Universitatea POLITEHNICA din București

Facultatea de Automatică și Calculatoare,
Departamentul de Calculatoare

LUCRARE DE DIPLOMĂ

Deep Learning în jocuri

Conducător Științific:
As. Drd. Ing. Tudor Berariu

Autor:
Florentina-Ștefania Bratiloveanu

București, 2015

University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Playing games with Deep Learning

Scientific Adviser:
As. Drd. Ing. Tudor Berariu

Author:
Florentina-Ștefania Bratiloveanu

Bucharest, 2015

Abstract

Deep Learning is a new, interesting and a fast-growing field of Machine Learning. It combines the classical model of multilayer perceptrons with layers of feature extraction inspired from visual cortex of the brain. Moreover, dealing with the curse of dimensionality is another advantage when we talk about Convolutional Neural Networks. This paper proposes an alternative to the classical reinforcement learning techniques, such as Q-Learning and SARSA. The question which arises is this: what if we make an agent and allow it to play a game based on information from visual frames and rewards recieved during the game?

Contents

Chapter 1

Introduction

1.1 Motivation

Human beings have always felt the need to explore and find a way to a better life. Starting with stone tools (2.6 million years ago) and continuing with personal computers, one can clearly see widespread proofs of human evolution. In the latter years, humanity has discovered Machine Learning which has the purpose to allow machines learn to do different tasks.

Deep learning is a subfield of the machine learning area, drawing its inspiration from the human brain's functionality. Through deep learning, we are trying to solve some of the most pressuring human problems such as cancer classification (benign or malignant tumor)[1], self-driving cars based on pedestrian detection[2] or recognizing digits form photos taken with Google Street View[3], all using unsupervised learning of features.

This paper's proposes an in-depth look on solving reinforcement learning problems, using convolutional neural networks. More exactly, we want to design an agent which is capable of learning to play a game[4] seeing only the pixels from the frames and being rewarded after finishing the game.

This ideea was brought to attention in 2013, when Alex Graves et al. from DeepMind[1] came with the idea of creating a convolutional neural network capable of playing different Atari 2600 games, being almost as good as a game tester. We will see later what almost means. The motivation behind this experiment is not only connected with the fact that we have a machine capable of playing games, but the more important problem is that they achieved the generalization of a machine that could learn several types of games, which in fact is the starting of a new revolution in machine learning. If we are capable of making one unique algorithm that can solve multiple tasks, we are capable of simulating the real human brain and capable of reducing complex problems to others, much simpler thus we can reduce the programming burden and concentrate on solving the more urgent problems such as cancer classification.

Moreover, there are also other types of applications that can reveal the power of convolutional neural networks and one of them is the neural algorithm for creating paintings[5] from ordinary pictures. The inputs is represented by two pictures, one which is the painting (**e.g** *The Starry Night* of Vincent van Gogh) and the other one which is a normal photo, like a picture of a house. The output of the network is the house painted in van Gogh's technique.

[1] http://deepmind.com/

1.2 Project description

This paper presents two different algorithms aiding the creation of a good agent which can learn to play games. It is worth mentioning that the agent does not know the rules and can not infere most of the rules at the beginning of the game. The whole topic has been split into three parts: running Q-Learning on Hanoi Towers game, learning values predicted by Q-Learning using a convolutional neural network and binding the first two parts together.

First of them is the classical Q-Learning, which searches for the optimal policy for taking actions in each state of the game. We will discus about the exploration vs exploitation problem, how to determine the learning rate suitable for our purposes, how many episodes we have to play until we determine a policy close to the optimal one.

The second topic we approach in this paper is the possibility of predicting a continuous output from an image. Practically, we take the dataset from Q-Learning which contains images as input and a table of four values corresponding to the four possible actions in the Hanoi Tower games: up, down, left and right. Here, we can determine whether we have a good or a bad convolutional neural network. In other words, we will test the capacity of learning samples, the capacity for generalization.

One must pay attention to the data pre-processing, this being an important step in data selection. For example, when we learn to classify things, we may not need the color information. Converting data from RGB to YUV gives us the possibility of separating luminance from chrominance.

After that we need to look closer at the model. We need a model that can be proven to converge at the optimal solution and also a model that can learn very fast. Datasets are very big and can contain millions of pictures.

It is also important that one knows when to stop training to avoid overfitting, our primary target being that of minimizing the test error. We have to take care of splitting the dataset in training, testing or validation, how to determine when to stop training the network and how to formalize the results afterwards.

Another important step is called the loss function, the moment when we determine how far we are from our target. Here, we can find the border between underfitting and overfitting, called the optimum state. If we have high bias, it means that we have underfitting and our classifier is not able to predict data well and if we have high variance, it means that we have solid training samples for learning, but new samples cannot be predicted.

The last part is the one where we combine the first two modules. We will try to get a closer look on what those from DeepMind have done with the Atari paper[4]. The most important part is the connection between the Q-Learning algorithm and the convolutional neural network. We will see how weights are updated according to rewards received during the game.

The main problem is that of creating an architecture capable of playing any game without adapting initial meta parameters. All the above-mentioned problems will be discussed in this aper and formalized using experimental results, plots or anything that can serve as a proof.

1.3 Technologies

All algorithms described in this paper have been implemented using Torch7[1], a deep learning framework written in Lua[2], a scripting language based on a C API. For creating the game, generating frames or modifying images the LOVE platform[3] has been used, this also being written in Lua. For interactive computing(image/filter visualization) iTorch[4] is preferred. For illustrating different functions Octave[5] is the best open-source choice.

Why Lua and Torch? They provide a fast environment as opposed to others[6], multiple modules with functions already implemented such as transfer functions(tanh, sigmoid), loss functions(Mean Squared Error, Negative Log Likelihood) or convolutional layers(SpatialMaxPooling, SpatialSubSampling)

1.4 Structure of this paper

Chapter 2 **State of the art** presents the entire architecture of a network based on a complex research in the machine learning, pattern recognition and reinforcement learning fields.

Chapter 3 **Related work** presents an in-depth look into the "Human-level control through deep reinforcement learning" paper is provided. This paper has been chosen for the fact that it is the primary resource which proves that the combination of deep learning with reinforcement learning is possible.

Chapter 4 **System design and implementation** describes several architectures using convolutional and pooling layers which may lead to learning. Every step in choosing the architecture is well documented, either with mathematical reasoning or with empirical reasoning.

Chapter 5 **Results** presents error plots provided by training and testing convolutional neural networks.

Chapter 6 **Conclusions** is dedicated to the personal views of the paper's author regarding this topic and also presents the systematic investigation in gathering information.

Chapter 7 **Future work** describes future tunings of the architectures and also further investigation on one of the following fields: machine learning, pattern recognition or reinforcement learning.

[1] http://torch.ch/
[2] http://www.lua.org/about.html
[3] https://love2d.org/wiki/Main_Page
[4] https://github.com/facebook/iTorch
[5] https://www.gnu.org/software/octave/

Chapter 2

State of the art

Being a large domain, deep learning imposes a research into multiple subdomains such as neuroscience, reinforcement learning, neural networks or convolutional neural networks. That being said, this chapter is dedicated to gathering information on each of them.

2.1 Reinforcement learning

2.1.1 History

The history of reinforcement learning starts with the pavlovian experiment[7] in 1903, when Pavlov tried to demonstrate that there is a connection between conditioned and unconditioned stimulus. The unconditioned response of salivation of a dog is fired up by bringing an unconditioned stimulus such as food. If we try to make the dog salivate in the presence of a neutral stimulus(the sound of a bell) we won't succeed, but if we use at the same time the unconditioned and neutral stimulus we can make the dog salivate and transform the bell and salivation into a conditioned stimulus and a conditioned response.

2.1.2 Q-Learning. Sarsa

Q-Learning

Q-Learning[8] is the algorithm for learning details about the environment in small steps. We know the initial starting state of the game and also a set of possible actions we can take in order to change the current state. At the beginning, we are constrained to choose randomly an action, because we do not know which one could lead us to win the game and this is the exploration phase. We keep randomly picking actions until the game is finished.

A game can have **n** number of states and **a** number of actions. We keep in memory a table **Q** of size **nxa** and we initialize all the corresponding elements with zero. During the game, rewards can be granted. Some of them can be good rewards and some of them can be bad rewards. When we receive rewards, we update the previous state with the obtained score.

Here we have to choose between exploration and exploitation. In the first states of the game, when we know nothing about the environment, we choose to explore new states in order to see if they bring us rewards. After a number of **e** episodes we learn which actions can bring us closer to the winning of the game. This is the moment when we start exploiting the things we have already learnt in the previous episodes. Assume we are in state **s** and have the next

possible actions $\{a_1, a_2, a_3, a_4\}$. Keep in mind that we recorded the Q-table. At this moment we choose the action which gives us the best utility from states s. If $\mathbf{Q(s,}a_3\mathbf{)}$ is the maxim value of **Q** in state **s** we will pick action a_3.

Sarsa

Q-Learning and Sarsa look alike except for the fact that Sarsa is updating its policy based on the action it takes which is not necessary the best action, but it is the action expected to pursue its policy.

When dealing with large negative rewards Q-Learning behaves better than Sarsa[9] because Q-Learning does not avoid risk situations while Sarsa does. Both algorithms have variables that have to be adjusted during the computation.

- **Learning rate** adjusts learning of new information flow. **0** is for not acquiring information and **1** is for considering only the new information.
- **Discount factor** adjust the importance of current rewards(**1**) vs future rewards(**0**)

ϵ-Greedy is the policy for choosing the next action. As we have already mentioned in the problem between choosing explorations vs. exploitation, this strategy is helpful when applied in the first few episodes when we don't have too many details about the environment. It says that we can choose randomly an action from the set of actions with probability ϵ and choose the best action with probability $1 - \epsilon$.

Figure 2.1 shows behaviour of Q-Learning and Sarsa

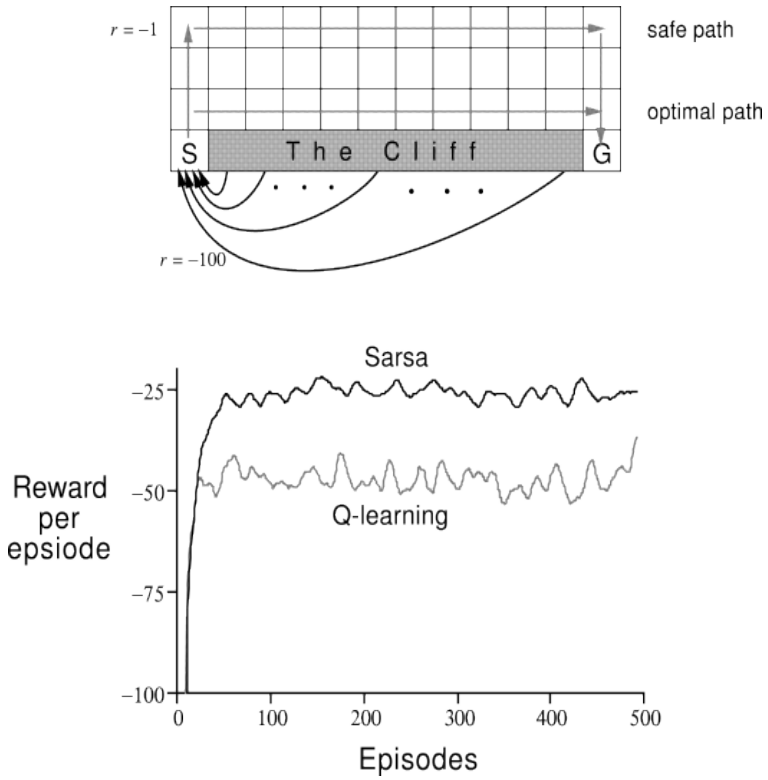

Figure 2.1: Q-Learning vs Sarsa[10]

2.2 Neural Networks and Convolutional Neural Networks

In this chapter we will discuss how networks works, the principles behind them and what optimization techniques we can use. It is worth mentioning that the information is splitted in data, model, loss functions, train and test.

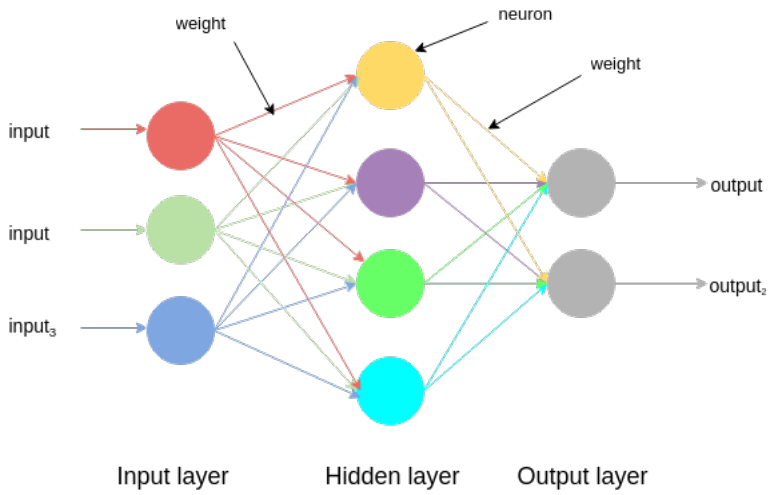

Figure 2.2: example of neural network

2.2.1 Data

The data resulted from the pre-processing step is crucial for learning. Differentiating between noisy images and/or bad lighting is a big problem for a network because it may learn the inappropriate features.

One of the first concerns is choosing the color space. We can have RGB(Red Green Blue) or we can have YUV(luminance blue–luminance red–luminance). In many cases RGB is the first choice instead of YUV[11]. If we are interested in color both of them are a good choice, but if we are interested in edges the luminance channel will be more suitable.

Figure 2.3: Lena: RGB and Y, U, V channels

If values have different ranges of scales, then logarithmic normalization should be used. In distance-based classification, if we have a feature with values between [0, 1] and another feature with values between [0, 2000] we need to normalize the data between [0, 1] or [-1, 1] such that every variation of each feature counts the same as the other features. We do not want to start by giving a higher level importance to one feature to the detriment the other ones.

Another aspect that should be considered is the importance of illumination conditions. If we are not interested in it then we remove the mean-value for each samples. If we have thousands of pictures with different illumination conditions we remove the average illumination by extracting the mean-value such that all the pictures have the same contrast.

Splitting data is another important aspect in training networks. We have to have three sets: train sets, test sets and validation sets. 50% of initial data goes to training and the rest of fifteen percent is split in 25% for testing and 25% for validation. This is used when the dataset is very large. Having a small dataset implies the statetgy of splitting data into $\frac{2}{3}$ for training and $\frac{1}{3}$ for testing[12].

Cross validation[13] should also be taken into consideration. Cross validation is the algorithm of splitting data into test dataset and train dataset forcing both sets cross-over in successive epochs such that every part in the initial dataset validates the model of the neural network. Assume that we have split data into {**A**, **B**, **C**}, each one being $\frac{1}{3}$ part of initial dataset. In the first iteration we take the model to train on **A** and **B** and test it with **C**. The second iteration implies the training on **A** and **C** and testing on **B** and the last iteration implies training on **B** and **C** and testing on **A**. In this way one can be assured that our model is working well for every case.

2.2.2 Model

Nonlinear activation functions[14, 15, 16]

Activation functions are used to define a connection between input and output.

Hyperbolic tangent function(tanh)

- outputs values between 0 and 1
- outputs are zero-centred
- if extremely large negative numbers are given as input, the output will be close to -1 and the error will propagate correctly
- generally used in hidden layers
- tanh layers learn faster than sigmoid layers because of the size of the gradient

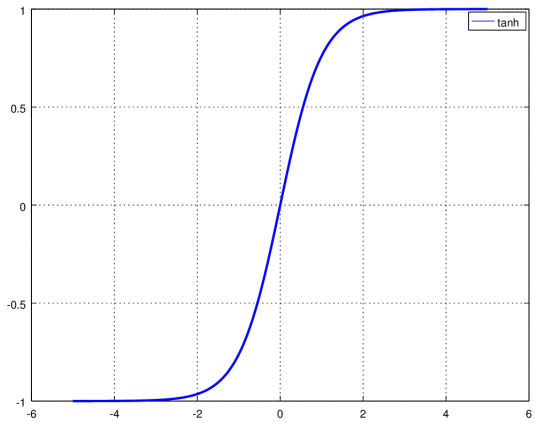

Figure 2.4: tanh function

Sigmoid function(sigmoid)

- learning values farther away from the threshold is difficult
- useful for backpropagation algorithms
- easy for calculating derivatives and used in gradient descent methods
- output values between 0 and 1
- can be used in binary classification problems with setting some thresholds
- if extremely large negative numbers are given as input, the output will be closer to zero and the error will not propagate correctly

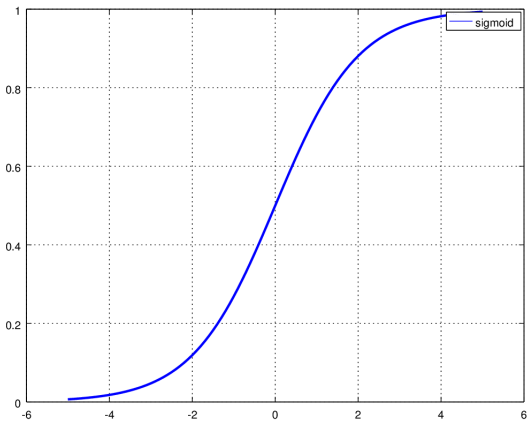

Figure 2.5: sigmoid function

Rectified Linear Units(ReLU)

- better results in networks combined with stochastic gradient descent than tanh/sigmoid[17]
- ReLU is determined to be better than tanh/sigmoid empirically

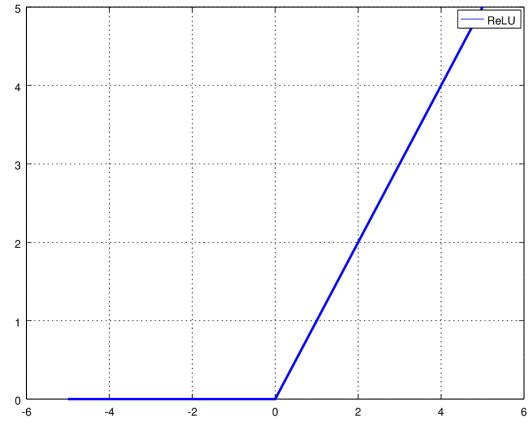

Figure 2.6: ReLU

Convolutional layers

Pooling layers can be used when getting rid of unwanted information and also can be useful in dimensionality reduction.

Spatial Convolution

- used to modify the frequency of neighbourhood pixels
- accentuates edges in combination with Gabor filters
- can be used to obtain gaussian blur effect
- suppose we have an image of **width x height** size, if we apply a convolution kernel of size **kw x kh** the size of the output will become **(width - kw + 1) x (height - kh + 1)**

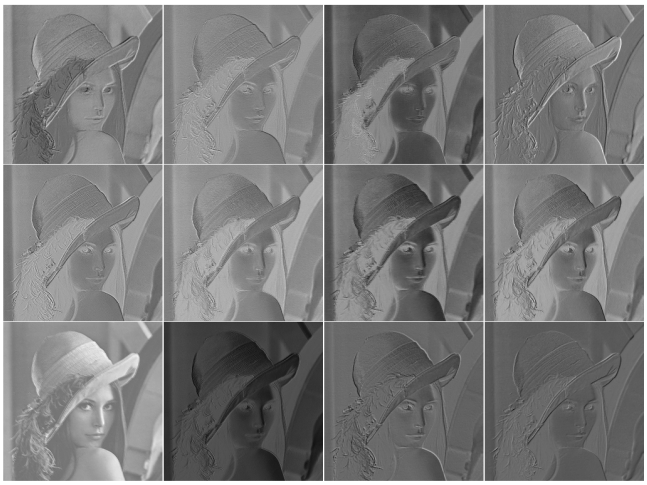

Figure 2.7: Lena: Spatial Convolution(2D) with 12 output channels, 5x5 kernel matrix and 2x2 stride

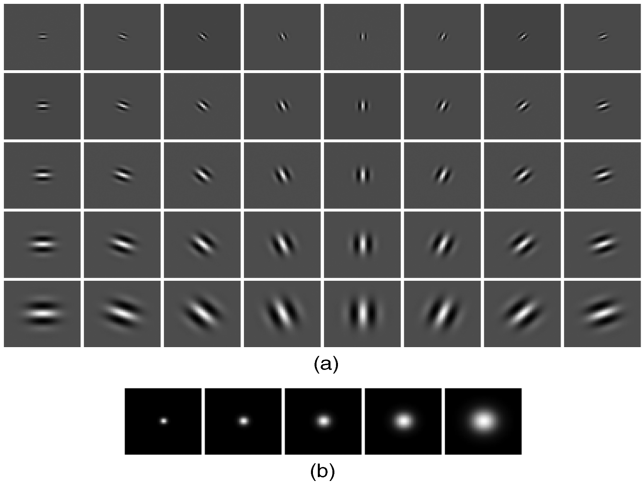

Figure 2.8: Gabor filters[18]

Average pooling

- useful for dimensionality reduction
- acts poorly in some situations as opposed to max-pooling

Figure 2.9: Lena: Average pooling with 10x10 kernel matrix and 2x2 stride

Max pooling

- useful for dimensionality reduction
- useful for activating features that have a low probability of being activated[19]
- if a feature appears in a cell it is less probably to appear in the neighbourhood cells

Figure 2.10: Lena: Max pooling with 5x5 kernel matrix and 2x2 stride

LP pooling

- the norm P=1 represents gaussian averaging and P=∞ represents max pooling
- computes the information from neighbourhood if P>1

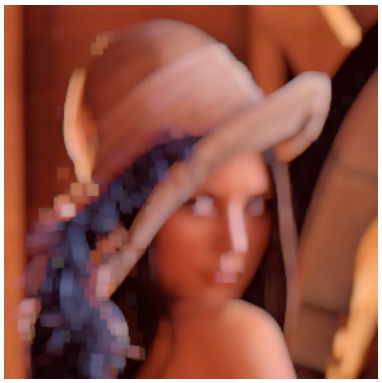

Figure 2.11: Lena: LP pooling with pnorm = 10, 15x15 kernel matrix and 2.5x2.5 stride

Spatial Substractive Normalization

- the norm P=1 represents gaussian averaging and P=∞ represents max pooling
- computes the information from neighbourhood if P>1

Figure 2.12: Lena: Spatial Substractive Normalization pooling with kernel size 11x1

Feature visualization

The technique called Deconvolutional Networks takes a neural network already trained and maps the features to pixels such that information from features learned can be human "readable".

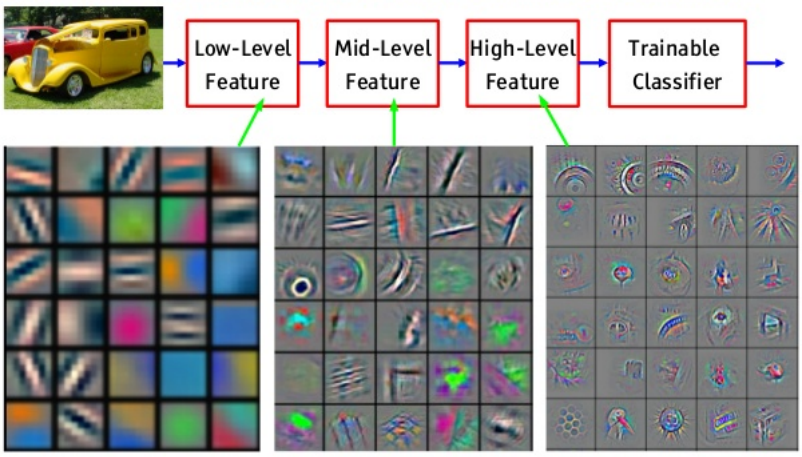

Figure 2.13: Feature visualization[20]

2.2.3 Loss functions

Loss functions are used to approximate how far we are from the solution, the difference between observed values and predicted values.

Mean squared error

- useful for regression
- not used in classification problems

$$J(weights) = \frac{1}{2 \cdot no_{samples}} \cdot \sum_{i=1}^{no_{samples}} \cdot (nn_{weights}(samples^{(i)}) - labels^{(i)})^2 \tag{2.1}$$

Hinge loss (margin criterion)

- useful for binary classifications
- forces labels to be -1 or 1

$$J(weights) = \sum_{i=1}^{no_{samples}} max(0, 1 - labels^{(i)} \cdot nn_{weights}(samples^{(i)})) \tag{2.2}$$

2.2.4 Optimizations for learning

Optimization algorithms are used in updating the weights in a neural network in concordance with a loss function. Assuming that our cost function is defined as follows[21]:

$$J(weights_n) = \frac{1}{2no_{samples}} \cdot \sum_{i=1}^{no_{samples}} (nn_{weights}(samples^{(i)}) - labels^{(i)})^2 \tag{2.3}$$

we state that the derivative of **J** w.r.t **weights** is[21]:

$$\frac{\partial J}{\partial weights} = \frac{1}{no_{samples}} \cdot \sum_{i=1}^{no_{samples}} (nn_{weights}(samples^{(i)}) - labels^{(i)}) \tag{2.4}$$

Updating weights[21] is done in accordance with the derivative of cost function.

$$weights_n = weights_n - \alpha \cdot \frac{\partial J}{\partial weights} \tag{2.5}$$

where α represents learning rate.

Gradient descent (GD)

- if α is too small, GD converges slow
- if α is too large, GD can converge to local minimum and not global minimum
- recommended for huge number of samples
- require more iterations than SGD

Algorithm 1 Gradient Descent[21]

1: **repeat**
2: **for** `j = 1, no_weights` **do**
3:

$$weights_j = weights_j - \alpha \sum_{i=1}^{no_{samples}} (nn_{weights}(samples^{(i)}) - labels^{(i)}) \cdot samples_j{}^{(i)} \quad (2.6)$$

4: **end for**
5: **until** convergence is obtained

Stochastic gradient descent (SGD)

- very fast is number of samples is small
- iterations are removed by computing $(X^T \cdot X)^{-1}$
- requires regularization
- needs feature scaling

Algorithm 2 Stochastic Gradient Descent[21]

1: **repeat**
2: shuffle dataset (randomly)
3: **for** `i = 1, no_samples` **do**
4: **for** `j = 1, no_weights` **do**
5:

$$weights_j = weights_j - \alpha \cdot (nn_{weights}(samples^{(i)}) - labels^{(i)}) \cdot samples_j{}^{(i)} \quad (2.7)$$

6: **end for**
7: **end for**
8: **until** convergence is obtained

2.2.5 Training and testing

The question which arises now is: when do we have to stop with training if a (convolutional) neural network ? We have to take care of how much we train a network. If we train it too much, it will learn the samples and will predict exactly the same values, but there will be a large error for samples it has not seen. If we train it too little, the network will not learn how to predict the samples it has seen and will also have a large test error. Mathematically, the moment we should stop with training is the moment when the test error will increase its value. In 2.14 we can see the optimum moment when we should stop training.

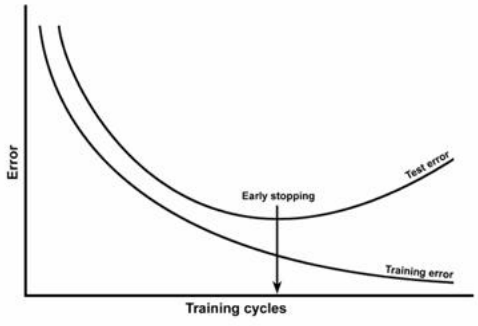

Figure 2.14: train-test error[22]

Underfitting also means high bias and represents the incapacity of the algorithm to recognize the pattern features in samples.

Overfitting means also high variance and represents the inability for the algorithm to adapt to new samples.

Both of them are illustrated in the figure 2.15.

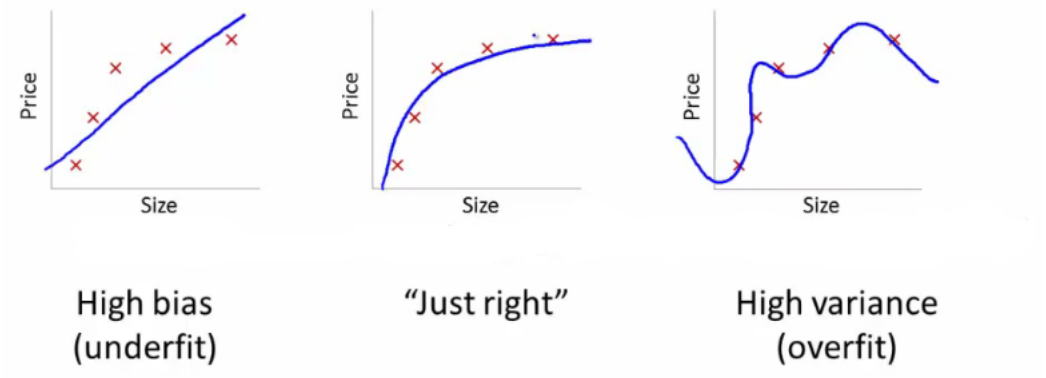

Figure 2.15: underfitting, bias-variance trade-off and overfitting[21]

Chapter 3

Related work

3.1 An in-depth look to "Human-level control through deep reinforcement learning" paper

This section is dedicated to analyzing the paper which is proposes a method for solving games using deep learning and reinforcement learning. The proposed agent, called Q-Network capable of playing 43 different games and adapting to each one without being modified.

Figure 3.1: Snapshots with five Atari 2600 Games: Pong, Breakout, Space Invaders, Seaquest, Beam Rider[23]

3.2 Architecture

The article[23] proposes a new type of network, Q-network which combines convolutional layers, a representation of the human receptive fields and pooling layers used for dimensionality reduction. The input layer is represented by **84x84x4** neurons for the image produced after preprocessing frames from the game and is followed by three convolutional layers(32 filters of 8x8 and stride 4 with ReLU, 64 filters of 4x4 and stride 2 with ReLU, 64 filters of 3x3 and stride 1 with ReLU), two fully connected layers: a hidden layer with 512 units and the output layer which has a number of neurons equal to the number of actions corresponding to each game. Each hidden layer is followed by a rectified linear unit (ReLU).

The network is used to approximate the optimal action-value function Q^*[23]:

$$Q^*(s,a) = \max_{\pi} E[r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \ldots | s_t = s, a_t = a, \pi] \tag{3.1}$$

The function Q is also parametrized with the weights and the agent state $e_t = (s_t, a_t, r_t, s_{t+1})$ is saved at each iteration in a set D_t $= (e_1, e_2, \ldots, e_t)$ With all the necessary information here is the computation of the loss function[23]:

$$L_i(\theta_i) = E_{(s,a,r,s')}[(r + \gamma \cdot \max_{a'} \cdot Q(s', a'; \theta_i^-) - Q(s, a; \theta_i))^2] \tag{3.2}$$

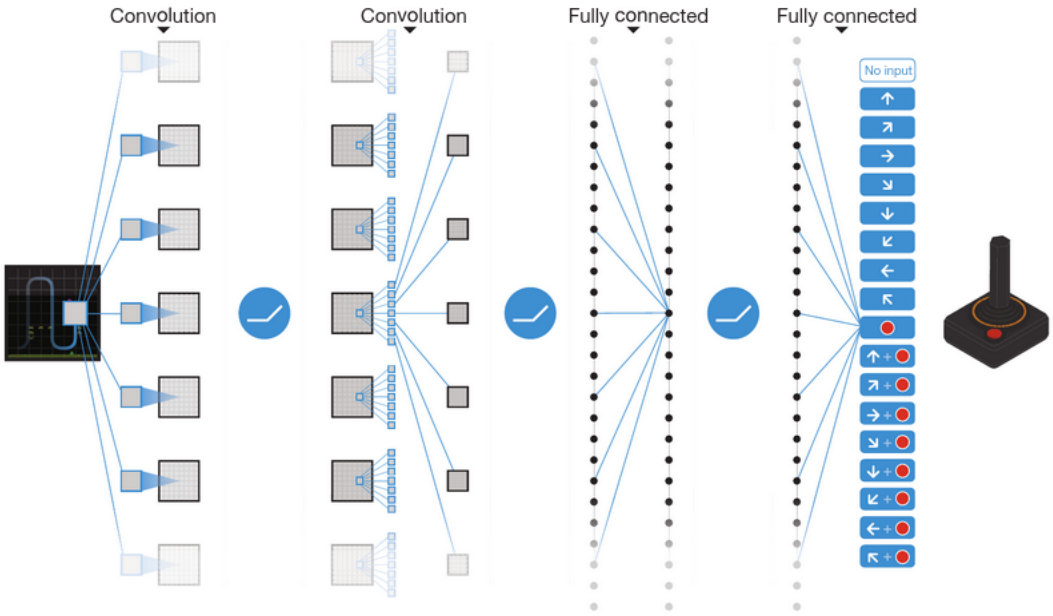

Figure 3.2: Q-Network architecture[23]

where γ represents the discount factor and ${\theta_i}^-$) is updated with θ_i every x defined steps and here is the gradient[23]:

$$\nabla_{\theta_i} L(\theta_i) = E_{s,a,r,s^i}[(r + \gamma \cdot \max_{a'} Q(s', a'; {\theta_i}^-) - Q(s, a; \theta_i))\nabla_{\theta_i} Q(s, a; \theta_i)] \tag{3.3}$$

The Q-learning action-value function can be updates using ${\theta_i}^- = \theta_{i-1}$ following the optimal policy with probability 1 - ϵ and selecting a random move with probability ϵ.

For minimizing the loss function, Stochastic Gradient Descent is used in combination with the experience replay technique for preventing 'dead' neurons and also, Q is not updated at each iteration. This avoids the instability of Q caused by the nonlinearity of network.

The ϵ-greedy policy was set to 0.05 and the Q-values were scaled because of big range values from rewards.

3.3 Algorithm

One of the techniques that Q-Network algorithm is using is called experience replay. This implies transitions between states to be stored. A number of N transitions (the previous state, the current state, reward and action) are stored in a pool of transitions. Until the network starts learning, we have to play many episodes. We begin from initializing the pool where we store the experiences and initalize two different networks(action-value network Q and target action-value network Q^-) with the same weights. For each episode we start by storing the current transition which at first will be composed by only one state. With porobability ϵ we choose a random action and with probability $1-\epsilon$ we choose the best action. After that, we apply the action on the current state of the game and observe reward and new state of the game. We put it in the pool and randomize it. The target value is the reward if game is finished or the sum between the reward observed and maxim-value predicted by target action-value Q^- multiplied with the discount dactor. Then we propagate the error as the mean squared error between target value and action-value function Q. Once **x** steps had pass we update the weights of Q^- with the weights of Q. The algorithm[4] used for training Q-Network is presented as it follows:

Algorithm 3 Q-Network

create replay memory D for storing N experiences
choose ϵ between (0,1)
init Q model weights with θ
save Q^- model weights to θ^-
init variable episode to 0 and choose MAX_EPISODES
while episode < MAX_EPISODES **do**
    generate state s_1 from frame: $s_1 = image_1$
    while game not over **do**
        r = random number between (0,1)
        if r < ϵ **then**
            a_t = random action
        else
            $a_t = argmax_a$ Q(s_t, a; θ)
        end if
        $s_{t+1} = (s_t, a_t, image_{t+1})$
        D = D + $\{(s_t, a_t, r_t, s_{t+1})\}$
        shuffle D
        extract (s_j, a_j, r_j, s_{j+1}) from D
        if episode is finished at step j+1 **then**
            $y_j = r_j$
        else
            $y_j = r_j + \gamma \cdot \max_{a'} \cdot Q^-(s_{j+1}, a'; \theta^-)$
        end if
        propagate error $(y_j - Q(s_j, a_j; \theta))^2$
        at each **x** steps save model weights θ to θ^-
    end while
end while

3.4 Results

Q-Network is capable of learning the optimal policy. For example, in Breakout what is being learned is to make a tunnel between blocks in order to send the ball into the back. Receiving only the pixel from the frames, rewards and using the same network structure it is capable of playing different games. The Q-Network can achieve 75% of a score of a human test at 49 games[23]. The games where the algorithm performs poorly are the games where the memory is needed. For example, on Montezuma's Revenge[1] we go from room to another rooms so it is very hard for the algorithm taking into account multiple scenarios changing instead of only one changing. Below there are the results for different types of games from Atari compared to the results of a proffesional human tester.

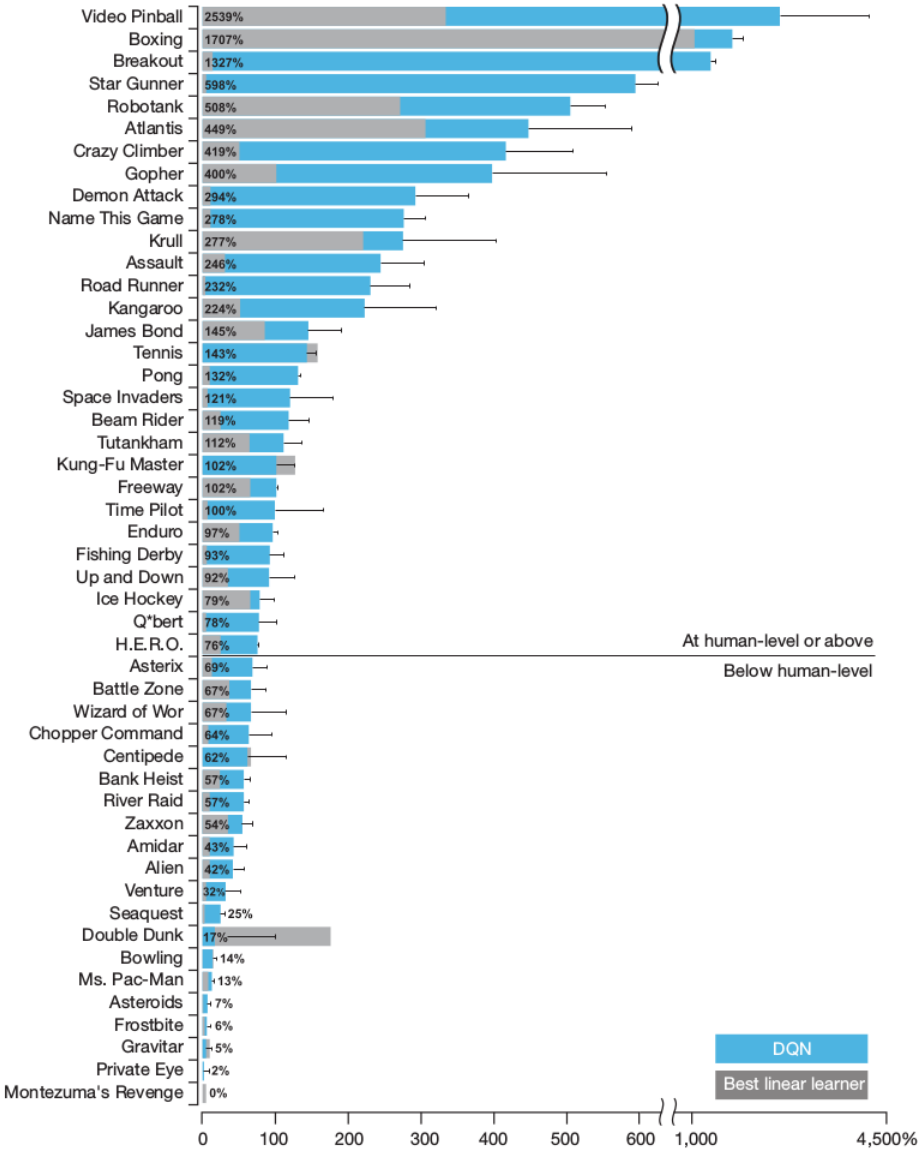

Figure 3.3: Comparision[23]

[1] https://atariage.com/software_page.php?SoftwareID=1158

Chapter 4

System design and implementation

This chapter presents the entire work of the author. The work is split into three parts in the attempt to combine deep neural networks with Q-Learning. The game chosen for the experiment is the Tower of Hanoi due to its simplicity and also, due to the static environment. Afterwards, the values obtained through Q-Learning have been used to train a deep convolution network combined with the generated frames. Finally, our attempt is to combine the Q-learning algorithm with convolution networks.

4.1 Q-Learning

In this section we attempt to find the optimal policy for playing the Tower of Hanoi game. The algorithm starts with a pool of actions initialized with UP, DOWN, LEFT, RIGHT and a table Q containing the values for every (state,action) pair.

The discount factor γ is set to 0.95, the future rewards being taken into account when updating the action-value function. The learning rate α is chosen at 0.1, thus the algorithm can converge to an optimal policy.

The exploration vs. exploitation problem is solved by using a variable ϵ which is initialized with 1 (1 is used when choosing only random actions and 0 when the action that can maximize the score is chosen). In order to make the current policy converge to the optimal policy the number of iterations used is set to 11 and the number of episodes per iteration set to 1000. After each iteration is finished, ϵ it is decreased with 0.1 until it will take the value 0. This method has been used to let the agent explore using random actions in the first iteration, afterwards exploiting the values learned in the last iterations.

After all the iterations are finished, the algorithm generates a dataset. For each value stored in Q-table, it generates the frame from each coded state.

On the next page, a pseudocode of Q-Learning is presented, which is an adaptation after the algorithm found in Norvig's book[8]. The algorithm tries to estimate an optimal action-value function using the ϵ-greedy policy, more exactly, it chooses a random action with probability ϵ and chooses the optimum action with probability 1-ϵ.

Algorithm 4 Q-Network

set size of Q to with no_states x no_action and initialize it
choose ϵ between (0,1)
let **s** be the current state of the game
let **actions** be a pool with a_1,a_2,...,a_n
while game not over **do**
    r = random number between (0,1)
    if r $<$ ϵ **then**
        a_t = random action
    else
        $a_t = argmax_a$ Q(s, a)
    end if
    s', r = apply_action(a, s)
    Q(s,a) = Q(s,a) + $\alpha \cdot$(r + $\gamma \cdot \max_a Q(s', a') - Q(s, a)$)
end while

4.2 Convolutional Neural Networks

This section comes with an attempt to discuss various architecture models that could be suitable for our problem. To see if the network fits the main theme of this paper, one must provide proof that the network model can be generalized and is not hitting the "high variance problem". In order to come with a scientific proof or at least an empyrical one, several experiments have been made.

4.2.1 Regression with deep-learning and complex model

The model proposed by Yann Lecun et al.[24] was slightly modified in order to map our problem. For the purpose of keeping things clear and well-separated every discussion about a network should be split into the next subjects: preprocessing data, chosen model for the network, loss function, training and testing.

For the first attempt, a model used in face detection and pose estimation has been chosen because good results in learning features relevant for object recognition have been promised. Also, it is possible for this model to be applied over large images.

Data

In this section 4.1, we generated a dataset of 160 images with 4 labels for each one (Tower of Hanoi has 160 states and offers 4 possible actions). In this section we try to build a convolutional neural network capable of predicting those values from images. This is made for the sake of bringing strong proof that our network model is capable of learning features from the dataset and not leading, instead, to a "dead" network which is "overlearning" and cannot be used on new samples.

First step was to resolve the problem of the small number of samples (160) from the dataset. This has been solved by generating 64 other datasets (to achieve approximately 10,000 samples) from the original one with noise added or color changed. In this way we can ensure that the network is capable of learning features dependent on size in this case.

The next figure(4.1) represents a state of the game and the next ones(4.2) represent altered images with noise added and color changed.

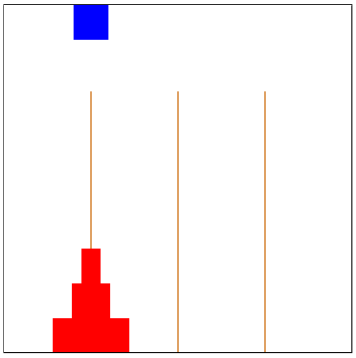

Figure 4.1: Initial state of the game "Tower of Hanoi"

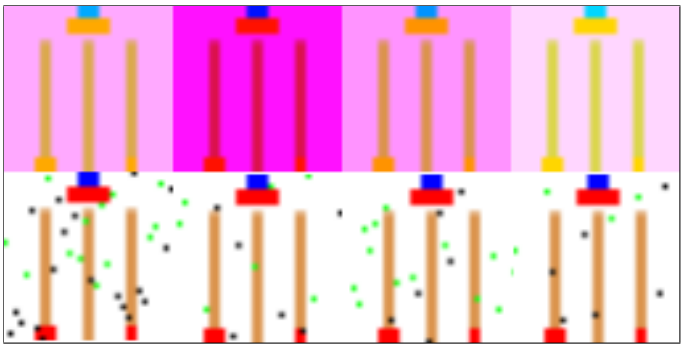

Figure 4.2: Alterated dataset

For the preprocessing data various problems have been encountered. First of all, there is a large discussion on the RGB vs. YUV topic. This paper implements the YUV transformation based on the assumption that luminance should be separated from chrominance. If features dependent on color should be learned, then the neurons connected with the chrominance pixels will be activated, otherwise the neurons connected with luminance pixels will be activated. Both types of channels are kept in the attempt to make an algorithm that could make a better generalization. Also, the YUV color is much closer to the human vision than RGB.

The decision not to work with the 1-channel colors spaces such as grayscale is well-founded. There are games where size should not be a feature to be learned and in this cases we are interested in learning features dependent on color.

The next step was to normalize the images for aligning the range of pixel intensity values to a normal distribution. In other words, the histogram of the images are stretched and the contrast is improved in order to get sharp edges.

The normalization of the images implies computing the mean and the standard deviation. These values are computed for the training dataset of each channel. Then, we add the mean and divide it by standard deviation. This operation is applied on each channel from both datasets (train and test).

Another way to improve the contrast is using a gaussian matrix which follows a gaussian distribution, as it is clear from its name. The gaussian distribution states that average things should occur frequently and extreme things should occur rarely. The gaussian matrix will slide all over the image and will modify the pixel values such that the pixels from the middle of the neighbourhood will be more important than the pixel values from the edge of the gaussian matrix. If the edge pixel values have a zero importance than the resulting image will be identical to the initial image.

On the other hand, labels also need to be processed. Because of the formula for estimating the action-value function (Q) different ranges of values were generated. For example, if we have to do regression on values as 0,..., 100 there would not be a problem because the delta range is 10^2, but the values generated follow a delta range of 10^5. This can cause instability in the network. These being said, the paper proposes to bring the values to the same range using a logarithmic function and of course, to normalize them to follow the normal distribution.

Data was also shuffled to avoid having consecutive states of the game.

Model

The model used for training the convolutional neural network is the model presented in the next paper: "Synergistic Face Detection and Pose Estimation with Energy-Based Models"[24] with some modifications such that the model becomes suitable for our network.

The model follows another model called LeNet-5 proposed by Yann LeCun et al. which was used in document recognition[25] for training a network that could recognize noisy handwritten characters. The results obtained by the network state that the error rate is small, under 1% which is extremely satisfying. Trained on noisy handwritten characters draws attention to one important aspect: if the model is capable of generalizing even in heavy conditions, why could it not be used in combination with Q-learning for playing games.

In the next figure(4.3) we can see how the network suggested in this article looks.

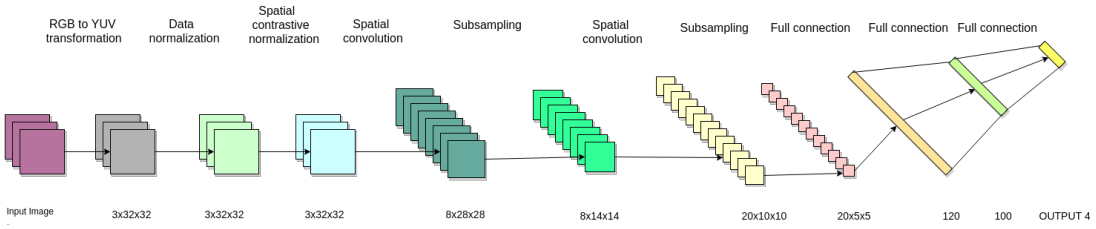

Figure 4.3: ConvNet Architecture

The network is composed of three layers. Each layer applies kernel convolutions and samples on the data. After each layer we use the hyperbolic tangent as the layer activation function. As discussed in the paper "Efficient Backprop" by Yann LeCun et al., the motivation for using the hyperbolic tanh is similar to the one of normalizing data. The mean of the output data tends to be closer to zero, thus it is less likely to cause instabilities in the network derivatives when used in combination with back-propagation.

After the final layer, the sigmoid activation function has been used because the labels are normalized by scaling between 0 and 1.

The input layer consists of no_features x no_pixels_width x no_pixels_height neurons and the output layer has 4 neurons because only 4 actions are needed to play Tower of Hanoi.

The last layers are fully connected and for hidden layers the Spatial Convolution and Sub Sampling have been used. In the appendix A.1 the implementation in Torch7 can be found.

Loss function

For computing the error, Mean Squared Error was used. Being a task of regression, and not a task of classification the MSE is most appropriate for use.

$$loss(target, output) = \frac{1}{no_{samples}} \cdot \sum_{i=1}^{no_{samples}} \cdot((output^{(i)}) - target^{(i)})^2 \tag{4.1}$$

4.2.2 Classification with deep-learning and complex model

Because the last results for doing regression with convolutional networks did not match the author's expectations, another option have been provided. What if we try to learn the same results using a clasificator? All in all, we try to predict the action for every state.

Firstly, we change the architecture from LeNet-5 with the one used for training on Google Street View House Numbers proposed by Yuval Netzer et al. in the "Reading Digits in Natural Images with Unsupervised Feature Learning"[3] paper. The dataset used by them is very similar with the MNIST database (Yann Lecun)[1] except for the fact, that SVHN[2] provides natural scene images.

Data

For predicting classes we need to transform continous values to labels and for each state of the dataset we choose the label (úp≙ 1, d́own≙ 2, ĺeft≙ 3, ŕight≙ 4) according the maximum value. If the value corresponding to the ùp̀-action is the biggest, then we choose label 1. Like the previous model architecture, the data is converted from RGB to YUV color space, to separate the luminance from chrominance. After this step they peform contrast normalization to sharpen edges and also, the normalization is performed.

Model

The model proposed for training the network is slightly different the the other one. The input reamins the same, 3 channels and 32x32 size. The output is formed by four neurons corresponding for each action. The hidden layer activation functions used for the model is also tanh. The network is composed by two convolution layer and two pooling layers.

Loss function

For computing the error, Log Soft Max was used. Being a task of classification, and not a task of regression we need our network to map categorical information (discrete values variables) on continous normalized data.

$$loss(x^{(i)}) = \log \frac{1}{\sum_j e^{x^{(j)}} \cdot e^{x^{(i)}}} \tag{4.2}$$

[1] http://yann.lecun.com/exdb/mnist/
[2] http://ufldl.stanford.edu/housenumbers/

4.2.3 Regression with deep-learning and simple model

Both of the last experiments using convolutional neural networks did not lead to satisfactory results and that is the reason we propose another model. We found that sometimes RGB may lead to better improvements than YUV[11] so we passed the images to CNN into RGB color-space. Another preprocessing step was removing contrast normalization because it does not improve the results, leading to overfitting. The new model comes with two convolutions and two poolings. The output activation functions remain the same, tanh for hidden layers and sigmoid for the output layers in order to predict the values from Q-Learning which are between 0 and 1. We removed one fully connected layer and let 150 neurons on the last hidden layers. An implementation in Torch7 can be found in appendix A.2.

Chapter 5

Results

5.1 Q-Learning

In the next figures the values predicted from Q are presented. As it can be noticed the optimal policy is as follows. The game allows the picker to move LEFT from first stack and move RIGHT from last stack (**e.g.** If the picker points to the first stack and chooses to move LEFT, then the picker will point to the last stack). In fig. 5.1 the picker points to the second stack so the optimal action will be to move RIGHT. As we can see the value for the RIGHT-action is bigger than the other values. In fig. 5.2 the agent has only one move to finish the game, in optimal conditions. Again, if we look at which value is greater than the others we can see that DOWN appears to be 100 which is exactly the value of the reward received when an episode is finished. Fig. 5.3 presents the initial state of the game, just as expected the game should begin with the picking for the first time of a disk and that is exactly what Q-Learning is doing.

Another important aspect is that the values for actions corresponding to a state **s** are close. Why? This happens because the game has a static environment and there is no negative reward, thus whatever happens, the agent will always lose or it will be forced to play until the game is finished. The agent can never loose.

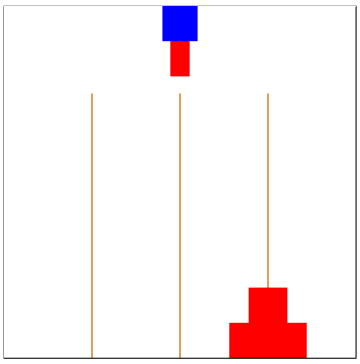

Figure 5.1:
UP = 90,6534
DOWN = 86,8787
LEFT = 89,1867
RIGHT = 94,2824

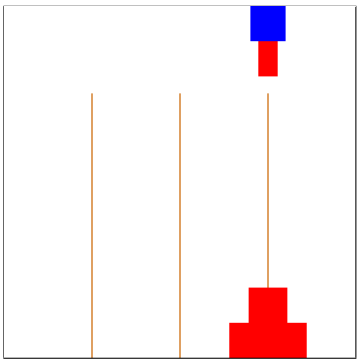

Figure 5.2:
UP = 97,6530
DOWN = 100,0000
LEFT = 93,8538
RIGHT = 92,5261

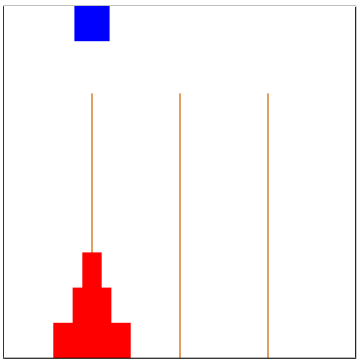

Figure 5.3:
UP = 26,3520
DOWN = 23,8452
LEFT = 23,8897
RIGHT = 22,8827

5.2 Regression with deep-learning and complex model

In the last chapter we discussed the composition of the model. We stated that the last layer is formed of 100 neurons. Sometimes, neural networks are not as easy as we imagine to implement. Even if we have a big dataset, or even if we alter data in a way that we think it might work for the sake of generalization, bad things happen. As we can see from the figures, the network is doing very well at learning samples from the training dataset. It is worth mentioning that one epoch is equivalent to learning the train dataset once and get the error for forwarding the test dataset over the network. Also, to make a parallel, notice that the output values are normalized to the 0-1 range. Now, we can return to the graph and see that the training error behaves as we expect it to and yes, it is decreasing. However, the test error is not behaving mathematically as shown in figure 2.14. This brings into question the validity of the model. One must observe that the slope of the train-line between the first epoch and the second epoch is steeper, which means that after only one epoch the network overlearned the train-dataset and that is one reason for not being predicted well on the test-dataset. In either case, one can state that the model with 50 neurons is an improvement as opposed to the one with 100 neurons. From this we conclude that if the model is too complex, the network could run into the overfitting phenomenon.

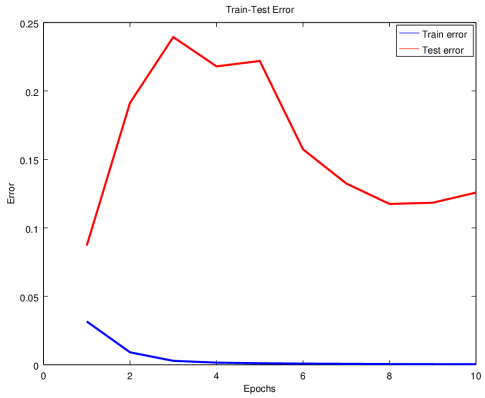

Figure 5.4: Last hidden fully connected layer has 100 neurons

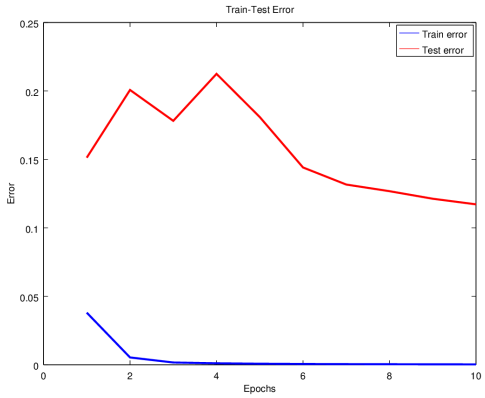

Figure 5.5: Last hidden fully connected layer has 50 neurons

5.3 Classification with deep-learning and complex model

For updating weights in combination with backpropagation, SGD was used with learning rate $\alpha = 10^{-3}$. The train and testing have been done for 10 epochs and the average accuracy computed during the 10 epochs is approximately 33%. The dataset used is the same from regression, 10,000 pictures created from 160 pictures with noise added and color changed. Once again, the experiment showed unsatisfactory results. Transforming qlearning learned values to labels resulted in having unequal number of samples per class. This affects the network to learn more how a picture of class **c** looks and of course, in not learning the samples for the other classes. The graphic below gives the plot of the accuracy on both, train and test datasets.

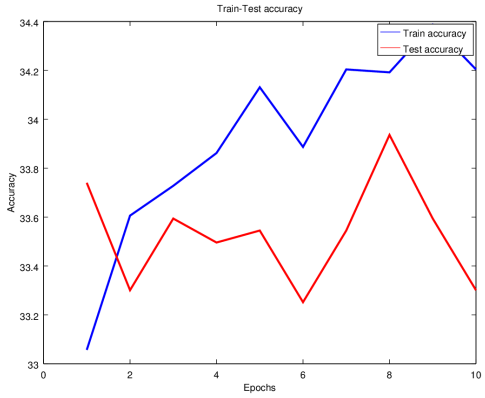

Figure 5.6: Train-Test accuracy - classification

5.4 Regression with deep-learning and simple model

Once again we used SGD with learning rate equals to 0.05. The train and testing has been done 10 epochs. The dataset used is the one with noise and color changed with 10,000 samples. This time, reducing the complexity of the network brings improvement in the results showed by the graphic below. After each epoch, the system is learning and this lead to a decreasing of both errors, for training and testing. While simple model can not learn to predict neither values they have seen nor new samples, complex model may learn very well examples and become incapable to predict for new samples. Both cases of anomalies lead to a neural network that could not be used for predicting anything. The main target is to have an agents that could do very well on unseen samples.

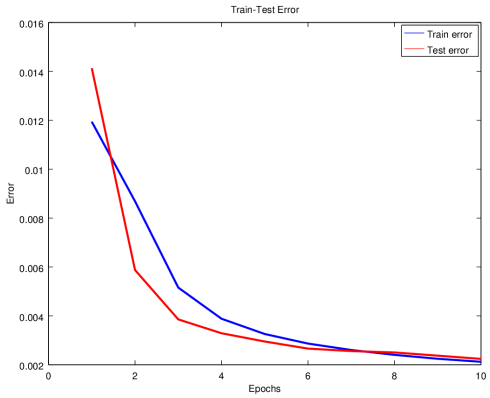

Figure 5.7: Train-Test error - simple model

Chapter 6

Conclusions

In this thesis we have presented the architecture of a convolutional neural network and optimization techniques for a better learning. Deep learning is considered to be more closer to artifficial intelligence, it gives the power to networks to learn features, to synthesize the information raw information as pixel values. We have seen that building a network that could learn very well implies a lot of things to be considered like choosing loss function or how many layers a networks should have or even how to preprocess data. We might have seen that simple model lead to unlearning samples and complex model lead to overfitting. Also, to find the trade-off need multiple experiments in order to achive the optimum model.

Chapter 2 **State of the art** was oriented to gather all necessary information on how the model of architecture should look. Making a parallel between reinforcement learning and convolutional neural networks was the best way to achieve an in-depth overview of the actual algorithms used in deep learning.

Chapter 3 **Related work** This chapter revealed the work of people from DeepMind. It described the implemenation of the whole system and it gave the starting point on the research.

Chapter 4 **System design and implementation** presented the implementation of the algorithms discussed in the previours chapter. Every layer of the network was discussed in order to build a solid argument on each choice from preprocessing data to training and testing. The author of the paper chose to combine the architecture implementation with results chapter for the sake of keeping results of each architecture next to its implementation.

We have seen that deep learning can be used efficiently in many problems, from recognizing handwritten digits to cancer classification. This discover could lead to a new era for artificial intelligence. What if we could have an alternative for preventing accidents or what if we could prevent diseases on time?

Chapter 7

Future work

The results presented in this paper lead to empirical observation when talking about building networks solving different problems. The paper's author proposes new system enhancements based on these experiments.

First of all, the algorithm need to be tested on more complex games where the state of universe is not fully observed(Donkey Kong[1]) by our agent or dynamic environments (Asteroids[2]) where the scenarios keep changing. In the last case, the reasearch should be done on replay memory technique.

Secondly, we have to test the algorithm on real-life sceanrios. NAO is a humanoid robot[3] built by Aldebaran Robotics[4]. In this case we can choose Tic-Tac-Toe game where Nao will play with a human. Every time the game is finished the human will release sounds corresponding on bad rewards and good rewards. Another good part with using Nao is the framework written in Python. This could lead in using another deep learning framework, Theano. Some certain aspect may have been taken into consideration. The network should be trained on different perspectives of the camera, noise reduction should be applied on the frames. Another important aspect is represented by illumination. We could have bad illumination or good illumination. This represents a real life scenarion where the agent will face many difficults and of course the engineer behind the project, but final result may be satisfactory and may lead to a new research idea.

Last but not least, the ideas presented in this paper could lead to development of another project. We could have self-driving cars based on using deep learning in pedestrian detection. Imagine what it would be like to have cars that can see human passing by and avoid accidents.

[1] http://www.atariage.com/software_page.php?SoftwareID=990&LabelID=6
[2] https://atariage.com/manual_html_page.php?SoftwareID=828
[3] http://doc.aldebaran.com/1-14/family/robots/motors_robot_v33.html
[4] https://www.aldebaran.com/en

Appendix A

Code examples

A.1 Overfitting model

```
model = nn.Sequential()

model:add(nn.SpatialConvolutionMM(3, 8, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialSubSampling(8, 2, 2, 2, 2))

model:add(nn.SpatialConvolutionMM(8, 20, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialSubSampling(20, 2, 2, 2, 2))

model:add(nn.SpatialConvolutionMM(20, 120, 5, 5))
model:add(nn.Reshape(120))
model:add(nn.Linear(120, 100))
model:add(nn.Tanh())
model:add(nn.Linear(100, 4))

model:add(nn.Sigmoid())
```

Listing A.1: CNN model which does overfitting

A.2 Optimal model

```
model = nn.Sequential()

model:add(nn.SpatialConvolutionMM(3, 4, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialSubSampling(4, 2, 2, 2, 2))

model:add(nn.SpatialConvolutionMM(4, 6, 5, 5))
model:add(nn.Tanh())
model:add(nn.SpatialSubSampling(6, 2, 2, 2, 2))

model:add(nn.Reshape(150))
model:add(nn.Linear(150, 4))
```

Listing A.2: Optimal model for doing regression with ConvNets

Bibliography

[1] DanC. Cireşan, Alessandro Giusti, LucaM. Gambardella, and Jürgen Schmidhuber. Mitosis detection in breast cancer histology images with deep neural networks. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*, pages 411–418. Springer Berlin Heidelberg, 2013.

[2] Yonglong Tian, Ping Luo, Xiaogang Wang, and Xiaoou Tang. Pedestrian detection aided by deep learning semantic tasks. *CoRR*, 2014.

[3] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y. Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, 2013.

[5] Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge. A Neural Algorithm of Artistic Style, August 2015.

[6] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[7] Ivan Petrovich Pavlov. *Conditioned Reflexes: An Investigation of the Physiological Activity of the Cerebral Cortex*. Oxford University Press: Humphrey Milford, 1927.

[8] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[9] Paul Davidsson, Brian Logan, and Keiki Takadama, editors. *Multi-Agent and Multi-Agent-Based Simulation, Joint Workshop MABS 2004, New York, NY, USA, July 19, 2004, Revised Selected Papers*, Lecture Notes in Computer Science, 2005.

[10] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[11] Pierre Sermanet. A deep learning pipeline for image understanding and acoustic modeling. Technical report, DTIC Document, 2014.

[12] KK Dobbin and RM Simon. Optimally splitting cases for training and testing high dimensional classifiers.

[13] Payam Refaeilzadeh, Lei Tang, and Huan Liu. *Cross Validation*. 2009.

[14] David Kriesel. *A Brief Introduction to Neural Networks*. 2007.

[15] Wlodzislaw Duch and Norbert Jankowski. Transfer functions: Hidden possibilities for better neural networks. In *9th European Symposium on Artificial Neural Networks (ESANN), Brugge 2001.*, pages 81–94, 2001.

[16] Y. LeCun, L. Bottou, G. Orr, and K. Muller. Efficient backprop. In G. Orr and Muller K., editors, *Neural Networks: Tricks of the trade*. Springer, 1998.

[17] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[18] Xudong Xie, Qionghai Dai, Kin-Man Lam, and Hongya Zhao. Efficient rotation- and scale-invariant texture classification method based on gabor wavelets. 2008.

[19] Y-Lan Boureau, Jean Ponce, and Yann Lecun. A theoretical analysis of feature pooling in visual recognition. In *27TH INTERNATIONAL CONFERENCE ON MACHINE LEARNING, HAIFA, ISRAEL*, 2010.

[20] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. *CoRR*, 2013.

[21] A. Ng. Lecture notes. CS 229: Machine learning. *Stanford University*, 2003.

[22] Schematic of a neural network training with early stopping. http://documentation.statsoft.com/STATISTICAHelp.aspx?path=SANN/Overview/SANNOverviewsNetworkGeneralization.

[23] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 2015.

[24] Margarita Osadchy, Yann Le Cun, and Matthew L. Miller. Synergistic face detection and pose estimation with energy-based models. *J. Mach. Learn. Res.*, May 2007.

[25] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.