University of Zagreb
Faculty of electrical engineering and Computing

MASTER THESIS num. 1733

Reinforcement learning in simulated systems

Vladimir Dragutin Livaja

Zagreb, June 2018.

UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 9 March 2018

MASTER THESIS ASSIGNMENT No. 1733

Student: **Vladimir Dragutin Livaja (0036473744)**
Study: Computing
Profile: Computer Science

Title: **Reinforcement learning in simulated systems**

Description:

Reinforcement learning is an approach to machine learning that is inspired by behaviorist psychology. Using this approach, algorithms are trained through a system of reward and punishment, similar to how a child learns to perform a new task.
Your task is to study and describe the reinforcement learning technique and to select a study case of a simulated system for which you will design and implement a behavior algorithm based on the reinforcement learning approach. Additionally, perform a detailed evaluation of the implemented algorithm.
The required literature and necessary working conditions will be provided by the Department of Telecommunications.

Issue date: 16 March 2018
Submission date: 29 June 2018

Mentor:
Associate Professor Krešimir Pripužić, PhD

Committee Chair:
Full Professor Siniša Srbljić, PhD

Committee Secretary:
Associate Professor Tomislav Hrkać, PhD

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 9. ožujka 2018.

DIPLOMSKI ZADATAK br. 1733

Pristupnik: **Vladimir Dragutin Livaja (0036473744)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Primjena podržanog učenja u simuliranim sustavima**

Opis zadatka:

Podržano učenje je tehnika strojnog učenja koja se temelji na psihologiji ponašanja. Korištenjem ove tehnike, algoritmi se treniraju korištenjem metode nagrade i kazne, slično načinu na koji se djecu uči savladavati nove zadatke.
Vaš zadatak je proučiti i opisati tehniku podržanog učenja te odabrati studijski slučaj simuliranog sustava za kojeg ćete dizajnirati i implementirati algoritam ponašanja koji se temelji na tehnici podržanog učenja. Osim toga, napravite detaljnu evaluaciju implementiranog algoritma.
Svu potrebnu literaturu i uvjete za rad osigurat će Vam Zavod za telekomunikacije.

Zadatak uručen pristupniku: 16. ožujka 2018.
Rok za predaju rada: 29. lipnja 2018.

Mentor:
Pripužić
Izv. prof. dr. sc. Krešimir Pripužić

Predsjednik odbora za diplomski rad profila:
Prof. dr. sc. Siniša Srbljić

Djelovođa:
Doc. dr. sc. Tomislav Hrkać

Contents

Introduction

In recent years we have seen great achievements in the fields of computer vision with deep neural networks. Both supervised and unsupervised learning have had great success with tasks like image classification, natural language processing, voice recognition and even the creation of synthetic data. However, even with all these great leaps forward and successes, none of these algorithms have been able to generalize and thus cannot be considered a 'general A.I.'.

In this work we will be implementing a reinforcement learning method that will learn to play the First person shooter game "Doom" and the famous Atari game known as "Pacman". We will implement a single reinforcement learning algorithm which we will then analyze on the various scenarios. Our algorithm will use state approximations via deep convolutional networks. Our simulated environment will produce states in the form of screen pixels, possible actions that we will be able to take and rewards we have received based on those decisions.

Our screen pixels representing the current state of our environment will be input to our deep convolutional network which will then produce output determining the appropriate action we should take. The idea will be to update the parameters in the convolutional network in such a way that it will always output the correct action we should take based on the current state we are in; such that we maximize our overall reward. Unlike the well-known supervised and unsupervised learning algorithms, reinforcement learning lays somewhere in-between. We will use a form of learning where many tuples of the current state, taken action, next state and reward are saved into memory buffers which are then at specific time intervals used to draw uniform samples from and are fed to the convolutional network.

This work will be broken up into four chapters. The first chapter will be a brief introduction into deep learning in general with a higher emphasis on deep convolutional networks. The second chapter of this work will focus primarily on reinforcement learning

and the current challenges it faces. The third chapter will have implementation details of the system implemented in this work. The fourth chapter will in detail show the carried out experiments and their results based on the model implemented in this work. Finally, a conclusion which will define the problems faced in this work and possibilities for future improvement.

1. Deep Learning

This chapter gives a brief introduction to convolutional networks and the ways these deep models are trained. Deep learning in general is a form of machine learning based on learning data representations rather than task-specific algorithms. Deep learning has been applied to fields such as computer vision, speech recognition, natural language processing, audio recognitions, bioinformatics and even drug design, where they have shown results sometimes even greater to that of human experts.

1.1. Convolutional layer

Convolutional neural networks are a specialized form of neural networks for processing data with a grid-like topology.

The operation of convolution is defined with two real functions and outputs a third function which represents the amount of overlapping between the first and second function.

The operation of convolution is widely used in areas such as quantum mechanics, statistics, signal processing and computer vision. We will focus only on the discrete form of convolution as it is the one used in computer vision. Discrete convolution can be viewed as multiplication by a matrix.

If we were to imagine performing a convolutional operation upon an image of size 256x256 then we can supply the next formula:

$$feature\ map = input * kernel = \sum_{y=0}^{256}\left(\sum_{x=0}^{256} input(x-a, y-b)kernel(x,y)\right)$$

In this formula our input will be an image with multiple different kernels which are also known as filters. What these kernels are attempting to do is feature engineering; or to learn what is relevant to the image which we can imagine as a filtering operation. In the context of convolutional neural networks instead of having fixed numbers in our kernels, we assign parameters to these kernels which will be trained of the data. So as we train our convolutional net, the kernel will get better at filtering a given image and extracting specific features.

An example of a two-dimensional convolution can be viewed in the image below:

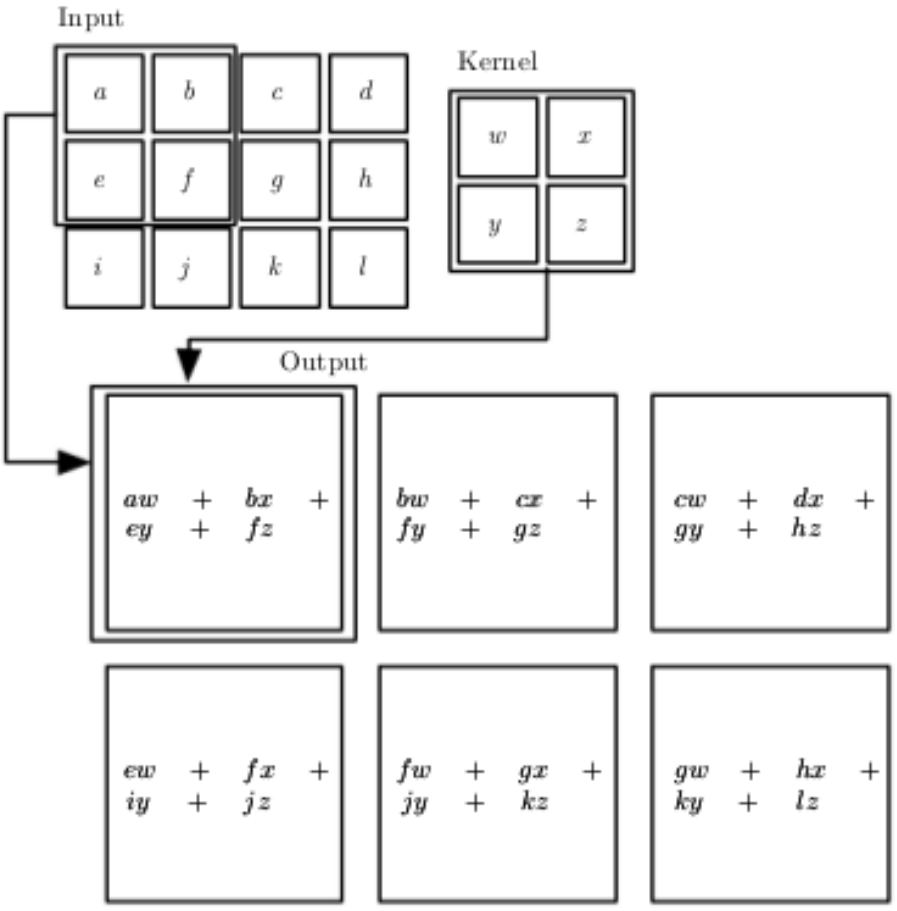

Figure 1: Sample of a convolution operation [2]

Convolution is very useful to neural networks because it's easy to integrate into a neural network and it has the property of being able to model local interactions and share parameters. Unlike fully connected layers inside a neural network where we have parameters for describing the interaction of each input and each output, Convolutional networks have sparse interactions; taking into account our kernel is usually smaller than the input image. The spatial extent of this connectivity is a hyper-parameter called the receptive field of the neuron. An example of an edge detecting convolutional kernel can be viewed in the next image:

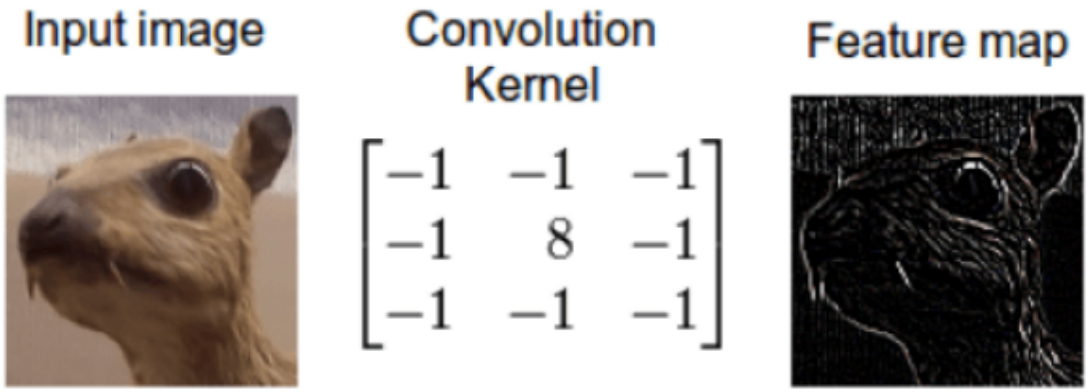

Figure 2: Edge detecting kernel [6]

Such layers require fewer parameters which speed's up evaluation time and memory requirements. If there are *m* inputs and *n* outputs, then for a fully connected layer the runtime complexity is $O(mxn)$. If we limit each output to have *k* connections to the input then the runtime complexity becomes $O(kxn)$.

Another advantage gained using convolutional layers can be viewed in the deeper layers. These deeper layers may indirectly interact with a larger portion of the input. This enables the network to describe complicated interactions that various parts of the input may have.

Parameter sharing in convolutional networks can be accomplished via the kernel. Each member of the kernel is used at every position of the input. Rather than learning a separate set of parameters for every location, we learn only one set. In classical fully connected neural networks, a parameter from the output is bound to a specific position in the input making it less efficient in terms of memory consumption and evaluation speed. The final advantage to be noted that a convolutional layer has upon its fully connected counterpart is the property of equivariance to translation. Equivariance simply means that the output changes in the same way as its input changes. However, convolutional layers are not equivariant to transformations such as scale or rotation of an image.

1.2. Activation functions

Activation functions are used for introducing nonlinearity into the neural network. Without them, neural networks would not be able to find non-linear dependencies in the data. In deep convolutional networks the most common activations used are the ReLu (1.3), LeakyReLu (1.4), hyperbolic tangent (1.2) and the sigmoid function (1.1).

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1.1}$$

$$f(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{1.2}$$

The Sigmoidal function was perhaps the first function ever used in neural networks in general. Sigmoidal units saturate across most of their domain. They saturate to +1 when the input is very large and 0 when the input is very negative and are only sensitive to their input when it varies near 0. This saturation make learning rather difficult because the gradients corresponding to very large inputs results in a gradient near 0 which makes the training of a neural network difficult. The hyperbolic tangent function is very similar to the sigmoidal function although it in general does perform better. It resembles the identity function more closely, in the sense that $tanh(0) = 0$. Because ***tanh*** is similar to the identity near 0, training a deep neural network resembles training a linear model.

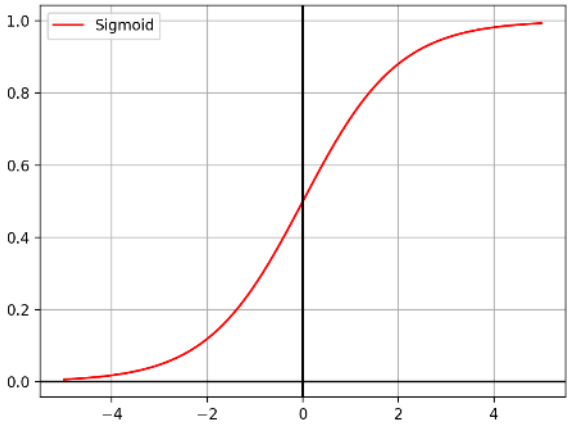

Figure 3: Sigmoid function

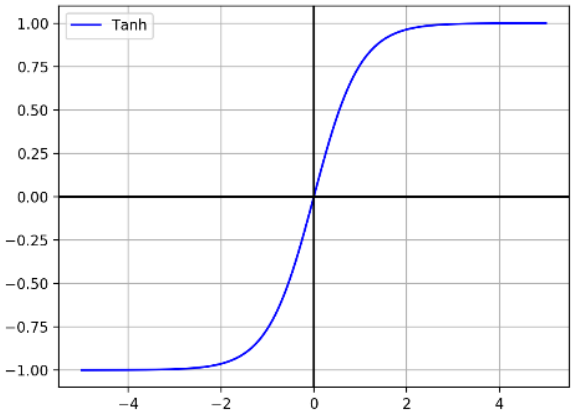

Figure 4: Hyperbolic tangent function

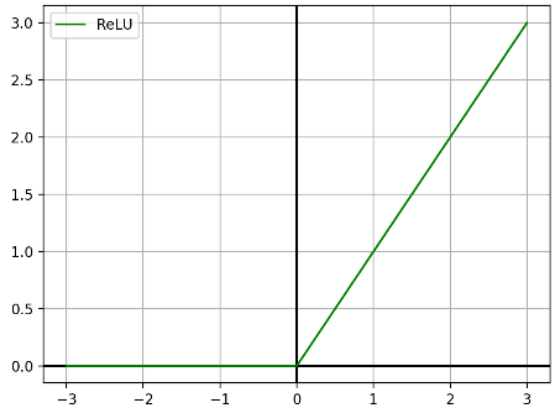

Figure 5: ReLu function

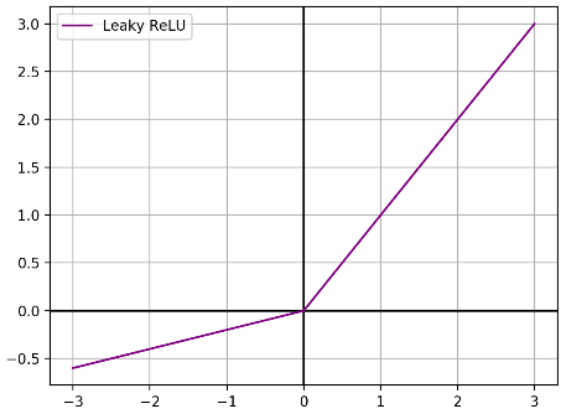

Figure 6: LeakyReLu function

$$f(x) = \max(0, x) \tag{1.3}$$

$$f(x) = \max(x, ax) \tag{1.4}$$

The ReLu activation function is the most used activation function today and a usual first choice. They are easy to optimize as they are similar to linear units. This in turn means that it does not saturate as the sigmoidal function mentioned earlier. For very large inputs, the gradient also remains large. The only drawback to the ReLu activation is that it cannot learn via gradient based methods for which their activation is equal to or below zero. To solve this problem of the vanishing gradient for inputs below zero we can use the slightly modified ReLu known as LeakyReLu. The LeakyReLu has one parameter called *alpha* which is generally a very small number which is used for negative inputs which in turn gives a small gradient for the case of negative input.

1.3. Optimization for training Convolutional neural networks

When speaking of neural network optimization in general, our objective function is more often than not, the mean squared error function. Optimization then consists of finding the values of the weights in the neural network to minimize the objective function. Historically, the gradient based technique known as gradient descent has been the most popular and widely known. However, gradient descent has shown many problems in the field of deep learning due to one parameter known as the learning rate. Gradient descent would have a fixed learning rate which contradicts the nature of the objective function, for it is often highly sensitive to some directions in parameter space and less so in others. In this part we will examine two optimization methods with adaptive learning rates based on model parameters.

1.3.1. RMSProp

RMSProp is an algorithm based on gradient descent with an adaptive learning rate which in turn speeds up convergence. RMSProp has shown many advantages such as being a very robust optimizer which has pseudo curvature information. It can also deal with stochastic objectives very nicely, making it applicable to mini-batch learning. RMSProp uses the magnitude of recent gradients to normalize the current gradients. We keep running averages over the root mean squared gradients which we use to divide the current gradient. If we let $f'(\theta_t)$ be the derivative of the objective function with respect to the parameters of the network at time *t*. given a step rate α, a decay rate γ , a running average over the root mean squared gradient v_t and finally a momentum term β we have the following:

$$m_{t+1} = \gamma m_t + (1-\gamma) f'(\theta_t)^2$$

$$v_{t+1} = \beta v_t - \frac{\alpha f'(\theta_t)}{\sqrt{{m_{t+1}}^2 + \epsilon}}$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

Our normalizing factor m_t with a properly adjusted decay rate takes into account more recent history of the gradients instead of keeping them all as do other algorithms prior to RMSProp making it much more adaptive to the objective function search space. G. Hinton, the author of the algorithm also proposes some values for these hyper-parameters as a good starting point. The decay rate $\gamma = 0.9$ and a learning rate $\alpha = 0.001$. Finally, the parameter ϵ serves only for the purpose of numerical stability by avoiding dividing by 0. It is generally set to 10^{-8}.

1.3.2. Adam

The Adam method computes adaptive learning rates for all parameters based on estimates of the first and second moments of the gradients. Adam is composed of two other algorithms known as AdaGrad and RMSProp which was mentioned earlier. Unlike RMSProp which adapts learning rates for the parameters based on the first moment of the gradients, Adam utilizes the average of the second moment also. Adam therefor computes the exponential running average of both the first and second moments of the gradient and uses two hyper-parameters β_1 and β_2 which are there respective decay rates. The following formula shows how Adam is implemented:

$$m_{t+1} = \gamma m_t + (1 - \beta_1) f'(\theta_t)^2$$

$$g_{t+1} = \gamma g_t + (1 - \beta_2) f'(\theta_t)$$

$$\widehat{m_{t+1}} = \frac{m_{t+1}}{1 - {\beta_1}^{t+1}}$$

$$\widehat{g_{t+1}} = \frac{g_{t+1}}{1 - {\beta_2}^{t+1}}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha \widehat{g_{t+1}}}{\sqrt{\widehat{m_{t+1}}} + \epsilon}$$

Some advantages of using Adam include its simplicity, computational efficiency, little memory requirements and invariance to the diagonal rescale of the gradients. It is well suited for large problems, non-stationary objectives and noisy or sparse gradients. The hyper-parameters of Adam are also very intuitive to interpret and require very little tuning.

1.4. Batch normalization

Batch normalization is a technique used in deep learning for increasing the training speed and improvements in stability in neural networks. As normalizing the input has shown to be very useful in almost all machine learning and deep learning tasks; batch normalization simply replicates the task of normalization of inputs in each layer in such a way that they have a mean output of zero and unit variance. It's called batch normalization because the normalization is performed based on the inputs from the batch instead of the entire training population. The mean and variance are separately calculated for each dimension of the input. The core problem that batch normalization tackles is what is known as Internal Covariate Shift. Internal Covariate Shift is the change in the distribution of network activations due to the change in network parameters during training.

However normalizing each input of a layer may potentially change the layers representational power. Normalization of inputs that are later fed to the sigmoid activation would constrain it to a linear regime. To avoid this, we make sure that the inserted transformation can represent the identity transformation. To succeed in this task additional parameters are introduced γ and β for each activation x which scale and shift the normalized value.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 7: Batch normalization algorithm [2]

These parameters are learned for each dimension of the input and restore the representational power of the network.

Batch normalization incurs many advantages:

- Networks train faster
- Allows the use of higher learning rates
- Simpler weight initialization
- Enables the use of activation functions such as sigmoid and hyperbolic tangent mentioned earlier
- Simplification of constructing deeper networks
- Provides slight regularization because of the added noise added during the obtaining of statistics based on the current batch being evaluated.

2. Reinforcement learning

In this chapter we will go through the building blocks of reinforcement learning that will be needed in creating an algorithm that will be able to learn in a 3D environment of the First Person Shooter Doom and 2D environment in a simulation of the Pacman environment. Reinforcement learning was initially inspired by behavioral psychology and is viewed today as being one of the first steps towards a general AI algorithm. Reinforcement learning has been used in various places such as games, trading, robots and even autonomous driving cars. However, training reinforcement learning models is a slow and difficult task, taking models days to converge on rather simple problems.

2.1. Elements of reinforcement learning

At the highest level, a reinforcement learning algorithm consists of an agent and an environment. An agent interacts with the environment in such a way that is takes actions in its environment which then leads it into a new state in the environment and so on. The agent primarily wants to learn how to behave optimally in its environment or to maximize its future reward. There are four main sub-elements of a reinforcement learning system.

A ***policy*** defines a learning agent's way of behaving at a given time in a given state. A policy is a mapping from perceived states of the environment to actions to be taken when in those states. Policies may be simple look up tables, whereas in other cases they can be deep neural networks. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic.

A ***reward signal*** defines the goal of the reinforcement learning algorithm. Based on the actions taken in specific states, the environment returns a reward to the agent. The agent's primary goal is to maximize the reward in the long run. The policy mentioned earlier is changed based on the reward signal. Actions leading to small rewards may be altered so that another action may be selected to accumulate a higher reward.

A reward signal tells us what is good in an immediate sense, whereas a ***value function*** tells us what is good in the long run. The value of a state tells the agent what amount of reward the agent may expect to accumulate in the future. Where rewards determine the immediate desirability of states in an environment, values tell us the long run desirability of states in an environment. Some states may yield small immediate returns yet still be of greater value than all other states because it can yield a high long run return in reward.

The final element of some reinforcement learning algorithms is a ***model*** of the environment. A model mimics the behavior of the environment and allows inferences to be made about how the environment will behave. Based on a state-action pair, the model can predict the resultant next state and received reward. Reinforcement learning problems that use models are called model-based methods as opposed to simpler model-free methods that are explicitly trial-and-error learners.

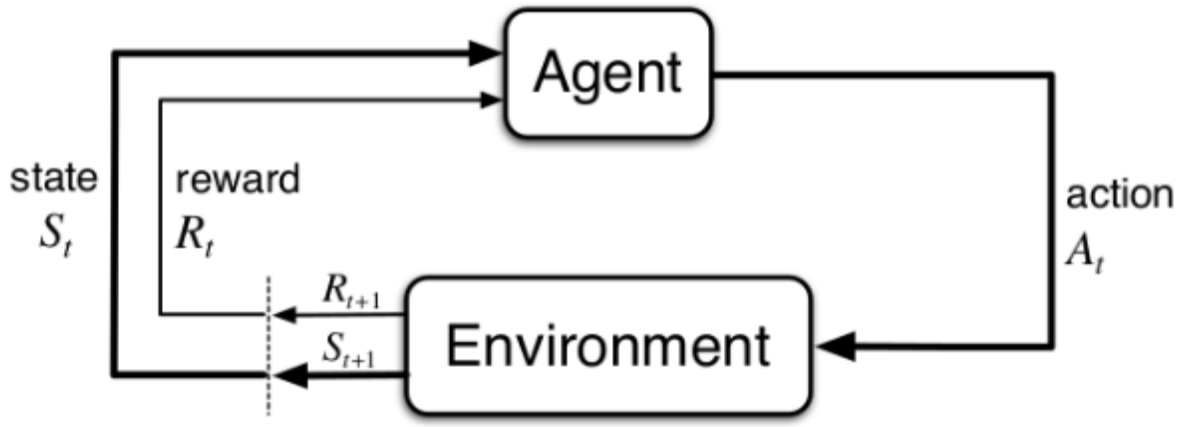

Figure 8: A reinforcement learning system depicting the three main components: an agent, its environment and the couples interaction. [1]

2.2. Goals of reinforcement learning

The purpose of an agent is formalized in terms of a special signal named the reward, which passes from the environment to the agent. At each time step, the reward is a simple number $R_t \in R$.

Instead of maximizing immediate reward, the agent maximizes the cumulative reward in the long run. The reward signal is not the place to impart to the agent prior knowledge about how to achieve its goal. We must not give rewards to agents for achieving sub-goals as the agent could potentially learn how to maximize reward via achieving these sub-goals without being able to achieve its primary goal we set it to learn.

2.2.1. Returns and episodes

We have stated that the end goal of the agent is to maximize the cumulative reward it receives in the long run. We can define all rewards received after time step *t* as $R_{t+1}, R_{t+2}, R_{t+3}\ldots$ etc. The sequence can be short or even infinite in size, therefor the natural question that arises is "what part of the future reward do we wish to maximize?". In reinforcement learning we wish to maximize the expected return, where we denote the return G_t. In its simplest form G_t is simply defined:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T$$

Where *T* is the final time step. The approach is sensible where there is a notion of a final time step, that is, the agent-environment interaction breaks naturally into subsequences, which we call episodes, such as a round in the game of Doom or Pacman which we will explore shortly. Each episode ends in special state called the terminal state. Episodes may end differently, winning or losing, however the next episode starts independently of how the previous one ended. Thus we can state that the agent always finishes in the same terminal state with different rewards and outcomes. Tasks with episodes are called episodic tasks.

As we mentioned earlier, episodes could last infinitely long and therefor even a small reward accumulated through many time steps could lead to an infinite reward, allowing the agent then to learn a random policy. To counter this we will introduce the concept of discounting. According to this approach are agent tries to select actions so that the sum of the discounted rewards it receives is maximized. The discounted return can be formally stated as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where γ is a parameter called the discount rate. Its value varies between 0 and 1.

The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. If $\gamma = 0$ the agent is referred to as myopic, concentrated only on maximizing immediate reward. As γ approaches 1, the agent takes future rewards more and more into account.

2.3. Policies and value functions

Value functions are almost universally used in all reinforcement learning algorithms; functions which determine how good it is being in a specific state (state-action pair) based on the expected return one may expect from it following the agents current policy.

A policy is a mapping from states to probabilities of selecting each possible action at the disposal of the agent. If the agent is following the policy π at time t, then $\pi(a|s)$ is the probability of the agent selecting the action a in state s.

The value of a specific state s under a policy π is denoted $V_\pi(s)$ which is understood as the expected return when starting in state s following policy π thereafter. We call this function the state-value function for policy π. We can formally define it as follows:

$$V_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right], for\ all\ s \in S$$

In a similar fashion we can define the value of taking a specific action a in state s under a policy π which we denote as $Q_\pi(s,a)$ representing the expected return after taking action a in state s following policy π.

We define this formally as follows:

$$Q_\pi(s,a) = E_\pi[G_t|A_t = a, S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|A_t = a, S_t = s\right]$$

We call Q_π the action-value function for policy π.

2.3.1. The optimal policy and its value functions

Solving reinforcement learning problems comes down to finding a policy which achieves the maximum amount of reward possible in the long run.

A policy π is deemed better than another π' if its expected return is greater than or equal to that of π' for all states or more formally $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)$ for all $s \in S$.

There may be more than one optimal policy which we will denote as π^*. Although many optimal policies may exist, they share the same state-value function, called the optimal state-value function, denoted V_* and formally defined as:

$$V_*(s) = \max_\pi V_\pi(s), \forall s \in S$$

As with optimal value-state functions, we can also define optimal action-state value functions, denoted Q_* and defined formally as:

$$Q_* = \max_\pi Q_\pi(s,a), \forall s \in S\ and\ \forall a \in A(s)$$

The state-action pair (s,a) gives the expected return for taking action a in state s and then continuing to follow an optimal policy π. The equation for the optimal action-state value function can be rewritten in terms of the optimal state-value function as follows:

$$Q_* = E[R_{t+1} + \gamma V_*(S_{t+1})|S_t = s, A_t = a]$$

2.4. Dueling architecture

Approximating Q-values is somewhat problematic in the sense that only the state which outputs the maximum Q-value is updated whereas the Q-values of remaining states remain unaltered. To avoid this we implement what is known as the dueling architecture. The dueling architecture separates the state values and the state-action advantages. The dueling architecture consists of two streams that represent the value and advantage functions, while sharing a common convolutional learning module. The two streams are aggregated via a special aggregating layer to produce the final Q values. We first define the advantage function as follows:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s, a)$$

This function gives us a sense of the importance of each action. V measures the quality of a specific state whereas the Q measures the value of choosing a particular action when in a specific state.

The key insight is that for many states it is not necessary to estimate the value of each action choice. The module that combines the two streams created from the convolutional output is as follows:

$$Q(s, a) = V(s, a) + (A(s, a) - \frac{1}{|A|}\sum_{a'} A(s, a'))$$

The advantage of this architecture is its ability in learning the state value function efficiently. With every Q value update, the value stream V is updated which contrasts the single stream architecture where only for one the actions is updated. The following image shows the structure of the defined dueling architecture.

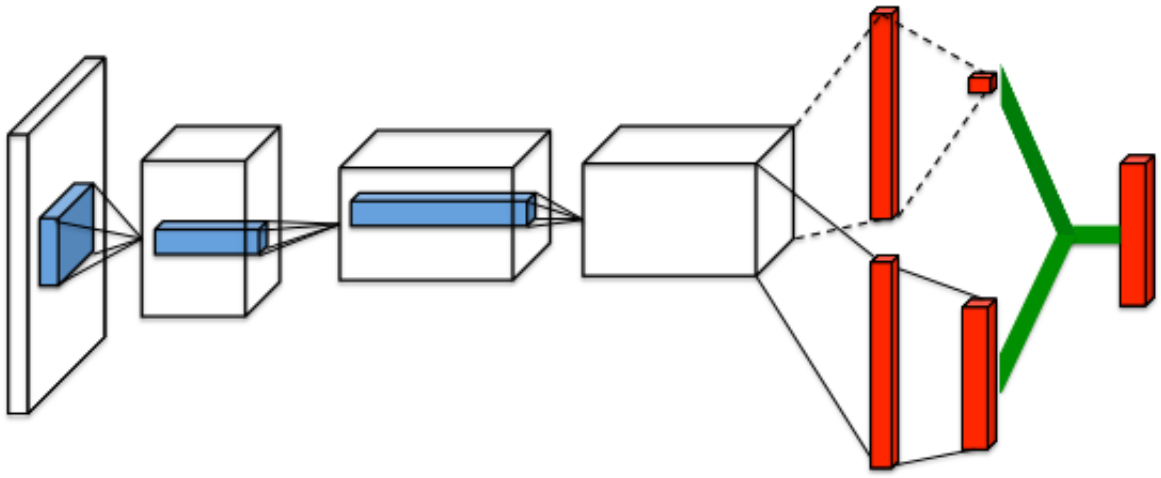

Figure 9: The dueling architecture where instead of a fully connected layer outputting Q values directly; the convolutional output is broken up into two streams outputting the state value and action advantages. [7]

2.5. Q learning

Before we articulate Q learning we will comment on the two specific types of reinforcement learning methods called off-policy and on-policy methods.

Off-policy methods evaluate or improve a policy different from that used to generate the data, whereas on-policy methods evaluate or improve the policy that is used to make decisions.

Off-policy methods use two policies, one that is learned to become the optimal policy and one that is more exploratory and is used to generate behavior. The policy being learned is the target policy and the policy being used to generate behavior is called the behavior policy.

Q-learning is an off-policy reinforcement learning algorithm which we formally define as:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a')$$

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha \left[R_{t+1} + \gamma \max_{a} Q(S_{t+1},a) - Q(S_t,A_t)\right]$$

These definitions are also known as the Bellman equation. As we can see, with a learning rate set to 1, both become equivalent. With the update rule defined, the learned action-value function Q, directly approximates Q_*, the optimal action-value function, independent of the policy being followed.

To give some intuition behind $Q(s,a)$, we can think of it as the score we will receive at the end of the game after performing action a in state s. The main idea in Q-learning is that we can iteratively approximate the Q-function using the bellman equation. In its simplest form the Q function is implemented as a simple lookup table.

The following image shows pseudo-code for the Q-learning reinforcement method:

Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal,\cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Figure 10: Pseudo code for implementing Q learning. [1]

2.6. Value function approximation

In many real world applications, the state space is far too large for tabular methods; therefore we must use good approximations using limited computational resources that we have.

In many tasks with large state spaces, almost every state we encounter will have never been visited before. To be able to make quality decisions, it is necessary for the agent

to generalize from earlier encounters which may in some ways be similar to the current state at hand.

Function approximation is an instance of supervised learning, because it takes instances from a desired function such as the value function or state-action value function and attempts to generalize from them to an approximation of this function.

Formally we can define the approximation of the value functions mentioned earlier. The approximation of the value function is then denoted as:

$$\hat{V}(s, w) \approx V_{\pi}(s)$$

Where w are the parameters of the function in the form of a weight vector $w \in R^{d}$.

We can also do the same for the action-state value function which we will denote with:

$$\hat{Q}(s, a, w) \approx Q_{\pi}(s, a)$$

In general the number of weights is much less than the number of states and changing one weight results in the estimated values of many states.

For function approximation to work let us examine an individual update with the notation $s \rightarrow u$, where s is the state updated and u is the update target that s's estimated value should be shifted toward. In simpleton terms, we are stating that s should be more like u.

Machine learning methods that mimic input-output examples in this way are called supervised learning methods.

2.6.1. Q function approximation with neural networks

In this work we will be approximating Q functions with deep neural networks which have shown great success in many reinforcement learning applications.

What we could do is input raw screen pixels from our environment, representing the state and make a forward pass in our neural network to output all Q values based on the current state. However the current state cannot be defined based on only a single frame as it doesn't give us information about moving or stationary targets or the speed of a target. A simple solution to this is to use multipal sequential frames and feed that as input to the neural network. As mentioned earlier, neural networks are incredibly powerful in coming up with good features for highly structured data. The next image demonstrates what we are trying to accomplish:

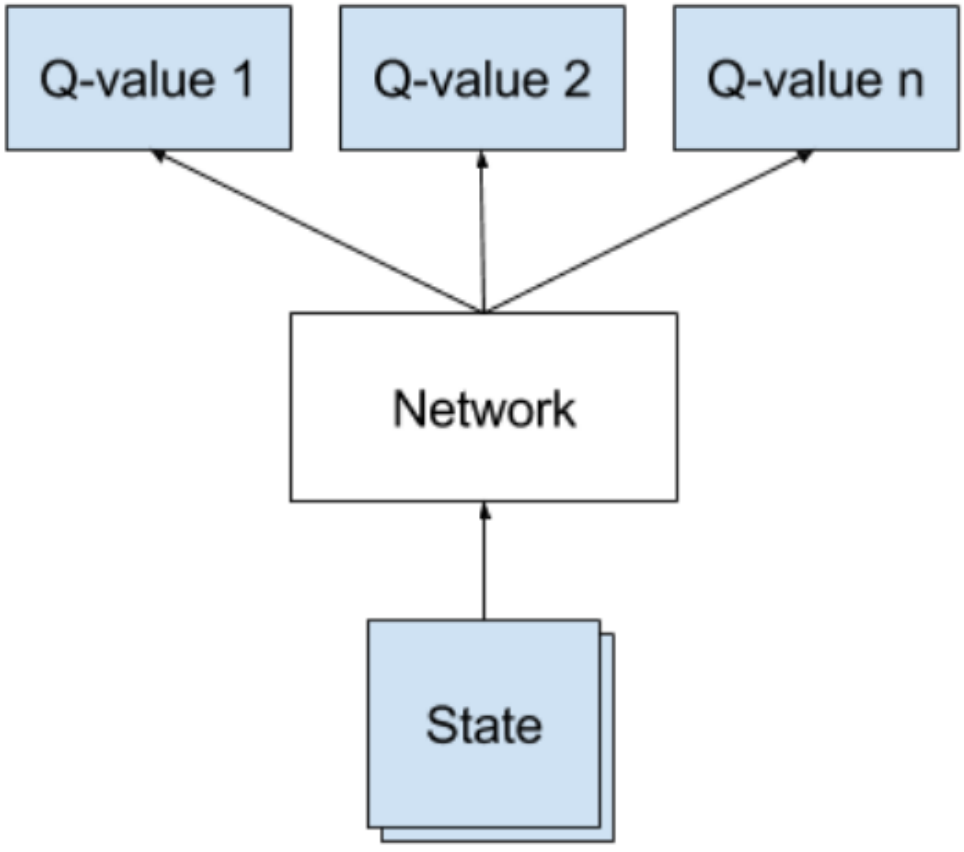

Figure 11: Q value approximation done via a neural network with the current state as the input. [1]

Since Q values can be any real number, it makes it a regression task that can be optimized with a simple squared error loss.

$$L = \frac{1}{2}\left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]^2$$

Given a transition which consists of the current state, action taken, reward received and next state the sequence of events for training such a network is as follows:

1. Attain Q values from feeding the current state s and making a forward pass in the neural network.
2. Do a forward pass in the network with the next state s' and select the action that gives the maximum Q value.
3. We set the Q value target for action a to $r + \gamma \max_{a'} Q(s', a')$. For all the other actions, the Q value is not changed.
4. Update the weights using backpropagation.

2.7. Experience replay

Training deep neural networks is almost universally done with the supervised learning method. If we were to train our Q learning algorithm with classical supervised learning, we would introduce a higher variance in our update because successive updates are highly correlated with one another.

Experience replay is a method which stores the agent's experiences at each time step in memory that is then later accessed to perform training and updates to the network. Each instance in the replay memory is consisted of a tuple representing the starting state, action taken, reward received and next state or more formally:

$$(S_t, A_t, R_{t+1}, S_{t+1})$$

After the accumulation of many such experiences, sequential updates can then be made to the network with mini-batches with a batch uniformly sampled from the replay

memory. This method has been shown to reduce the variance of the updates which in turn makes the algorithm more stable during training.

2.8. Exploration vs. Exploitation

As we have been able to observe, Q learning in its essence tries to assign values to actions based on state and it does so by propagating rewards back in time.

In the beginning we know nothing. Our Q values are random as our weights are randomly initialized. If we were to have a greedy algorithm, it would always be selecting the action with the maximum Q value, which would probably be a wrong selection. Our network hasn't been given a chance to explore its possibilities and to find out which action truly is the greatest selection based on its current state. In Reinforcement learning this is known as the Exploration-Exploitation dilemma.

Thus what we want is for our algorithm during the very beginning of its training to explore as much as possible and as it slowly converges to start exploiting these Q values it has learned. There are many ways of tackling this problem, we will discuss only two such algorithms which will later be implemented.

The first of such algorithms we will analyze is the $\epsilon - greedy$ algorithm. With probability ϵ choose a random action is selected, otherwise we go with the greedy action or the action that gives the highest Q value based on the current state. We usually start with a very high ϵ which we then as time goes by, decrease to a very small number such as 0.01.

The second algorithm is the Boltzmann exploration. The Boltzmann exploration utilizes the information in the estimated Q values produced by our network. Instead of taking either the optimal or a random action, this method involves taking an action with weighted probabilities.

To accomplish this we use the softmax over the networks estimates of values for each action. This way the best action has the highest probability of being chosen but

it is not guaranteed. The Boltzmann methods main advantage over the $\epsilon - greedy$ method mentioned earlier is that the value of the other functions can also be taken into account. This method has then the ability to entirely ignore sub-optimal actions but give a chance to very promising actions. The following images demonstrate both the $\epsilon - greedy$ and the $Boltzmann$ methods.

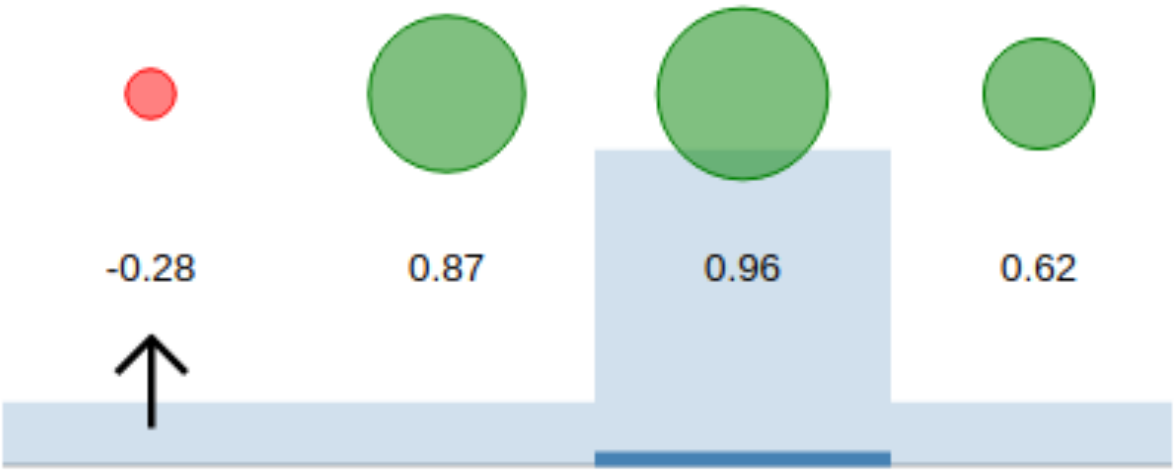

Figure 12: Probability distribution based on the epsilon-greedy algorithm where the action with the largest Q value has a probability of being selected 1-epsilon, whereas the rest are evenly distributed.

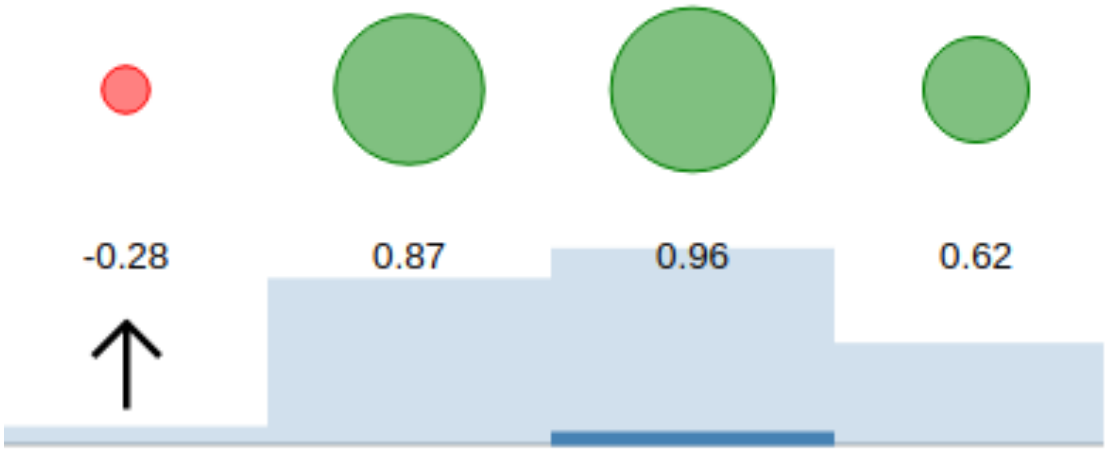

Figure 13: Probability distribution based on the Boltzmann method where an actions probability of being selected is based on its Q value which passes through the softmax function.

3. Implementation

In this work, we will develop a deep Q recurrent network in the Python 3.6 programming language. We will also use specialized libraries for deep learning and matrix manipulation such as Tensorflow and Numpy. Tensorflow is an open-source software library for dataflow programming across a range of tasks. It's a symbolic math library, and is also used for machine learning applications such as neural networks. It can be used both for research and production. The Numpy framework is used for efficient computations over multidimensional arrays.

3.1. Model environment

For our environment we will be using a platform called ViZDoom and OpenAI. ViZDoom is a Doom-based AI research platform for reinforcement learning from raw visual information. It allows developing AI bots that play Doom using only the screen buffer. ViZDoom is primarily intended for research in machine visual learning, and deep reinforcement learning, in particular. In our specific case, we will be using 2 different scenarios from ViZDoom. We will start of by using the basic scenario and finally the "defend the center" scenario. OpenAI is similar to VizDoom in the sense that it provides a much richer toolkit and environments for training reinforcement learning agents. From the OpenAI gym we will be using the Pacman environment. All of our environments are episodic tasks.

3.1.1. Basic Scenario

The basic scenario is our most simple environment. Each episode lasts for 300 tics (35 tics in a second) or when the monster gets killed. Each action will also produce a reward; -6 for shooting and missing which will incentivize our agent not to waste ammunition, 100 for killing the monster and -1 otherwise. As we can see, the environment is a Markov Decision Process where states will be the raw screen pixels.

Our preprocessing will be only remove the roof as it garners no relevant information and then each image will be resized to 84x84x1 and also converted to greyscale. The actions that can be taken in this scenario are moving left, moving right and shooting. Our agent has only 50 bullets at his disposal and one bullet is sufficient in eliminating his opponent.

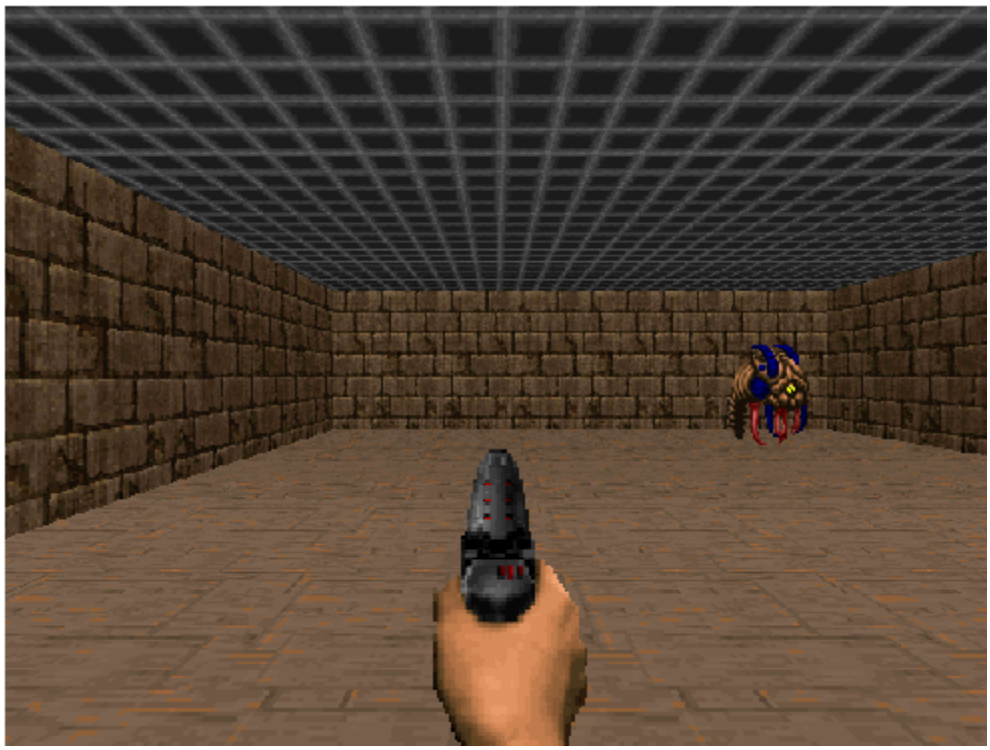

Figure 14: A frame from the basic environment in the VizDoom simulation.

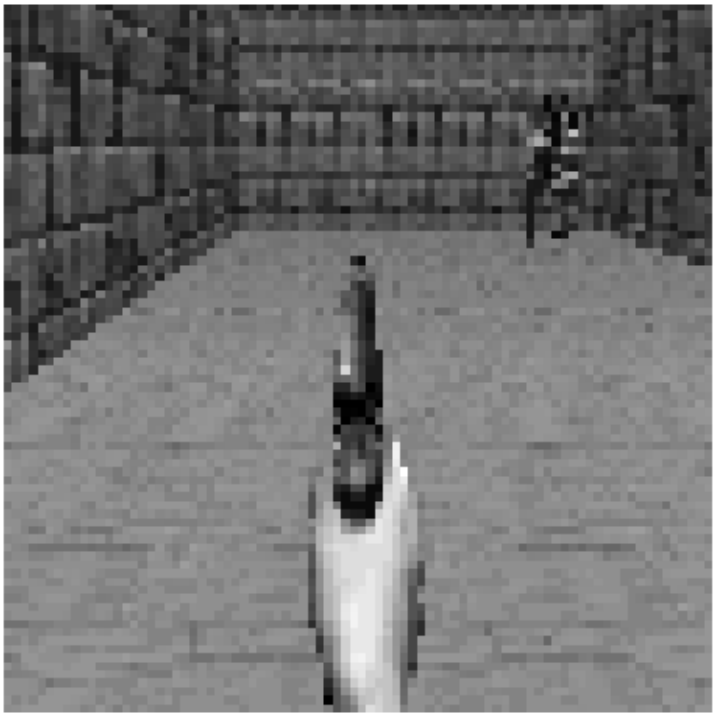

Figure 15: Preprocessed frame from the basic environment. Each frame is down sampled to a size of 84x84x1 with the roof removed as it contains no relevant information.

3.1.2. Defend the center

The purpose of this scenario is to teach the agent that killing the monsters is good and monsters killing you are bad. In addition, wasting ammunition is not very good either. In the original scenario the agent was rewarded only for killing the monster and the rest was left to the agent to figure out. However, in order to speed up training, we altered the game in such a way as to produce negative 3 reward for firing a shot and missing and a positive 55 reward for eliminating an enemy and left the death reward unaltered in an attempt to produce better results. Altering the game system is beyond the scope of this work; however I recommend using the Doom editor tool known as “Slade” for anyone interested. The map is a large circle. The player is spawned in the exact center. 5 melee-only, monsters are spawned along the wall. Monsters are killed after a single shot. After dying each monster is respawned after some time. Episodes end when the player dies (it's inevitable because of limited ammo). This scenario is also much more

complex because of the fact that this is a partially observable Markov decision process. The raw pixels as input is not our entire state, hence our agent doesn't know what's going on behind him or to his sides. Our preprocessing task will be exactly the same as was explained earlier for the basic scenario. The actions that are available to us in this scenario are turning left, right and attacking. We also have limited ammunition of only 26 bullets at our disposal.

Figure 16: A frame from the defend the center environment

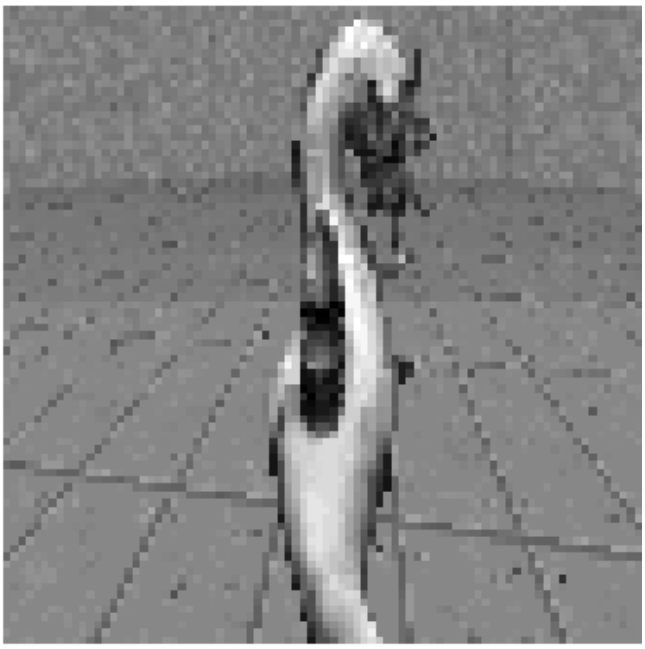

Figure 17: Preprocessed frame from the defend the center scenario. The roof is removed and frame is converted to greyscale.

3.1.3. Pacman

In this environment, the observation is an RGB image of the screen, which is of shape 210x160x3. Each action is repeatedly performed for a duration of k frames, where k is uniformly sampled from {2, 3, 4}. The environment has 9 possible actions which represent the 9 possible positions of the joystick: center, up, right, left, down, upper-right, upper-left, lower-right, and lower-left. The goal in this environment is to achieve as high as a reward as possible by collecting the cherries whilst avoiding being eaten by the ghosts. The Pacman has three lives and the maximum score achievable to him is an integer value of 2800.

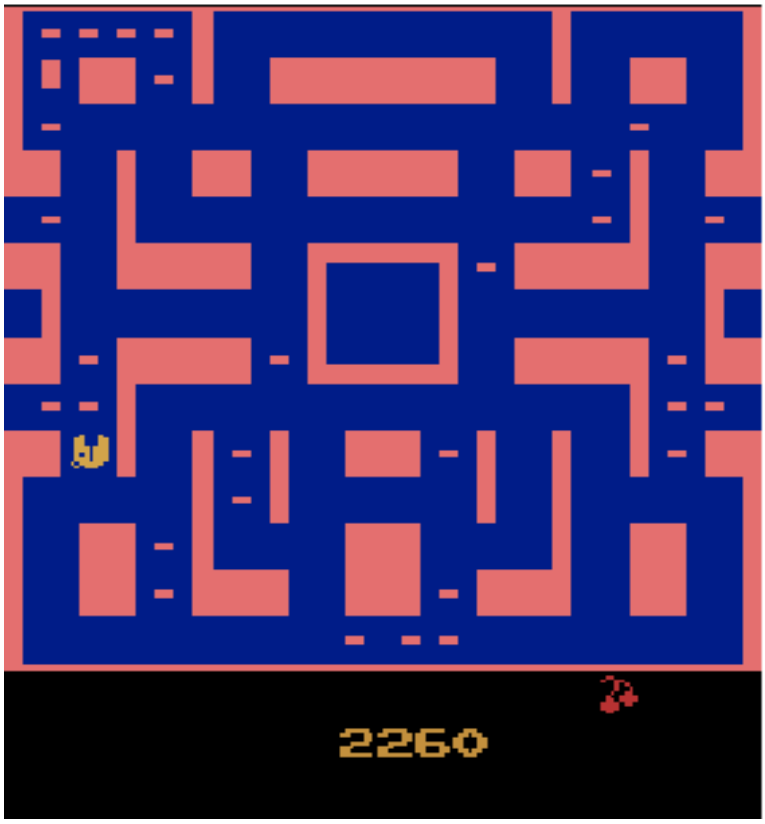

Figure 18: Frame from the pacman environment.

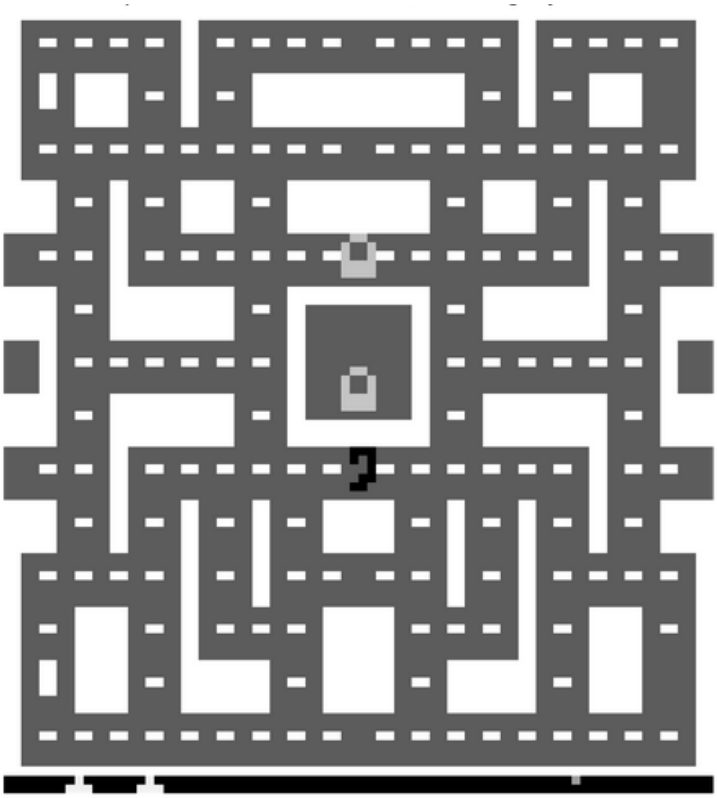

Figure 19: Preprocessed frame of the pacman environment. The image is down sampled to a size of 84x84x1.

3.2. The Agent

Our agent consists of a deep convolutional network that it uses to extract features from the input and outputs Q values which in turn determine our choice of action. Our agent's deep convolutional network consists primarily of only 4 convolutional layers with the amount of filters per layer growing from 32 in our first convolutional layer, to 512 in our fourth convolutional layer. Each convolutional layer uses batch normalization except for our fourth and final convolutional layer. Our filter sizes range from 8x8 in our first convolutional layer, 4x4 for the second, 3x3 for the third and 7x7 for the fourth respectively. Strides decrease linearly from a 4x4 to a 1x1 stride in the fourth convolutional layer. Our output from the fourth convolutional layer is 1x1x512 which we flatten into a vector of size 1x512. For our agent to get a sense of motion and speed, inputting one frame will not be enough. We have two possibilities in solving this

problem, stacking multiple frames to represent a state or using a single frame but having a recurrent neural network in our model. We will choose the latter and the output from the fourth convolutional layer will be the input to our recurrent neural network (LSTM). Based on the input size, our recurrent network will have 512 neurons. Instead of directly outputting Q values, we will break the output of our recurrent network into two equally sized streams and connect each stream to a fully connected layer with no activation function. Our first stream will have outputs corresponding to the amount of actions we have at our disposal while the second will have only one output representing the state value function. These outputs are finally combined as mentioned earlier abiding by the dueling architecture formula. This agent is used in all environments and is trained from scratch each time.

Table 1. The architecture of the neural network used for all the environments.

Input	84x84x1 raw image
1. Layer	Convolutional layer with stride 4x4, 32 filters and filter size 8x8
Output	20x20x32
2. Layer	Convolutional layer with stride 2x2, 64 filters and filter size 4x4 and added batch normalization
Output	9x9x64
3. Layer	Convolutional layer with stride 1x1, 64 filters and filter size 3x3 and added batch normalization
Output	7x7x64
4. Layer	Convolutional layer with stride 1x1, 512 filters and filter size 7x7
Output	1x1x512
5. Layer	LSTM recurrent network with 512 neurons
Output	1x512
6. Layer	Two fully connected layers without activation each taking half of the previous input
Output	1xaction_size and 1x1
7. Layer	Combining previous outputs via dueling architecture
Output	1xaction_size

4. Results

The implemented model was evaluated on the environments defined in the previous chapter. The evaluation of reinforcement learning models is extremely difficult, more so in my case with limited resources and no GPU's. Our evaluations will primarily focus on the reward per episode that our agent is able to attain. The agent was trained with the help of Microsoft Azure's servers. The configuration of the server is 32GB of RAM with 4 cores.

4.1. Basic Scenario

We first use the basic map for evaluation. Being a much simpler map, our agent's task is to eliminate its opponent (Monster). The agent has 50 bullets of ammunition; each missed shot in turn gives a negative reward of negative 5, eliminating the monster gives a positive reward of 100. The episode has a max duration of 300 tics. In the following graph we can see the agent's progress per episode upon received reward.

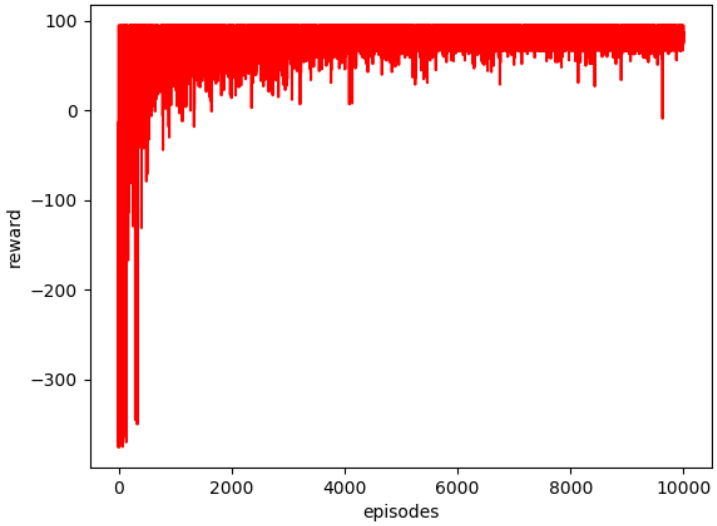

Figure 20: The graph depicts the convergence of the reinforcement agent as the training goes on for longer.

As we can see in the preceding graph, the agent's policy converges in this scenario rather quickly in turn giving maximum reward. Training time took around 2 hours with the earlier mentioned configuration.

4.2. Defend the center

This scenario proves much more difficult than the former. The maximum reward that can be achieved in this scenario is positive 1351, whereas the least amount of reward to be achieved is negative 79. The agent has a limited 26 bullets of ammunition. The following graph demonstrates the achieved results per episode.

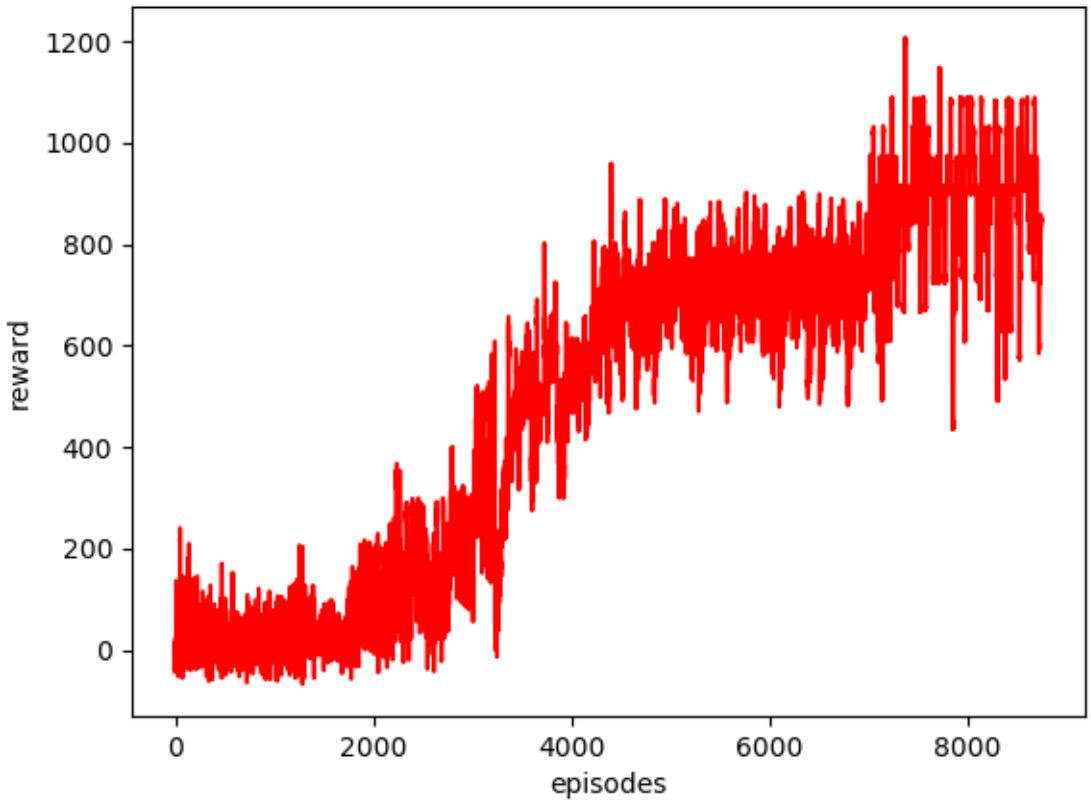

Figure 21: Graph depicting the slow convergence of the "defend the center" scenario.

The graph in the beginning shows a very low reward due to the fact that the agent has 5000 iterations of pure random action selection to help in the exploration of Q values. As the randomness dissipates and the agent becomes greedier we observe a growth in reward per episode with peaks at around 1200. Although it should be noted that even with a relatively good score, there are still high oscillations which could be a result of a lack of training or perhaps the model itself has a lesser capacity than is needed for solving the task. Training of this model lasted for about a week. We would have to restart training due to a memory leak in the VizDoom environment.

4.3. Pacman

This environments complexity lies between the basic and defend the center scenarios. As it is a Markov decision process, the image input is a complete representation of the state. The model was trained for 6 days until acceptable results were achieved. The following graph will show the increase in reward per episode.

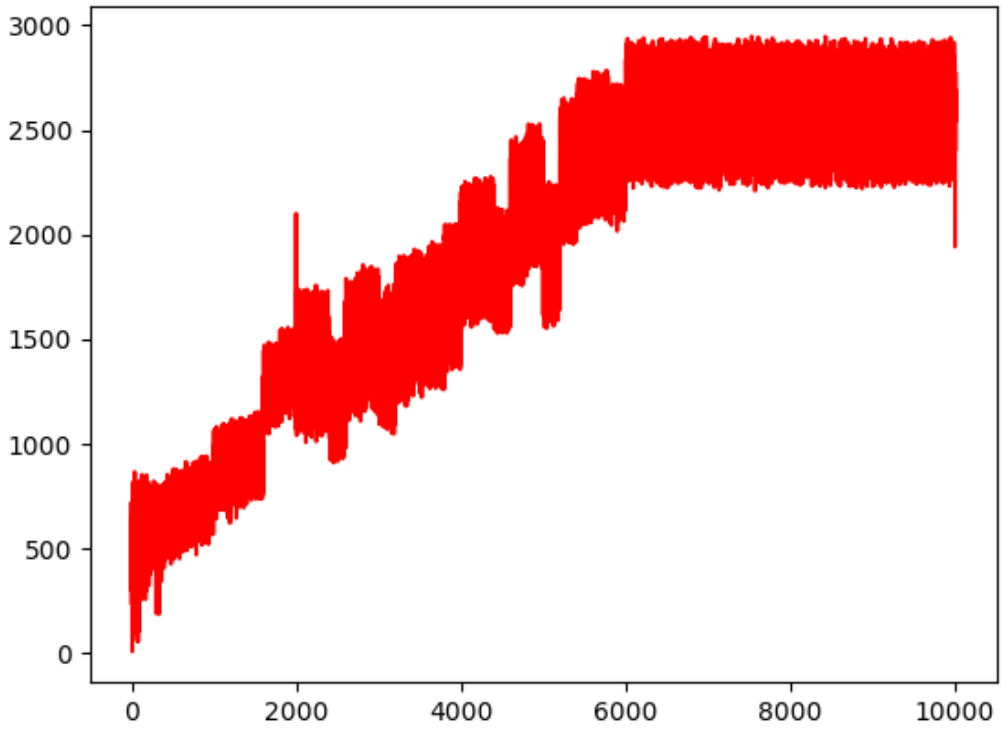

Figure 22: Graph depicting the convergence of the Pacman scenario.

The x-axis represents the episodes while the y-axis represents the received reward. It is also worth noting that the maximum Q value per episode grows slowly from a mean value of 0 and is bounded by an upper value of around 250. Showing that as training

progresses the model is more certain about the actions that it takes. Viewing a replay of the gameplay shows that the agent in some scenarios gets stuck in certain positions which is probably a result of a lack of training.

5. Conclusion

The field of reinforcement learning is gaining significantly more attention with the slow cool down of other field in artificial intelligence. Reinforcement learning shows the ability to generalize as has been shown in this work. The focus of this work was to explore reinforcement learning and that very capability of generalization which other forms of deep learning have not been able to deliver. The scope of reinforcement learning is much larger than was shown in this work. From object detection in images to self-autonomous driving cars. In the scope of this work we implemented a deep recurrent reinforcement learning agent using Tensorflow. The model was evaluated in two different Doom scenarios and Pacman. The models show some good results whilst taking into account the lack of resources at hand.

In future work it would be interesting to see GAN's implemented with a reinforcement agent thereby inserting some form of intuition to the agent; similar to what us humans have. Being able to follow a "gut feeling" as humans would say rather than just following what is always learned to be correct could perhaps result in much greater performance. Results could also be much greater in scale if the model was trained for longer and with better resources. Better resources would enable better hyperparameter searches, faster training.

Literature

[1] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. 2018.

[2] Ian Goodfellow, Yoshua Bengio and Aaron Courville. Deep Learning. 2016. URL http://www.deeplearningbook.org/

[3] Bojana Dalbelo Bašić, Jan Šnajder. 1. Uvod u strojno učenje. 2016. URL https://www.fer.unizg.hr/_download/repository/SU-1-Uvod%5B1%5D.pdf

[4] Serena Yeung Fei-Fei Li, Justin Johnson. Cs231n: Convolutional neural networks for visual recognition. 2016. URL http://cs231n.stanford.edu/

[5] Sergey Ioffe, Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. 2015. URL https://arxiv.org/abs/1502.03167

[6] ujjwalkarn. An Intuitive Explanation of Convolutional Neural Networks. 2016. URL http://jmlr.org/papers/v15/srivastava14a.html

[7] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas. Dueling Network Architectures for Deep Reinforcement Learning. 2016. URL https://arxiv.org/abs/1511.06581

Summary

Reinforcement learning in simulated systems

In this thesis we focused on implementing a reinforcement agent that was able to generalize to various simulated systems and in doing so; showing the ability of the reinforcement learning algorithm to adapt. Fundamentals of deep learning were also described. Using Tensorflow, a reinforcement learning agent was implemented. The agent was evaluated on three different scenarios: Pacman from the Atari 2600 games and two different scenarios of the game Doom. The evaluation results were shown via graphs.

Key words: Reinforcement learning, agent, environment, deep neural networks, Q values, state values, experience replay, recurrent networks.

Sažetak

Primjena podržanog učenja u simuliranim sustavima

U ovome radu smo se fokusirali na implementaciji podržanoga agenta koji je bio sposoban se adaptirati u razne simulirane sustave i tim putem predstavili kapacitet i moć podržanog učenja. Osnove dubokog učenja su objašnjene u ovome radu. Koristeći se tensorflow-om smo implementirali podržanoga agenta. Agent je evaluiran kroz tri različita simulirana scenarija. Pacman iz Atari 2600 i dva scenarija iz igrice zvane "Doom". Sve evaluacije su grafički prikazane.

Ključne riječi: Podržano učenje, agent, okruženje, duboko učenje, Q vrijednosti, vrijednosti stanja, ponavljanje iskustva, povratne mreže.