combined, what leads to better solutions of the problem.

5 Experiments

In this section we describe practical examples which are done in Python3. For every example the MDP (Modular Toolkit for Data Processing) library is used in order to get SFA (Slow Feature Analysis) features. Afterwards, the feature extraction model is saved and used for further tests.

5.1 Simple example of Q-learning table usage

As explained before, the objective of Q-learning is to find an algorithm which can predict the right agent's action, which should be taken, in given environment state, using the policy. As an example let us take a simple navigation game. To understand all features used by Q-table, this game will be represented as a

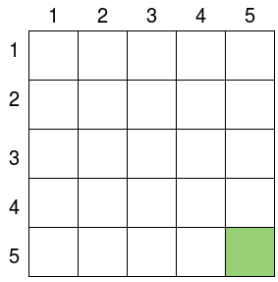

Figure 11: Simple navigation problem with 5 rows and 5 columns where the goal is a lower right corner, colored green.

5×5 board. The point in the lower right corner, colored green in Figure 11, is the goal position which the agent should find. The agent's start position can be every random position on the board, except the goal point, (5,5). The game board is shown in Figure 11.

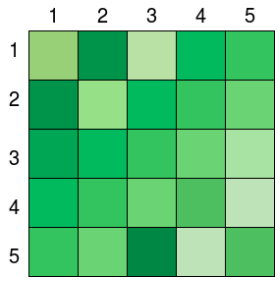

Figure 12: Q-table is initialized with arbitrary small values.

Since the agent's initial state can be every random position, the agent can use next four actions: go up, go down, go left and go right. Then he needs to

learn that in every state he should go down or right, depending on the border lines, to reach the goal. At start, the Q-table should be initialized with zero values or arbitrary small random values, as is shown in Figure 12.

Then for every state according to the policy, some of four possible actions should be executed and the Q-table will be updated with the reward from the new state.

How that actually looks is shown in Figure 13, where the agent is on position (2,3) and according to his action choice, he will get appropriate reward. If he goes up or turn left, he gets reward -1 because he moved in the wrong direction. In the case of turning right and going down, he gets positive reward +1. Through every played game the agent should update values of the Q-table,

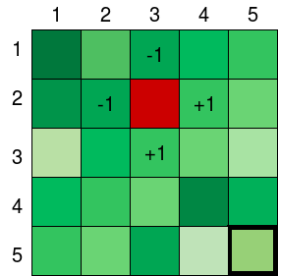

Figure 13: Agent's choice rewards according to the specific state where actions up and left bring negative rewards and down and left positives.

where after some time (learning through epochs) he obtains a perfect learned model, as it can be seen in Figure 14. Learned Q-table is shown in Table 1,

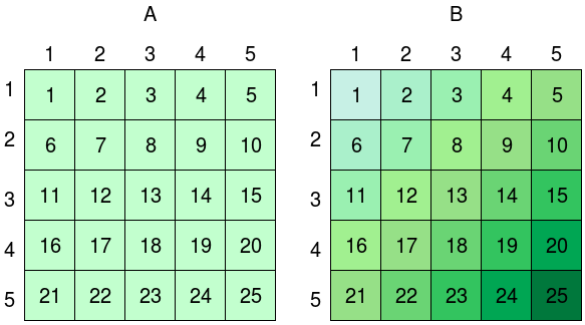

Figure 14: Initialized (A) and well learned model (B) where movement down and to the right side will be positive rewarded.

where the relationship between states and actions can is represented. Agent can be positioned on 25 different states and use four different actions, which leads to our representation of Q-table data, which size is from 5×5 converted to size

of 25×5. On this way it is easier to understand what every state represents and which action is the best choice for our agent.

state/action	$\uparrow$	$\downarrow$	$\rightarrow$	$\leftarrow$
1	0	0.2	0.2	0
2	0	0.3	0.3	0.1
3	0	0.4	0.4	0.2
4	0	0.5	0.5	0.3
5	0	0.6	0	0.4
6	0.1	0.3	0.3	0
7	0.2	0.4	0.4	0.2
8	0.3	0.5	0.5	0.3
9	0.4	0.6	0.6	0.4
10	0.5	0.7	0	0.5
11	0.2	0.4	0.4	0
12	0.3	0.5	0.5	0.3
13	0.4	0.6	0.6	0.4
14	0.5	0.7	0.7	0.5
15	0.6	0.8	0	0.6
16	0.3	0.5	0.5	0
17	0.4	0.6	0.6	0.4
18	0.5	0.7	0.7	0.5
19	0.6	0.8	0.8	0.6
20	0.7	0.9	0	0.7
21	0.4	0	0.6	0
22	0.5	0	0.7	0.5
23	0.6	0	0.8	0.6
24	0.7	0	0.9	0.7
25	0.8	0	0	0.8

Table 1: Representation of Q-table data for simple navigation problem, where actions up, down, left and right will be chosen according to greater Q-value.

To understand the values from table, we can say that lower number means more distance from the goal. According to that our agent chooses the greater Q-values.

Let us assume that agent's start position is (1,1). Using data from the table 1 he should be able to find, how to reach the goal (position 5,5). His trajectories are shown in Figure 15.

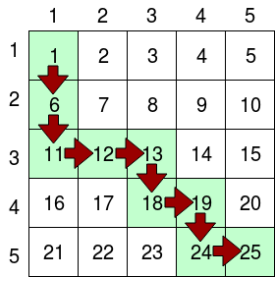

Figure 15: Simple navigation problem trajectories, where number in cells represent states. For example, state 14 is the position $(3, 4)$ on the game board.

5.2 Simple Navigation Game

This game is designed as a test example where every RL and SFA algorithm has been tested before on an actual Atari game. This is accomplished due to the complexity of Atari games which have a huge number of possible states. This game is rather simple, but it contains different goal positions where the agent should learn how to behave in four different situations. So, on $B \times B$

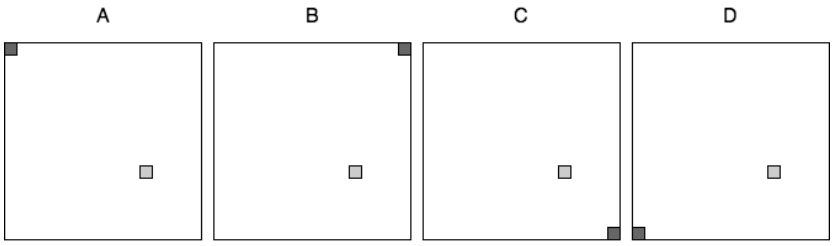

Figure 16: Simple Navigation Game Representation where A, B, C and D represent the four possible goal states (left corner above, right corner above, right corner below and left corner below, respectively).

pixel board the two dots are marked with different colors and represent our game (for B, board size, we used different numbers: 10, 20, 40 and 80, where the last one should be test used for a Pong game because of its dimensionality). The agent's task is finding a path to the goal in each game, which can be in any of four possible corners. With each new start of the game, the agent's position is random, as is the goal which can always be in a different random corner.

When an agent finds a path from his starting position to the goal, he gets a reward of +10. In the case that he does not find the solution during 1000 attempts, he loses the game and receives a reward of -1. For all other steps and updates a reward of 0 is used.

The goal of this experiment is to test the SFA architecture, Q-learning algorithm and Reinforcement Learning on SFA.

Here we will present the 5 trained models. Every model has different input features (same input information in different shape):

1. RL with Q-table directly on images (10×10 size) — first model
2. RL (Q-learning with linear function approximation) directly on images (10×10 size, 20×20 size, 40×40 size, 80×80 size) — second model
3. RL (Q-learning with linear function approximation) on coordinates (x, y, bias) — third model
4. State Variables from SFA — fourth model
5. RL (Q-learning with linear function approximation) on SFA features (64,bias) — fifth model

One epoch is given by one game of max. 1000 steps. That means, if the goal is found before 1000 steps, the agent wins, otherwise he loses the game.

5.2.1 Reinforcement learning on images with Q-table

In this experiment we use the whole image information as an input for the training, where 10×10 frame size is used.

We used online Q-learning with a greedy policy, which mathematical representation is shown in the equation (15).

Metaparameter type	Metaparameter value
learning rate	0.5
gamma (discount factor)	0.1
start epsilon (for greedy policy)	1
discount epsilon (for greedy policy)	0.99
epochs	100
number of actions	4

Table 2: Hyperparameters for Q-table

Every input feature is converted to an array of 100 elements which represents the state and is then normalized between 0 and 1.

Example of one state: $s = [1, 0, 0, ..., 0, 0.5, 0, ...0]$

The Q-table is a matrix with the size: number of possible states $\times$ number

of possible actions. Furthermore, the number of possible states can be calculated as 4 possible goals multiplied by 99 different states, which leads to size 396. The game has 4 possible actions: up, down, left and right. According to that we have a Q-table of size 4×396. Then we implemented mapping between all states (size of 100×1) and indices in the Q-table and trained the algorithm a 20 times over 100 epochs, where every epoch is given by exactly one game. Results of 20 simulations are shown in Figure 17 through their mean value, together with standard deviation.

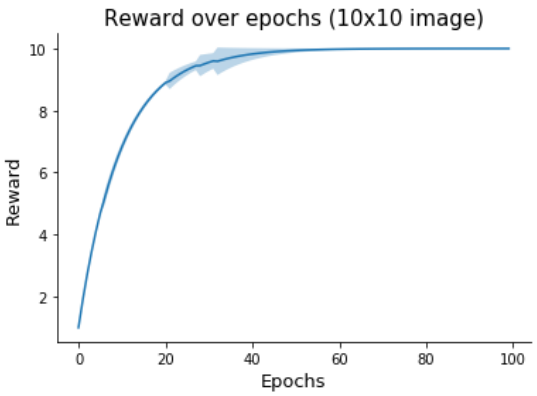

Figure 17: Navigation problem, where the whole normalized image is used as input for a frame size 10×10.

The resulting reward over epochs is shown as a smoothed reward, whose definition is shown in the equation (19),

$$r_{smoothed} = 0.90 r_{smoothed} + 0.10 r \tag{19}$$

where r is reward of one game and $r_{smoothed}$ smoothed reward.

As it can be seen in Figure 17, our Q-table is stable after 50 epochs/number of games.

5.2.2 Reinforcement learning on images with linear function approximation

Here the whole image information is used as an input for the training, where for input four different frame sizes are simulated:

1. 10×10 pixels;
2. 20×20 pixels;
3. 40×40 pixels;

4. 80×80 pixels;

Afterwards the whole image for every frame size is represented as a vector of size B^2 for an image size of $B \times B$, where B is frame/image size. According to that, the input of 10×10 image is converted to a vector of size 10. Input values are normalized between 0 and 1.

In this experiment we used online Q-learning with with linear function approximation and ϵ-greedy policy, which mathematical representation is shown in the equation (16). A major limitation of Q-learning is that it only works in environments with discrete and finite state and action spaces.

Metaparameter type	10×10	20×20	40×40	80×80
learning rate	0.05	0.005	0.0005	0.00005
gamma (discount factor)	0.1	0.1	0.1	0.1
start epsilon (for greedy policy)	1	1	1	1
discount epsilon (for greedy policy)	0.99	0.99	0.99	0.99
epochs	200	800	3200	12000
number of actions	5	5	5	5

Table 3: Hyperparameters for Q-learning with linear function approximation

For all four cases we used metaparameters shown in Table 3, where every input feature corresponds to its input frame size converted in 1-size array. According to that, in first case for 10×10 pixels we have a array with (100, 1) size.

Actions are up, down, left, right and do nothing.

The resulting rewards over epochs are shown as a smoothed reward, whose definition is shown in the equation (20),

$$r_{smoothed} = 0.99 r_{smoothed} + 0.01 r \tag{20}$$

where r is reward of one game and $r_{smoothed}$ smoothed reward.

Results are shown in Figure 18 where we can see that the algorithm learned faster for the lower frame size, which is expected. We can notice that every model is well learned after ca. B^2 number of epochs.

5.2.3 Reinforcement learning on coordinates with linear function approximation

The RL method used here is also online Q-learning with linear function approximation and ϵ-greedy policy, which mathematical representation is shown in the equation (16). From the $B \times B$ pixels boards the main two points are extracted, where B is the number of board size. Those are x and y positions of the agent

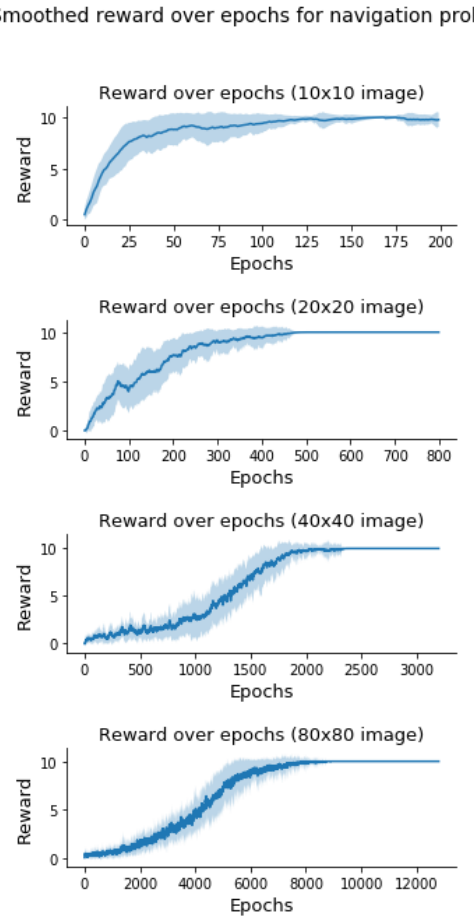

Figure 18: Navigation problem, where whole normalized images were used as input. Smoothed reward over epochs for Q-learning with linear function approximation. Results for images of four different sizes are shown: 10×10, 20×20, 40×40 and 80×80.

and goal on the board. Then we added one additional element, a bias with the fixed value of 1. That leads to vector of 5 elements. An example is shown in Table 4, where Q values are shown for the two states shown in Figure 19.

The possible combinations of x and y coordinate-positions are between 0 and $B - 1$, including 0 and $B - 1$. For the board size we used 10, 20, 40 and 80. Then coordinate values are normalized to the range between 0 and 1. So now we have an input array with five elements: normalized x coordinate of agent, normalized y coordinate of agent, normalized x coordinate of goal, normalized

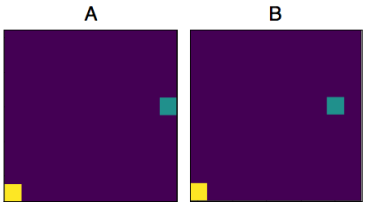

Figure 19: Navigation problem, two example states for a frame size 10×10

action	Q value for A state	Q value for B state
up	5.99728397e-15	6.23995729e-14
right	5.43107193e-15	7.29199640e-14
down	2.47018428e-16	5.20844631e-16
left	3.05524694e-17	5.63622183e-16

Table 4: Q values of two example states from Figure 19

y coordinate of goal and bias which is always 1.

We did some simulations with 12.500 training iterations. One iteration cor-

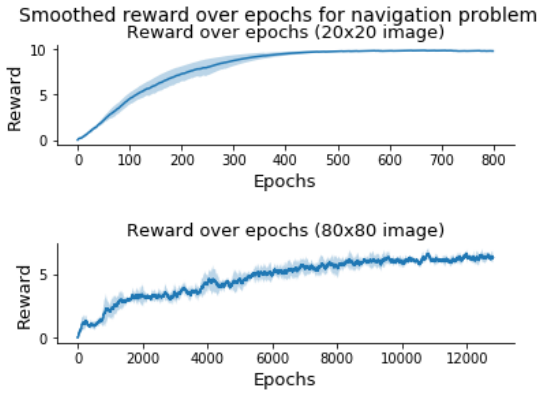

Figure 20: Navigation problem, where normalized image coordinates were used as input. Smoothed reward over epochs for Q-learning with linear function approximation. Results for images of two different sizes are shown: 20×20 and 80×80.

responds to one game, where game corresponds to maximal 1.000 steps. All training parameters, except input images, are the same as in the previous section (Table 3).

After 12.000 episodes/iterations the well prediction-model is created. We can

see it in Figure 20 where we plotted mean values of 20 experiments per frame size.

To better understand the points in every single game, we made a plot of 30 test examples where for every game we saved the number of minimum possible steps to reach the goal and number of steps which are done according to predicted trajectories. In the following text, predicted steps will represent the number of steps which are done by prediction trajectories, in this case our learned model. So, now we can compare information and find out how well our model is learned. See Figure 21. When the predicted number of steps is around 200

Figure 21: Ideals vs. number of steps used by agent over episodes. According to start agent position the smallest distance is calculated(calculated by Manhattan distance) between the agent and a goal. Then this distance is compared with the number of steps which are done per current epoch.

- 300, as it is shown in Figure 21, then we can say that our model is well trained.

Trajectories
Two different test examples are shown in Figure 22 where the agent's predicted steps/actions are demonstrated. The green box is agent's goal and the blue box is his random start position. The learned model based on Q-learning is tested on two games where every agent's movement is plotted in Figure 22. When the agent visits certain positions only once, then the path is colored by a light grey color. If he visits position two times, the color is grey. The last case in the figure is dark gray, which demonstrates position visiting of three times (for example he goes back and then again finds a right path or he stays for three times at the same position).

5.2.4 State Variables from SFA

In this section we describe SFA behavior on two different input image sizes: 20×20 and 80×80.

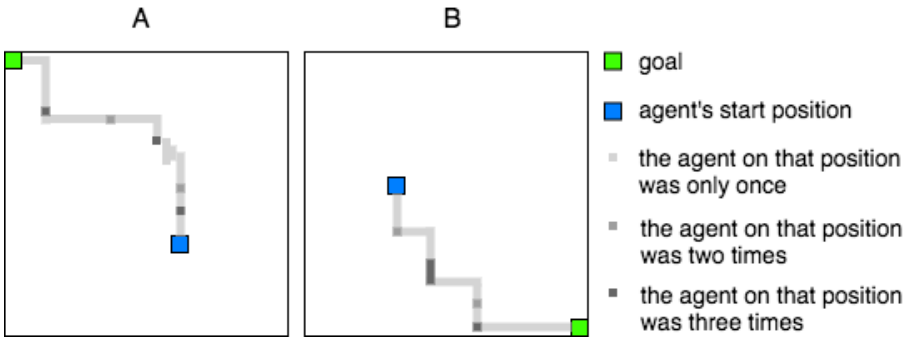

Figure 22: Trajectories for two test games, A and B. For the game A is goal at the left upper corner on the board and for game B the goal is at the right lower corner. For the both games the agent finds the goal and makes small number of errors (wrong path predictions).

Input Image 20x20

Layer	Number of nodes	Field size	Overlap	SFA output per node
0	20×20	-	-	pixel
1	6×6	10×10	2	32
3	1	6×6	-	32

Table 5: Representation of SFA network architecture for simple navigation problem where the input image is of size 20×20.

In Table 5 we show the SFA network through its layers. The first layer consists of the original image as an input, which has size is 20×20 pixels. In the second layer we use 6×6 number of nodes with overlap 2, which leads to a field size of 10×10. The SFA nodes output 32 slow features. The last layer contains 6×6 field size and 64 SFA outputs per node.

In our SFA implementation, every layer has sub-nodes that are: adding some noise, performing linear SFA, quadratic expansion and again linear SFA. Clipping is not used in this experiment.

For SFA training we used 30.000 time steps, where the input images 20×20 are converted to 400×1 size. This conversion leads to 30.000×400 as an input for the first SFA layer. The last layer gave us the final results from the SFA algorithm where the output has a size 30.000×32.

An example of SFA features is shown in Figure 23.

To verify the correctness of SFA features linear regression was used, where

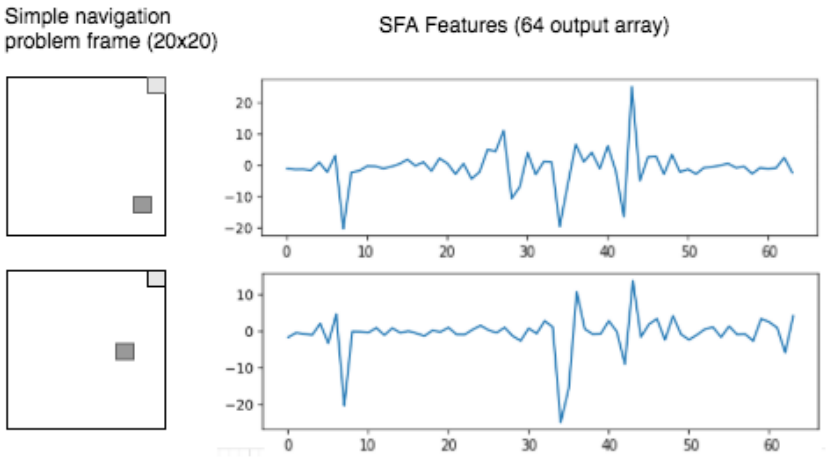

Figure 23: SFA features vs. original 20×20 images

97% of SFA data were used as training data and the rest, 3%, we later used for testing.

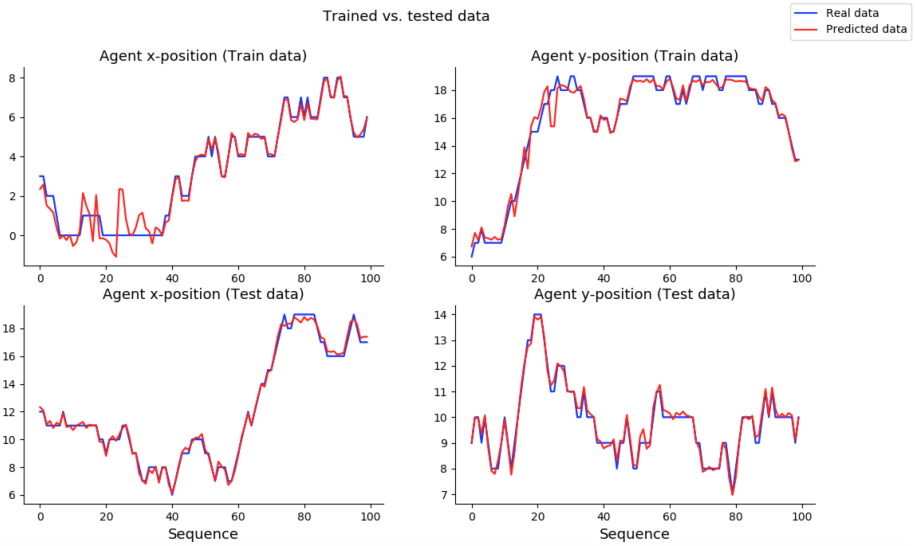

Figure 24: Trained vs. tested data of SFA features performed on linear regression where the input image size was 20×20.

Results are represented through graphs shown in Figure 24, where we can see that SFA features well represent important information of original images.

Input Image 80×80

Layer	Number of nodes	Field size	Overlap	SFA output per node
0	80×80	-	-	pixel
1	14×14	15×15	5	32
2	5×5	6×6	2	32
3	1	5×5	-	64

Table 6: Representation of SFA network architecture for simple navigation problem for an input image of 80×80 size

In this case we did not use any Q-learning method. Goal of this experiment is to calculate SFA features which can be later used as an input for different Q-learning algorithms.

Here we used 15000 frames of simple navigation problem game, which are converted to 6400×1 size which leads to a 15000×6400 matrix for SFA input, whose network architecture is shown in Table 6. On all three SFA layers five sub-actions are made. First some noise is added, then linear SFA is performed, after that quadratic expansion and again linear SFA and at the end the output was clipped to the range -4 to 4.

Two random images are used and their representation of SFA features are shown in Figure 25.

As it can be seen in Figure 25, it is really hard to understand SFA features and agent's positions on this 64×1 output array. So, to be sure that the SFA model is well learned, we used here also linear regression for checking the results. From python sklearn library linear-model function is used for linear regression implementation and training.

The agent's x and y coordinates are detected on trained and tested SFA feature data and they are shown in Figure 26.

5.2.5 Reinforcement Learning performed on Slow Features with linear function approximation

In this section we describe Reinforcement Learning performed on SFA features, which were already learned and described in the previous experiment and section. Input images are 20×20 pixel images which are converted to size 400×1 and used as an input for SFA. As SFA outputs we get 64×1 arrays. This values are clipped to the range -4 and 4 and normalized between 0 and 1. The 64×1 representation of image together with bias value 1 are used as an input for RL. According to that, input feature size is 65. As an RL algorithm Q-Learning is

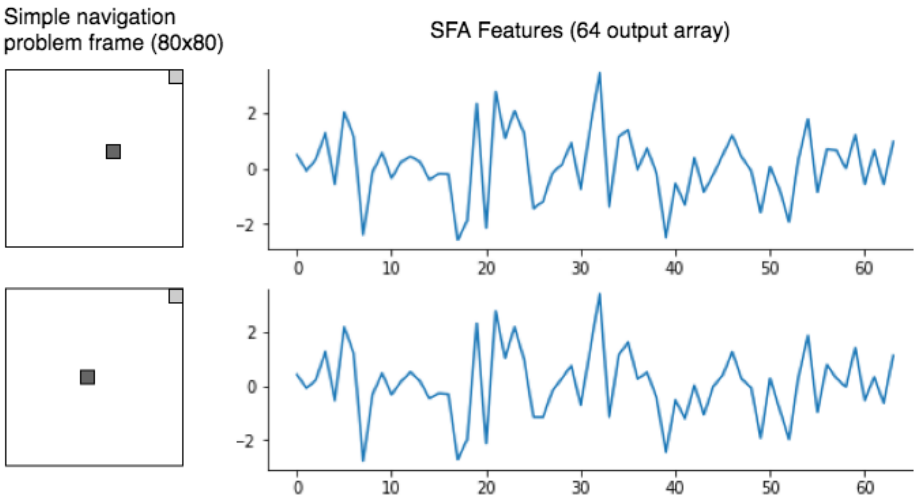

Figure 25: Linear Regression on SFA features performed ond simple navigation game. Agent's x and y coordinates, as well as goal position are coded in SFA features represented as 64 output array

.

Figure 26: Linear Regression on SFA features performed on simple navigation game. Agent's x and y coordinates are trained and tested.

used, described in Section 3.2 (equation (16)).

Q-learning is trained online, which means, after every game step the weights are updated. We used ϵ-greedy policy with a start value 1 for ϵ which was multiplied by 0.99 in every next step. This value should decrease over epochs until it reaches the value 0.1. For γ 0.1 was used and learning rate α in this experiment is 0.0001. One epoch consists of one game. Every game has a maximum of 1000 steps, which the agent can reach only in case of a loss. Otherwise an agent wins.

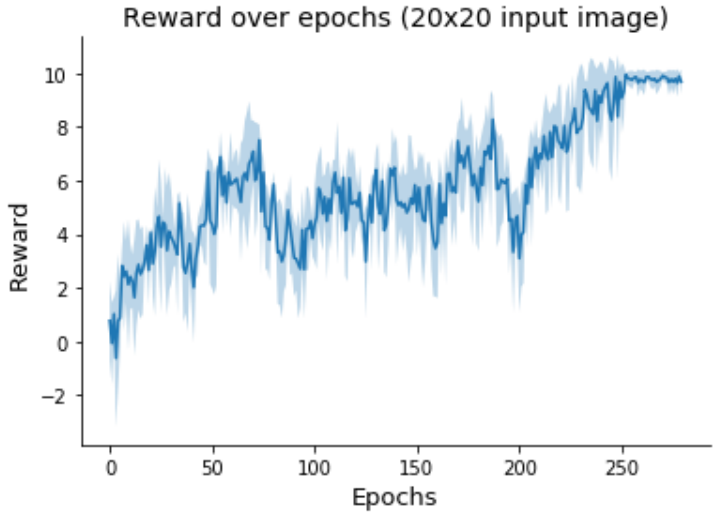

Figure 27: Simple navigation game, RL performed on SFA where 65×1 size is used as an input.

Using the described parameters we trained Q-learning on 280 different games, where after 250 epochs our model is well learned. Then the model is starting to have a stable behaviour where every next game finished as a success for the agent.

Figure 27 shows the smoothed reward over epochs for the 20×20 size game. Smoothed reward is described in Section 5.2.1.

5.2.6 Comparison

In this section we will compare the results between Reinforcement Learning on images, Reinforcement Learning on coordinates and Reinforcement Learning performed on Slow Features. Also it is important to mention which algorithm is the fastest one, which metaparameters we used, what is the advantage of different input features and so on.

For the comparison we use the 20×20 frame size version of the simple navigation game.

According to the given results from the sections 5.2.1, 5.2.2 and 5.2.4, we can conclude, that reinforcement learning on images and reinforcement learning on coordinates give us similar results. With the same learning parameters (exclusive input features), the models are well trained after ca. 480 epochs. The difference, which we can notice, is, that by reinforcement learning on coordinates, the model is a little bit unstable in the first 400 epochs, which is not the case with the first algorithm, where the smoothed reward exponential by increases over epochs until the model become stable. Comparing the results from reinforcement learning performed on Slow Features, it is clear that there is a stable model approached after 250 epochs/games, which leads to the fact that it is the fastest algorithm in sense of the number of epochs, ie. game steps.

If we consider the execution time of one epoch of 1000 steps, then we have a completely different situation. Here we have 0.14, 0.2 and 7.5 seconds, respectively.

5.3 Pong Game

Pong, released by Atari in 1972, became the world's first popular video game and first became known in the 1970s on gadgets. It is considered the forefather of video games, although video games had been developed before. The

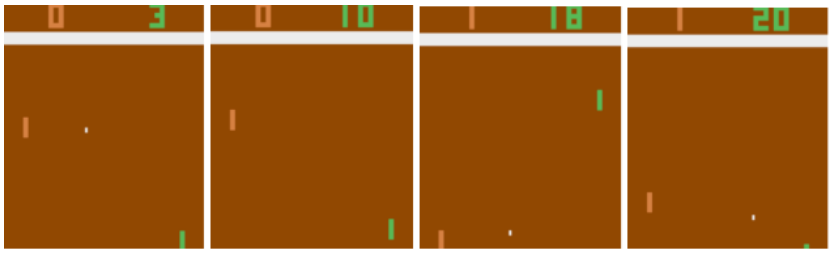

Figure 28: Atari Framework examples of Pong game from Python gym library where the four different frames are shown.

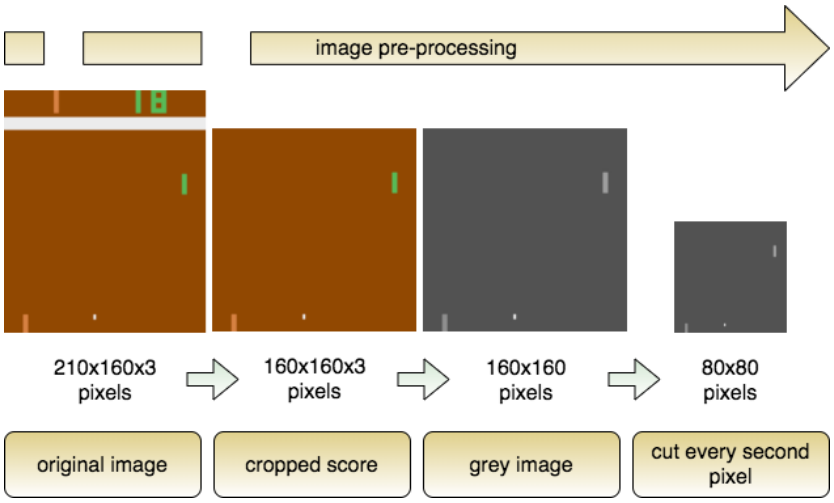

Figure 29: Pre-processing on Pong Image Frame where on the RGB 210x160x3 pixel image three operations are performed: score cropping, converting to grey image and cutting every second pixel, respectively.

game-play of Pong is simple and similar to table tennis: A point, in this case a "ball", moves left and right on the screen. Each of the two players controls are represented as a vertical lines and can move with a rotary knob (paddle) up and down. To win, one of them needs to reach 21 points.

The Pong Game is the part of python3 gym library where the whole environment can be easily controlled.

Some examples of game frames are shown in Figure 28, where every frame has 210x160x3 pixels. In this work image pre-processing is done, which is used as an input for SFA and it is shown in Figure 29. The main idea behind the image pre-processing is to decrease the dimensionality of the input images, which leads to faster SFA calculation. Then the calculated 80x80 grey image is used as an input for all SFA algorithms in the experiments with the Pong Game.

5.3.1 Reinforcement learning on input images

In this experiment we used Pong image frames as an input. Here is some kind of pre-processing done. First, we convert all pixel values from RGB to Gray image. Then all values greater a 0 are assumed as 1. In this case we have only 1 and 0 as frame values. Then we crop image score which leads to image size 160×160. To reduce the dimensionality we crop every second pixel. According to that, we have 80×80 input images which consist only of values 0 and 1. Since for input is 1D array necessary, we convert the 80×80 matrix to 1600×1 array.

As input features we used 1600×1 array frame representations, but they are used as a subtraction between previous and current game step. Only in this way it is possible to find out if the ball goes up or down. For the first time step we always used 1600×1 zero array.

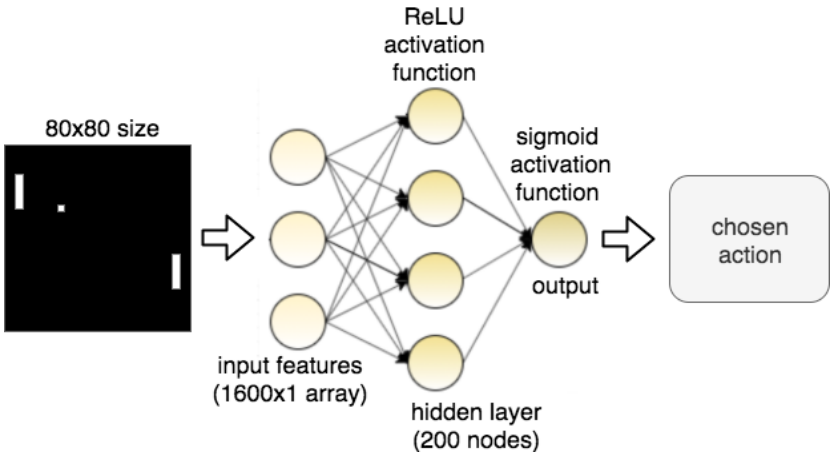

Figure 30: Deep Q-learning network which consists of an input layer, one hidden layer and an output layer. Input layer represents 1600×1 array representation of game frames. Hidden layer has 200 nodes with ReLU activation function. Output layer uses a sigmoid activation function.

The Reinforcement Learning algorithm used here was batch-constrained Deep Q-learning, described in section 3.3. This means, after some number of iterations/game steps the weights will be updated. For batch size we use 32. The

Deep Q-learning network is shown in Figure 30.

The Hidden layers capture more and more complexity with every layer by discovering relationships between features in the input. The Hidden layer, used in this experiment, consists of 200 nodes which are applied with ReLU activation functions to produce the output, which is an input for output layer. ReLU is linear (identity) for all positive values, and zero for all negative values. At the end the output layer uses the sigmoid as an activation function. As a result the appropriate agent's action should be chosen.

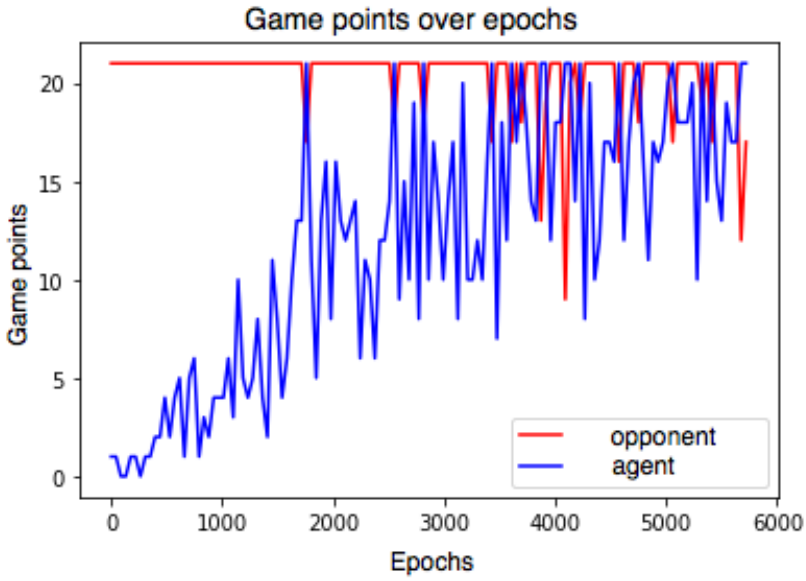

Figure 31: Game points of left and right controller during the learning process, where our agent (right controller) starts to win after 1900 epochs.

The agent is trained over 6000 epochs, where every epoch represents exactly one game. How many steps the game has, depends on the agent's behaviour. For example, when the agent loses the game with 0 points, that leads to around 1000 steps. In case of a victory, more then 3000 steps will be executed. In Figure 31 reward is shown over epoches, where we can see that with the number of epoches our agent obtains more points. After 1900 epoches he starts to win and reaches 21 points.

5.3.2 State Variables from SFA

Slow Feature Analysis is performed on the Atari 2600 game Pong, where preprocessed images are used as an input to the SFA. An example is represented in Figure 32.

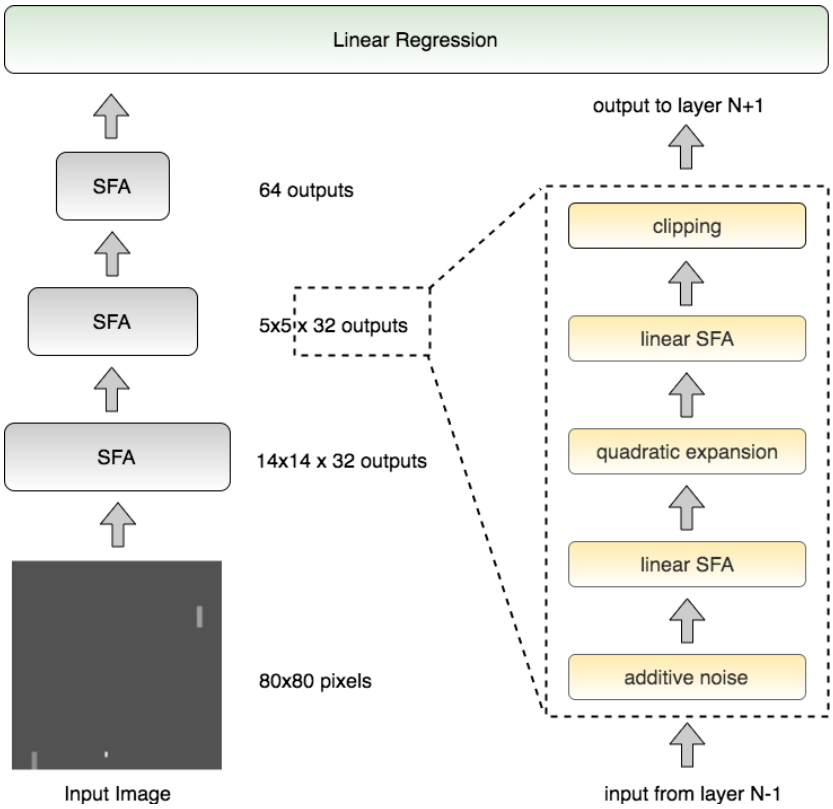

Figure 32: Model architecture of SFA model performed on Pong grey image as an input. The hierarchical network has three layers where every layer has parameters like field overlapping, filed size and nodes.

To better understand the network structure, all nodes information are shown in Table 7. **Layer 0:** In this test example, 80×80 array is used as an input, where all points are only the numbers, which represent gray color, between 0 and 127, which are scaled to range from 0 to 1. **Layer 1:** On 6400 length array, SFA is performed where field size of 15×15 with field overlap 5 is used as an input. As a result, 32 outputs with the size 14×14 are calculated. **Layer 2:** On all calculated 14×14 inputs, field size of 6×6 with overlap 2 is performed, which leads to 5×5 nodes. The number of outputs per node here, is also 32. **Layer 3:** On the last layer, input of 5×5 nodes is converted to array of the length 25, on which the result array of length 64 of a single node is calculated.

Layer	Number of nodes	Field size	Overlap	SFA output per node
0	80×80	-	-	pixel
1	14×14	15×15	5	32
2	5×5	6×6	2	32
3	15×5	-	64	

Table 7: Representation of network architecture

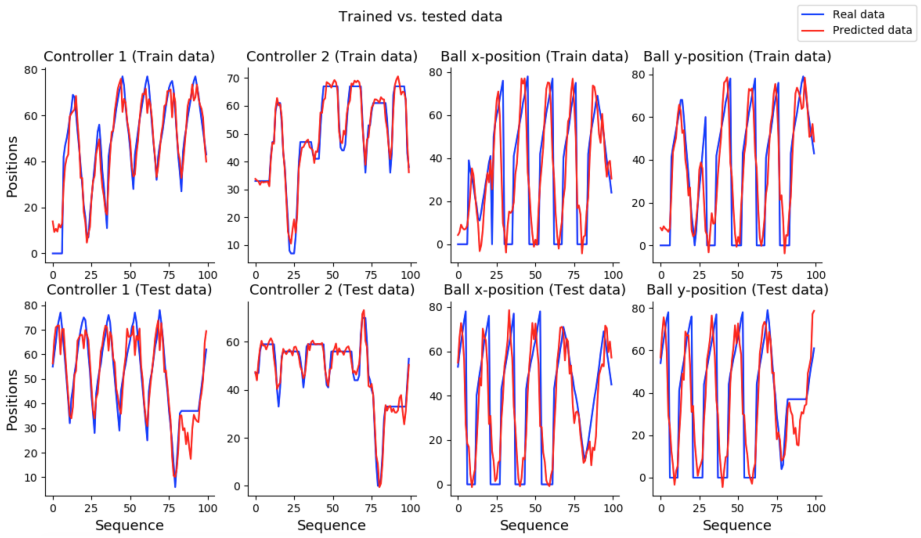

Figure 33: Trained vs. tested data of linear regression performed on SFA features. Random 100 sequence samples for trained and tested data are shown. SFA model is well learned as can be seen by coordinate predictions.

10000 frames (ca. 8.5 games) are used as an input for Slow Feature Analysis. As explained in Figure 29, on every input image the pre-processing is done, what leads to size of 10000×6400 as an input for training.

Every SFA layer has the five actions or substeps:

1. add noise;
2. perform linear SFA;
3. perform quadratic SFA;
4. perform linear SFA;
5. clipping from -4 to 4;

Then, the results from the last layer are in the range between -4 to 4. Because of easier calculation, calculated Slow Features are normalized to a range between

0 and 1, which are used later for linear regression.

Two examples of SFA features from this trained data are shown in Figure 34.

Figure 34: Two SFA feature vectors together with their Pong images are represented as a grey-scale, where 64 elements are used as a size of SFA demonstration. This way, every game frame is re-scaled from the size 80×80 to 64×1.

Why is linear regression used?

Main idea here is to find a way to check and verify if the calculated Slow Features really contain the important position information of the ball and both controllers. Accordingly, for every controller only its y position is relevant, since x is always a constant. For the ball are important the both, x and y, positions.

First, the model of size 10000×64 for Slow features is created together with the information of its real ball and controller positions. Then, it is trained on 96% data, in this case 96000, where the rest of 4% is used as a test set, or more precisely, 400 samples.

Tested and trained data are shown in Figure 33, where it is clear, that all positions are well detected and decoded. Now, those features can be used as an input for Reinforcement Learning, Q-Learning.

5.3.3 Reinforcement Learning performed on Slow Features

The Pong Game is a little bit complex, because it can have a lot of different states. For instance, moving from one to another controller, the ball should be

updated ca. 50 times, what means, 50 states for only one ball path. There is also a problem with the usage of the current state without any information about the previous one, where agent can't know if the ball should go up or down.

In this experiment, we merged two states, past and present, into one vector, to make it easier to recognize agent movements. For example, in the previous section, 5.3.1, we used pre-processed images as input, where the images were represented as a vector of zeros and ones, we always subtracted the vectors of the current and previous images. Thus, as an input, we obtained 1×1600 vector size, which has values of -1, 0 and 1.

Here we approach the problem in a different way. We first pre-processed each image as described in section 5.3.3. Then we applied the learned SFA model from the previous chapter to each image, size 1×1600. Here we got the size 1×64 as the output. After each layer, except the last one, we did a cropping between -4 and 4. After that, we obtained the SFA features rounded to four decimal places. In order for the model to distinguish between the state of the ball leading up or down, we always merged the previous and current state into one array, and added a bias with a value of 1. This leads to 1×129 input features: current SFA state (1×64), old SFA state (1×64) and bias (1×1).

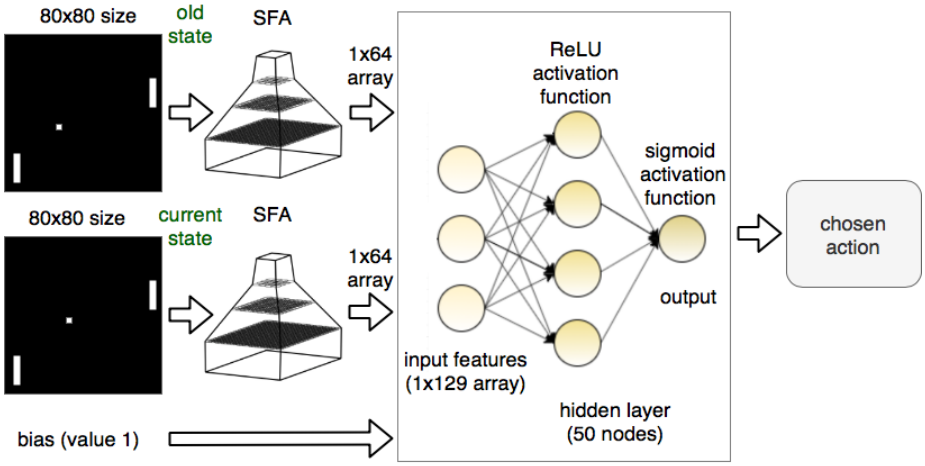

Figure 35: Reinforcement learning performed on SFA where the whole network architecture, together with input and output feature is shown.

The reinforcement learning algorithm used here was deep Q-learning, described in section 3.3. This means, after some number of iterations / game steps the weights will be updated. In this experiment, we update the weights after each epoch. The epoch, as in all experiments of this work, represents one game. In the neural network we used one hidden layer with 50 nodes with ReLU activa-

tion functions. For the output layer we used the logistic sigmoid as an activation function. Our neural network can be seen in Figure 35.

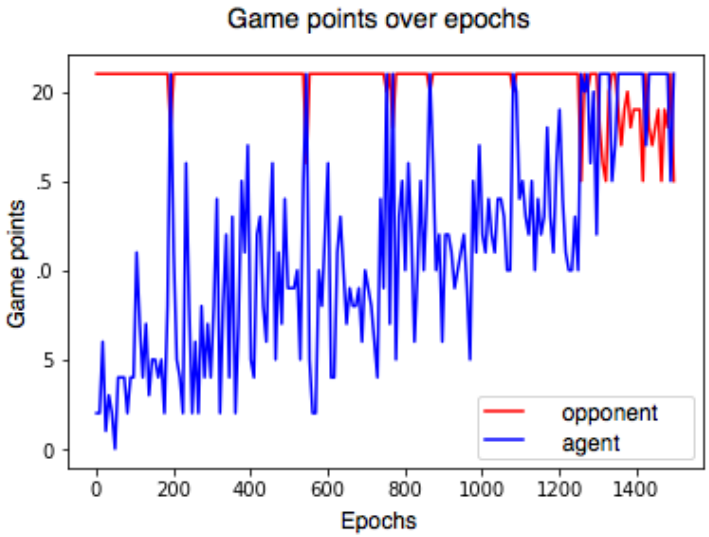

Figure 36: Game points of left and right controller during the learning process on SFA features, where our agent starts to win after 200 epochs. Every eighth game score is plotted for easier display of results.

The learning rate used here is 0.005 and the discount factor 0.99. We trained the model for 1500 epochs. This algorithm shows a significantly improved and accelerated way of learning, where after the 200 epochs our agent starts winning the game. Figure 36 shows the results in terms of game points of our agent and the opponent.

When we compare our results with [1], where the model is well trained with 10 million frames, we can say that we got the similar results. Our average epoch consists of 5000 frames, which leads to 7.5 millions trained frames after 1500 epochs.

5.3.4 Comparison

When we compare the usage of the same algorithm, Deep Q-learning on input images and reinforcement learning performed on slow features, we can clearly say that a significant improvement in agent learning is seen when using SFA features. In the first algorithm, the agent won for the first time after 1900 epochs/number of games. In the second experiment, the agent won for the first time after 200 epochs/number of games, hence, the learning speed increased by 9.5 times.

6 Conclusion and Outlook

In this work, we present a Deep Q-Learning as RL algorithm that uses SFA features as data input, where input data have significantly reduced dimensions, leading to a drastically reduced number of epochs required to obtain a well-trained model on the Atari 2600 Pong game. This leads us to the conclusion that the representation of input images, which contains all important information about it, can reduce the duration of learning. In this way, we get desired results faster, and we can consider SFA as a potential solution for image representation of the all more complex video games.

The aim of this work was to prove that different input representations have a different impact on the same RL algorithm, which is shown in Section 4. The work represents the first version of the software environment that offers game selection options, where we currently have Pong Atari 2600 and our Simple Navigation Game. It is then possible to choose what type of image inputs we want to use. In case of SFA features, we can use a dynamic GUI made primarily for modeling the SFA network and learning it. Each image representation can be tested using linear regression. The type of RL algorithm in the current version is always Deep Q-Learning with one hidden layer. The next version should be based on the improvement of already existing functions. For example, in addition to image representation algorithms such as standard pre-processing and SFA features, we can add PCA and auto-encoders. By adding various new games, we can accordingly add new ways of learning. Likewise, research related to Deep Q-learning and experience replay can be added for RL algorithms.

References

[1] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. NIPS Deep Learning Workshop 2013, arXiv:1312.5602, 2013.

[2] Lukasz Kaiser et al. *Model-Based Reinforcement Learning for Atari*. ICLR 2020, 2020. arXiv: 1903.00374 [cs.LG].

[3] Russell Kaplan, Christopher Sauer, and Alexander Sosa. *Beating Atari with Natural Language Guided Reinforcement Learning*. CoRR, 2017. arXiv: 1704.05539 [cs.AI]. URL: http://arxiv.org/abs/1704.05539.

[4] Ionel-Alexandru Hosu and Traian Rebedea. *Playing Atari Games with Deep Reinforcement Learning and Human Checkpoint Replay*. CoRR, 2016. arXiv: 1607.05077 [cs.AI].

[5] Ankesh Anand et al. *Unsupervised State Representation Learning in Atari*. Published in NeurIPS 2020, arXiv:1906.08226v5 [cs.LG], 2020.

[6] Vincent Franc-Lavet et al. *Combined Reinforcement Learning via Abstract Representations*. AAAI 2018, DOI:10.1609/ aaai. v33i01. 33013582, 2018.

[7] R Legenstein, N Wilbert, and L Wiskott. *Reinforcement Learning on Slow Features of High-Dimensional Input Streams*. PLoS Comput. Biol. 2010; 6(8): e1000894. doi:10.1371/journal.pcbi.1000894.

[8] Mathias Franzius, Niko Wilbert, and Laurenz Wiskott. *Invariant Object Recognition with Slow Feature Analysis*. In Proc. Int'l Conf. on Artificial Neural Networks, Sept. 2008, pp. 961–970. DOI: 10.1007/978-3-540-87536-9_98.

[9] Mathias Franzius, Henning Sprekeler, and Laurenz Wiskott. *Slowness and Sparseness Lead to Place, Head-Direction, and Spatial-View Cells*. Vol. 3. 8. Aug. 2007, e166. URL: http://cogprints.org/5711/.

[10] L. Wiskott et al. *Slow feature analysis*. Scholarpedia, 2011, p. 5282. DOI: 10.4249/scholarpedia.5282.

[11] Escalante-B., A.N., and L. Wiskott. *Improved graph-based SFA: information preservation complements the slowness principle*. Mach Learn 109, 999–1037, 2020. https://doi.org/10.1007/s10994-019-05860-9, 2020.

[12] T. Zito et al. *Modular toolkit for Data Processing (MDP): a Python data processing frame work*. Front. Neuroinform. (2008) 2:8. doi:10.3389/neuro., 2009.

[13] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning*. The MIT Press Cambridge, Massachusetts London, England, 2018.

[14] Mnih Volodymyr et al. *Human-level control through deep reinforcement learning*. Nature 518, 529–533, 2015. https://doi.org/10.1038/nature14236, 2015.

[15] Dave Kuhlman. *A Python Book: Beginning Python, Advanced Python, and Python Exercises*. Platypus Global Media, 2011.

[16] Greg Brockman et al. *OpenAI Gym*. CoRR, 2016. arXiv: 1606.01540. URL: http://arxiv.org/abs/1606.01540.

[17] Fabian Pedregosa et al. *Scikit-learn: Machine learning in Python*. Vol. 12. Journal of Machine Learning Research, 2011, pp. 2825–2830. URL: https://www.bibsonomy.org/bibtex/2beb36b0b9e07fb9bb86e5198faebf14a/salotz.

Appendix

A Libraries and Programs

The practical implementation of this work is based on the script language Python3 together with libraries such as gym, mdp, sklearn etc.

A.1 Python3

Python is an universal, usually interpreted as higher programming language. It aims to promote a legible, concise programming style. For example, blocks are structured not by curly brackets, but by indentations. Because of its clear and concise syntax, Python is considered easy to learn. It supports several programming paradigms, e.g. B. object-oriented, aspect-oriented and functional programming. It also offers dynamic typing. Like many dynamic languages, Python is often used as a scripting language. The language has an open, community-based development model that is supported by the non-profit Python Software Foundation [15].

This programming language is today very often used for solving the ML (machine learning) problems. It has integrated a lot of libraries, which contain ML algorithms. Also it provides easy implementation of GUI interface and easy usage of the most popular games used for the ML assumptions.

A.2 gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of our agent, and is compatible with any numerical computation library, such as TensorFlow or Theano. We can use it from Python code, and soon from other languages. [16]

In this work it is used for simulation of the Atari Pong game. For each time step this tool returns a new environment state as a RGB $210 \times 160 \times 3$ image, current score/reward and information if the game is finished/done. Given that information, the game can be simulated and RL model calculated.

A.3 MDP - Modular toolkit for Data Processing

The Modular toolkit for Data Processing (MDP) package is a library of widely used data processing algorithms, and the possibility to combine them together to form pipelines for building more complex data processing software. It is developed to be used for scientific data processing development. This library contains different algorithms for supervised and unsupervised learning, principal and independent component analysis and classification. The above mentioned algorithms are used for data processing [12].

Our task is to prepare an input data for the MDP data processing. Then we

should choose an algorithm and its metaparameters. Also our output dimensionality should be defined. Then mdp tools will train given data and give us a result. If the input data are to complex, we need to use feed-forward network architectures with a defined number of flows/levels/layers. Metaparameters as input and output dims, node type, and so on, should be defined for every node. One flow (layer) can contain one or more nodes.

The number of available algorithms is steadily increasing and includes signal processing methods (Principal Component Analysis, Independent Component Analysis, Slow Feature Analysis), manifold learning methods ([Hessian] Locally Linear Embedding), several classifiers, probabilistic methods (Factor Analysis, RBM), data pre-processing methods, and many others.

In this work the mdp library is used to get SFA features. Example of one network architecture is shown in Figure 37.

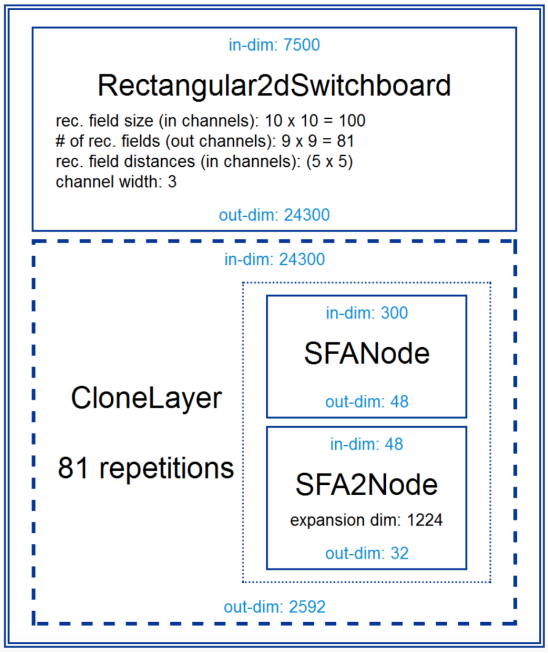

Figure 37: Layer representation of SFA, where as an input 10×10 matrix, converted to 100×1 array is used [12].

A.4 sklearn

The used python library for supervised and unsupervised learning algorithms is called sklearn. There are all mathematical implementations already integrated

and we just need to choose appropriate learning method together with its meta-parameters [17]. In this work we used default values for linear regression to get all important positions/coordinates of our games - simple navigation problem and Pong Atari 2600. SFA features of mentioned game frameworks are used as an input for linear regression to get ball and agents predicted coordinates on the board.

B GUI - Graphical User Interface

A Graphical User Interface is used red to ease experimental setups. It is really hard to define which network architecture is good enough to give us the outputs which contains all important information of input images. Also, for every training example we need to calculate which field size, step size and number of nodes could be used for given input size. To avoid all unnecessary calculations we implemented the Graphical User Interface which should give us all combinations of field and step size with calculated number of nodes.

The first step is to click on button "+" which is used to add a new layer, as is shown in Figure 38. Then the Layer 1 will be shown, as it is demon-

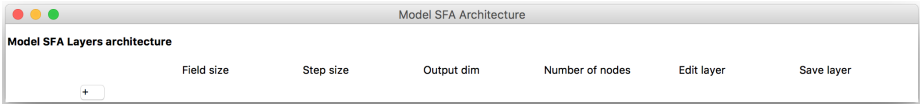

Figure 38: Model SFA Architecture, add a first layer

strated in Figure 39, with already defined field size, step size, output dim and calculated number of nodes. Since field and step size are represented through comboboxes, we can choose all possible combinations of this two parameters. It is important to mention, that choosing the field size will result in step size automatically adjusted to correspond chosen field size. Output dim is shown as integer number and by default it is 32 for all layers, except the last one. When

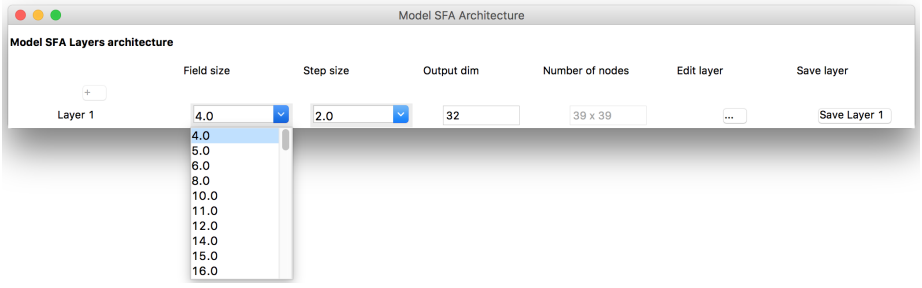

Figure 39: Model SFA Architecture, how to choose field size. There can be adjusted step size and output dim if needed

the layer parameters are chosen, we need to click on button "Save layer x", where x represents the number of layers. Defining every other layer, except the first one, we have also the option "Add last layer" as it is shown in the Figure 40. Difference between the last layer and the others, is that field and step size

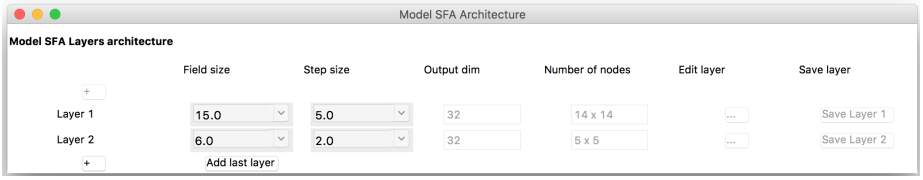

Figure 40: Model SFA Architecture, add last layer option where field and step size cannot be changed and output dim can be adjusted

can not be adjusted. The only parameter which we can change is the output dimension, which is by default 64. When the last layer is saved then the rest of parameters will be shown, as it is represented in the Figure 41.

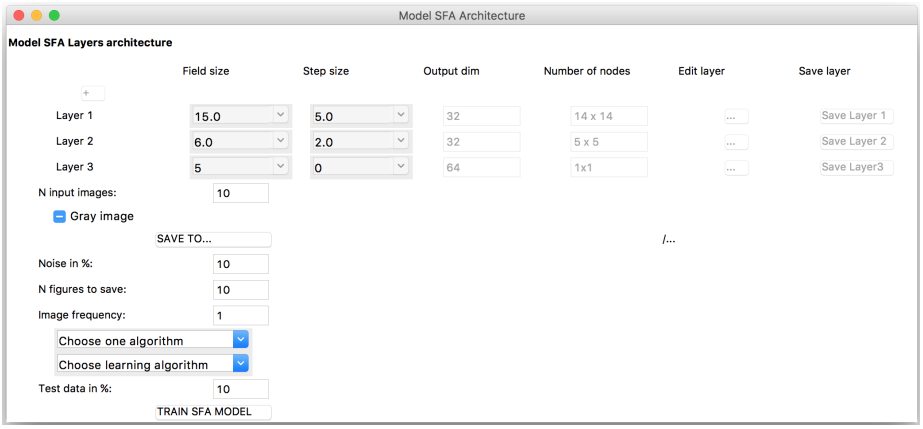

Figure 41: Model SFA Architecture, adjust training data as number of training and test data, image frequency, feature extraction algorithm, learning algorithm, number of images to save to separate PDF file, if input images should be converted to grayscale, and so on.

Here we can define the number of input images (frames) by "N input images" field. Image is by default RGB but with one click on "Gray image" every input image will be converted to grayscale.

Button "Save to..." is used to define where all our results by SFA training should be saved. Noise in percentage represents added random normal distributed numbers to given RGB/grayscale image.

"N figures to save" is used to define how many real images vs. SFA/ICA features should be saved in the separate PDF file.

We can use image frequency to define which n, number of images, should be assumed as an input. According to that, image frequency 3 means we use every third images and save it in the input data.

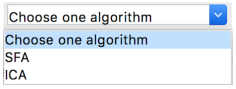

Figure 42: Model SFA Architecture, feature extraction algorithm can be chosen between SFA and ICA

Our output features can be shown through SFA or ICA features, which is added as an option shown in the Figure 42.

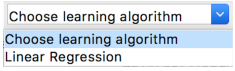

Figure 43: Model SFA Architecture, learning algorithm can be chosen

Also a learning algorithm, for testing the SFA/ICA feature extraction, can be chosen. The only one in this version is "Linear Regression", as it is shown in Figure 43.

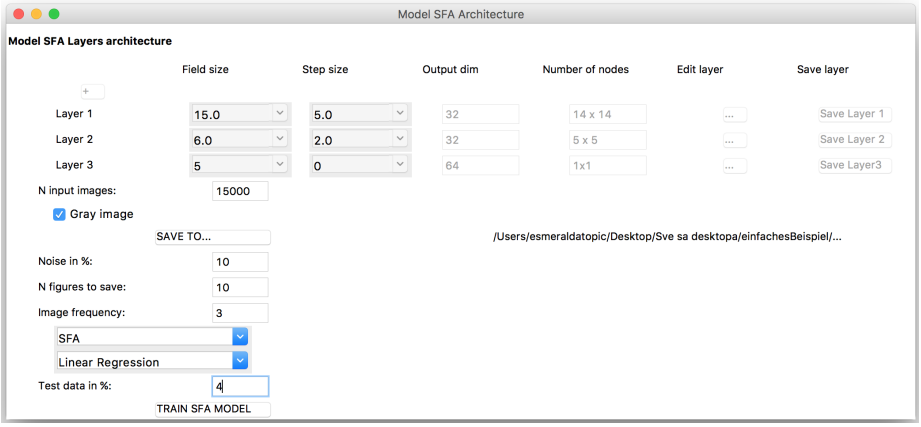

Figure 44: Model SFA Architecture, whole software framework window

The last option from Figure 44 is used to define what percentage of data should be used as test data. The rest is used for training. At the end we can click on button "TRAIN SFA MODEL" to train defined input data with modelled SFA layer architecture. When the training process is done, we get the figures of learned data, where all learned layers and extracted features are already saved in the folder which we defined.