THE APPLICATION OF CONVOLUTIONAL NEURAL NETWORKS FOR RECOGNIZING OBJECTS IN DIGITAL GAMES

Marko Franjić

Zagreb, June 2019.
UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING
MASTER THESIS COMMITTEE

Zagreb, 11 March 2019

MASTER THESIS ASSIGNMENT No. 2026

Student: Marko Franjić (0036476767)
Study: Computing
Profile: Software Engineering and Information Systems

Title: The Application of Convolutional Neural Networks for Recognizing Objects in Digital Games

Description:
Convolutional neural networks are nowadays widely used for image recognition and classification. Your task is to give an overview of the potential of applying artificial intelligence and neural networks in the domain of digital game development. Using as an example a previously developed game that enables players to draw certain objects, your task is to use the Unity development platform to develop and test a convolutional neural network that will recognize drawn objects. Furthermore, your task is to study how network training and testing is dependent on the number and types of layers, with the goal being to obtain the optimal layers for which the object recognition network is sufficiently functional while minimizing the learning period. All necessary literature and working conditions will be provided to you by the Department of telecommunications.

Issue date: 15 March 2019
Submission date: 28 June 2019

Mentor:
Associate Professor Lea Skorni-Kapov, PhD

Committee Secretary:
Associate Professor Boris Mišinović, PhD

Committee Chair:
Associate Professor Igor Mekterović, PhD
SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 11. ožujka 2019

DIPLOMSKI ZADATAK br. 2026

Pristupnik: Marko Franjić (0036476767)
Studij: Računarstvo
Profil: Programsko inženjerstvo i informacijski sustavi

Zadatak: Primjena konvolucijalnih neuronskih mreža u digitalnim igrama na problem prepoznavanja oblika

Opis zadaća:
U današnje doba postoji široko primjena konvolucijalnih neuronskih mreža za potrebe prepoznavanja klasificiranja slika. Vaš zadatak je dati pregled primjene područja umjetne inteligencije i neuronskih mreža u razvoju digitalnih igara. Na primjeru prethodno razvijene igre koja omogućuje igračima crtanje određenih likova, Vaš zadatak je razviti i testirati konvolucijalnu neuronsku mrežu koja prepoznaje nacrtane predmete koristeći programsko okruženje Unity. Nadalje, potrebno je provesti analizu učenja i testiranje mreže u ovisnosti o broju i tipu slojeva kako bi se pokušali dobiti optimalni slojevi za koje je mreža za prepoznavanje predmeta dovoljno funkcionalna nakon što kraćeg perioda učenja. Svu potrebnu literaturu i uvjete za rad osigurati će Vam Zavod za telekomunikacije.

Zadatak uručen pristupniku: 15. ožujka 2019.

Mentor:
Izv. prof. dr. sc. Lea Skorin-Kapov

Djelovoda:
Izv. prof. dr. sc. Boris Milišinović

Predsjednik odbora za diplomski rad profila:
Izv. prof. dr. sc. Igor Mekterović
# Table of Contents

1. Introduction ................................................................................................................................. 6
   1.1. Problem statement and motivation ...................................................................................... 6
   1.2. Method of solution ............................................................................................................... 8
   1.3. Thesis structure ................................................................................................................... 8

2. Artificial intelligence and neural networks .................................................................................. 9
   2.1. Artificial neural networks ..................................................................................................... 9
   2.2. Convolutional neural networks .......................................................................................... 12
   2.3. History of convolutional neural networks ......................................................................... 18
   2.4. Applications of convolutional neural networks ................................................................. 20

3. Artificial intelligence in the gaming industry ............................................................................. 22
   3.1. Pushing the boundaries of game development .................................................................... 22
   3.2. Important historic benchmarks ......................................................................................... 23
   3.3. Growing need of improved AI solutions .......................................................................... 24
   3.4. Examples of artificial neural networks usage in virtual games ........................................ 24

4. Development of a CNN for Recognizing Objects in Digital Games ........................................ 26
   4.1. Unity game engine .............................................................................................................. 26
   4.2. Way to implement a functional CNN in Unity ................................................................. 27
   4.3. Construction of convolutional neural network parts ......................................................... 29
   4.4. Training process of the constructed network .................................................................... 36
   4.5. Testing process of the trained network ............................................................................. 40

5. Dataset Collection ....................................................................................................................... 42
   5.1. Training datasets ................................................................................................................ 43
   5.2. Testing datasets ................................................................................................................. 45

6. Performance Analysis .................................................................................................................. 47
   6.1. Network testing parameters ............................................................................................... 47
   6.2. Multiple solutions of the network for data analysis ......................................................... 49
   6.3. Final network analysis ....................................................................................................... 51

7. Conclusions and Future work ..................................................................................................... 54

Bibliography .................................................................................................................................. 56

Abstract ......................................................................................................................................... 60

Sažetak .......................................................................................................................................... 61
1. Introduction

The digital games industry is developing at a rapid pace in the last couple of years, attracting more interested parties with its success than ever before. It made $116 billion in 2017, which was a 10.7% growth from 2016 [29], $135 billion in 2018, which is again a steady 10.9% growth from 2017 [30], and is expected to reach over $180 billion by 2021 [31]. Through these last couple of years, mobile gaming revenues grew the most, which is a result of player audience expanding to casual and hyper-casual games that are simple and require less time to be invested in them [32]. With that in mind, currently it is the right time to get involved in this industry without a need for tens or hundreds of people in a development team.

The main goal of this thesis will be the implementation of a convolutional neural network (CNN) used in an interactive comic book game called “Me and SuperBoy” that is being developed outside the scope of this thesis, as part of a student project involving 6 students from different faculties of the University of Zagreb, with the author of this thesis being one of them. The thesis will build on this developed game by applying a CNN for object recognition, as will be described later. The game itself will not be discussed in detail, but since the network will be one of its main functionalities, key characteristics of the game will be explained to understand why some of the choices were made, whether it be tools used or programming necessities.

1.1. Problem statement and motivation

The key characteristic of a CNN is its ability to recognize patterns [33], and that will also be the goal of the functionality being developed. One of the best examples of this usage is a game called Quick, Draw! developed by Google, which also acted as one of the main motivations for choosing a functionality that works in a similar way for our game. In that game the players can freely draw whatever object they want, and they will be offered with a result that the game thinks is right for that picture.
That functionality of the game “Me and SuperBoy” that is being developed in the scope of a student project differs with respect to similar games of the same genre. In this interactive storyline-driven comic game, players are faced with moments when he/she needs to draw an object on the screen, and that object needs to be recognized by the game out of a couple of sketches that are given to the player to draw from.

Furthermore, that object can be drawn anywhere on the designated part of the screen and be of any size, while still being recognizable. Preferably, it should also be recognizable under different angles, because the players could draw some of the objects, like mining pick or ladder, that way. Furthermore, objects drawn by players should be recognized in “real-time”, i.e., it should not take the game too long to process the input. Because of that, when designing the functionality, the project team has decided that the image processing and the decision process need to be done in less than 1 second.

Early on in the project, the decision was made to have different objects that can be drawn during different levels in the game. That means that the structure should be adaptable for a large amount of different forms and shapes. Those objects are also prone to change during the whole game development process, so that should also be taken into account. Since there are not a lot of examples to go by when designing, programming, and testing a functionality like the one described, or at least not the ones our student project team had access to or was able to find, it was decided that the solution being made needs to be at least good enough for a demo version of the game which includes the first story arc in which 6 simple objects (Figure 1.1) need to be recognizable; a closet, a ladder, a lasso, binoculars, a mining pick, and an animal.

Figure 1.1. Six objects to be drawn by players that need to be recognizable by the demo version of the CNN implemented within the scope of the game “Me and SuperBoy”
1.2. Method of solution

There are many ways in which the problem of object recognition can be solved. The very basic one would be to just have a base of objects that needed to be drawn and, while the player is drawing, to compare the positions of pixels to see what object it suits the most. While this is not a good solution for our problem because of the position and rotation of the image, it could be useful for a simplistic and quick solution and would probably work for similar types of functionality. Furthermore, most similar solutions like that one will suffer from both the time restriction and the adaptability for future implementations, which would require a lot of tedious work to make a mirror image with which the drawn object would be compared to. That is why a CNN was chosen for this task. It offers us an opportunity to easily transfer it through the levels, since it can be easily learned to recognize the objects for each level. Also when it comes to the time it would take the trained network to process an input object, it would be negligible, as was explained earlier, where in the testing phase, only the forward pass is being performed. The issue that remains is that for training the network, there is a need for a large dataset of object samples. There is also a need to investigate for how long and under what parameters the network should be tested before it will be functional. Because of that, an analysis is given at the end of the thesis to understand this network better for future work.

1.3. Thesis structure

In the first part of the thesis, the theoretical background will be given, explaining all the terms, structures and processes that will be used in the other chapters. Later in chapters 4 and 5, the whole process of development of a CNN will be given, together with the explanation of why certain tools were used, trials and errors of certain methods and the detailed overview of each part of network and some problems that had to be overcome. Moreover, the last part of the thesis will discuss possible improvements and future work. The analysis will test the successfulness of the network under different parameters and structural changes. Conclusions will be discussed, as well as guidelines for perfecting and continuing the future development of the game functionality.
2. Artificial intelligence and neural networks

   Human behaviour such as being able to comprehend languages, solving mathematical tasks or recognizing our visual surroundings can be marked as "intelligent" behaviour [24]. Through the development of complex computer systems, some of them have been built to be able to perform those or similar tasks. That being said, we can say that these kind of systems have some degree of "artificial intelligence".

   Over the years, researchers have been developing structures, algorithms and techniques that have been steadily tested and improved. Some of them were a result of complex mathematical tasks, some came as conclusions of empirical data gathered from different experimentations and others began as ideas that they got from looking at and researching the systems we can find around us, in the nature [24].

   The field of artificial intelligence (AI) is very broad, ranging from proving mathematical theorems, automatic programming, robotics, natural language processing and many more. Like with human intelligence, one of the key features of AI is for computer system to be able to learn from its inputs and computing of those inputs. Deep learning, which we will focus on in this thesis, is a part of machine learning algorithms that enables systems to extract and learn complex features from raw input data [25].

2.1. Artificial neural networks

   Artificial neural networks (ANNs) are computing structures that deep learning is based upon [18]. Those structures can be seen as sets of algorithms whose main ideas were found in the nature, in the human brain itself. The term neural originates exactly from there, the main purpose of those algorithms is the interpretation of the sensory data they have been given and from that being able to learn and recognize patterns. The way they interpret the data is most often through some kind of machine perception, labelling or clustering raw input.
Just like a biological brain is a vast network of structures called neurons which are intertwined with connections called synapses, artificial neural networks got their name because of a similar structure. An ANN is based on a collection of units or nodes called **artificial neurons** that have a job of receiving a signal, processing it and then signalling other neurons that are connected to them with connections called **edges**.

To know which artificial neurons are connected and what is the origin and what is the destination of the signal, ANNs are divided in layers. Below (Figure 2.1) we can see the general structure of a network, with one main input layer whose artificial neurons receive raw input signals, one or more hidden layers whose neurons are receiving already processed data, processing it further and sending it to the next layer of neurons, and the output layer which serves to show the result of network’s overall interpretation of the input data [1].

![Figure 2.1 Representation of multi-layer fully-connected ANN](image)

Edges between two connected artificial neurons have a certain value \((w)\) that they possess, which is important for the calculation of output in every node (Figure 2.2). Each node of hidden or output layers calculates the weighted sum of the inputs from the previous layer and passes it through a function to get the result which it sends to the node in the next layer. The function \(f\) is called a **transfer or activation function** and it basically decides in which form the data should be passed through the node. It needs to have a degree of more than one (non-linear) for the purposes of training the network to be able to learn more complex non-linear mappings from inputs to outputs [2]. Most popular types of activation functions include sigmoid, logistic, hyperbolic
tangent or rectified linear units (ReLU) [19]. The last one of those is currently most popular in the world, and its calculation is quite simple, it just changes all the negative values to 0, which basically disables their influence for further network calculations [19].

\[ x_1 \]
\[ x_2 \]
\[ x_3 \]
\[ \vdots \]
\[ x_n \]

\[ \sum f \]  
\[ y_1 \]

**Figure 2.2 Node of hidden or output layers in an ANN [1]**

The other algorithms that are included mainly for training the neural networks will be discussed in detail in the next chapter, which will go over a specific type of ANN that has some additional components. Generally, ANNs are viable computational models for a wide variety of problems, such as pattern classification, speech synthesis and recognition, adaptive interfaces between humans and complex physical systems, function approximation, image data compression, associative memory, clustering, forecasting and prediction, combinatorial optimization, nonlinear system modelling, and control [18].
2.2. Convolutional neural networks

Since the main focus of this thesis will be on analysing visual input, it is important to explain the structure of a specific type of ANN that is most commonly used for exactly that purpose. A Convolutional Neural Network (CNN) has a couple of extra main parts (Figure 2.3) that helps it process visual input [2]. Those are input layer, one or more convolutional layers and one or more pooling layers. Each of them has a different, very important role in the overall data flow through the network.

![Figure 2.3 Representation of a convolution neural network](image)

The first layer is the input layer, which is a place where the raw sensory data is given to the network to process. That data is shown in the form of an image, more precisely the pixels that form the image. Those pixels are given to the input layer in an array of their numerical values, whose size depends on the resolution and size of the image itself (Figure 2.4).

![Figure 2.4 Visual representation of raw data of input layer](image)
This is where the differences between the input and convolution layer end, because both of those types have the same main functionality although they process different kinds of their respective input data. A crucial part of every convolution layer are filters, which are an array of numbers often called *weights or parameters*. The process of convolution (Figure 2.5) is the task of shifting the filter over the input array and calculating the output which is given by adding together all the multiplications of singular numbers in the filter with their pair in the area that is being convoluted upon (called *receptive field*) [4]. Crucial to note is that the input data can have depth also (3D array) and that means that the convolution filter needs to have a depth of the same amount.

![Convolution of receptive field over input image](image)

*Figure 2.5 Convolution of receptive field over input image [4]*

After sliding the filter over all the locations, the output array has a size that depends on the size of input and filter arrays, but always has depth value of 1. This data is called *activation or feature map*. In one convolution layer there can be multiple filters with different weights that convolve over the same input data, giving multiple different activation maps. That data then acts as input data for the next layer in the network.

When it comes to values of filter weights, those are one of the most important parts of the whole network. During the process of network training, which is explained in-depth later in this chapter, they are subject to change depending on the training sets. That means that when filters are first being initialized, their value can be either randomized, set to the same value, or be set using some specific function which depends on the place in the network that the filter is in (how far from the input layer that layer is).
After every convolution layer there is generally a layer where pooling happens. Before going into the details, it should be mentioned that there are a few main types of pooling: mean pooling, sum pooling, and max pooling. The focus here will be on max pooling.

Since these layers come after convolution layers, their inputs are feature maps, which means that where the numerical values are higher, certain features are detected on that part of the input image. Max pooling (Figure 2.6) is the process of artificially losing the unnecessary data in a way that an area of a fixed size is moving from the top-left corner along the rows and for every group of singular numerical inputs it covers, it calculates the maximum value and that is then a singular result in a structure called a pooled feature map [5]. Through the pooling layer, networks acquire a property called spatial variance, which enables the network to detect the object no matter the differences in size, rotation, image’s texture or similar properties.

![Feature Map](image)

*Figure 2.6 Input and the result data of Max pooling layer [5]*

In any single CNN there can be any number of convolutional and pooling layers, but at the end of the network structure, the last feature map or pooled feature map is always an input of a fully-connected layer. That is that part of the regular neural network in which every neuron from one layer is connected to every neuron of the second one. The only difference is in that how the last feature map is translated into a first part of a fully-connected layer. The answer to that is flattening (Figure 2.7), and as a result there is just a long vector of input data [6].
Inside the fully-connected layer there can be multiple internal, *hidden*, layers. Their main purpose is to combine the extracted features from previous layers into a wider variety of attributes that make the network capable of classifying. The final part of this layer, called *output* layer, has the same amount of neurons as the number of objects that need to be classified among.

After the image enters the network, and goes through all the calculations in different layers, the given result is shown in the output layer. The biggest numerical value of a certain neuron shows that the network believes the object that represents that neuron was the one in the input image.

This whole process is called *forward pass* (Figure 2.8), in which the input data is forwarded through the network, and is the only part of the CNN training, but only one of the four main parts of network training. During testing, the filter weights and weights of fully-connected layer connections have already been “trained”, which means that they are not being changed anymore with new data inputs (testing datasets).

For those values to become that way, the network needs to go through a process of training. During training, there is an extra vector that comes into calculation with the values in the output layer. In that vector, which is of the same size as the output vector, all the values are 0, except for a single one, which is set to 1. That marks the “real” output, the one that should be calculated if the network would be able to classify the input image. That vector is then subtracted from the output vector and that data goes through a function called *loss function*, which is the second most important part of the
training process. There are a few different functions that can be used to calculate the loss, but the one that will be used in the scope of this thesis will be explained in detail later.

After obtaining the loss function, that result is propagated backwards through the whole network structure in a process called backpropagation or backward pass. The purpose of that is to adjust numerical values of every weight in the network in an optimal set. That is done by finding out which weights contributed most directly to the loss of the network and adjusting them so that the loss decreases.

To do that we need to derivate the loss function and pass that value backwards through the network, recalculate it regularly on the way (Figure 2.9). The new adjusted values of weights are being calculated at the same time [8].
The learning rate can be chosen at will, depending on how large the steps in weights update want to be. Higher learning rate means that it might take the network less time to converge to optimal values for weights, but if it is too high then the overstepping of optimal values might happen and the weights might not be able to converge.

When it comes to weights update, as the last part of the network training process, there are different approaches to it. They can be updated after every input image and its backpropagation, or after a batch (more different ones) of images while they are all calculated with the same weight values.

The training process can be very different depending on what kind of network structure it as and what kind of data is being processed. Generally, there are some main terms that will be used in this thesis when it comes to training.

A sample is a single input unit, so in this case it will be an input image. Training and testing datasets are comprised of many samples that are all different from one another. If the weight update is done after more than one input (sample), then we refer to a batch. Datasets can be divided into one or more batches. Last but not least, working through an entire dataset is called epoch, so if we are using the same training data multiple times, we have more epochs.
2.3. History of convolutional neural networks

To get a better understanding of convolutional neural networks, it is important to find out what researchers and scientists in the past went through and what challenges they needed to solve to come up with discoveries that led its development. It all started in 1959 when two neurophysiologists, David Hubel and Torsten Wiesel, have shown the main response properties of visual cortical neurons which was a result of their experimentation with a cat (Figure 2.10) [9]. The experiment included observing the neural activity of the cat’s primary visual cortex area. After some initial failure, the scientists discovered that there are simple and complex neurons and concluded that visual processing starts with simple structures like sharp, oriented edges.

According to [9], the same year, another important breakthrough in computer vision happened, scanning of a first digital image. Computer pioneer Russell Kirsch made a machine that could transform an image to a numbers grid, out of which the first was a one of his infant son measuring 176x176 pixels. Next, in 1963 Lawrence Roberts made a program that was able to extract info about 3D objects from their 2D photographs. The main idea behind the program was to process the photographs into line drawings and then make a 3D representations from them.
In 1982 a British neuroscientist David Marr expanded on Hubel and Wiesel’s idea in a way that he concluded that vision is hierarchical. He was the first one to introduce a framework for vision where first the simple shapes like edges and curves are detected, which are then worked upon towards detecting more details in an image. Also at the start of the 1980s, a Japanese computer scientist Kunihiko Fukushima developed an artificial network called Neocognitron. It could recognize image patterns while being unaffected by the position of object in the image. He was the first one who included convolutional layers with receptive fields that had weight vectors, similar to filters nowadays. The network was primarily used for handwritten character recognition.

Finally, according to [9], the main foundation after all these breakthroughs for modern CNNs were set in 1989 when a French scientist Yann LeCun had built a version of backpropagation learning algorithm on top of Fukushima’s network architecture. That structure, called LeNet-5 (Figure 2.11) was also focused on character recognition and after working on that project for a few years, LeCun gathered a dataset of handwritten digits called MNIST, which is still one of the most famous benchmark datasets for machine learning [10]. In the late 1990s the focus of computer vision as a field was shifted towards creating 3D models of objects, which is outside the scope of this thesis.

![Figure 2.11 Architecture of LeNet-5 convolutional neural network [10]](image-url)
2.4. Applications of convolutional neural networks

CNNs are being used for a wide variety of applications, in all of which the input data can be convolved upon. Through that the different patterns can be recognized and artificially learned. Here a couple of main fields of application will be explained to show the effect of CNNs.

The biggest application is in the area of computer vision, where CNNs are commonly employed to identify the hierarchy or conceptual structure of an image. According to [20]: “Instead of feeding each image into the neural network as one grid of numbers, the image is broken down into overlapping image tiles that are each fed into a small neural network”. In the field of computer vision, a CNN is often used for applications like image classification, facial recognition, action recognition, human pose estimation and document analysis and handwriting.

For image classification (Figure 2.12), the network is given a huge training dataset of images of targeted objects, ranging mostly between 1000 and 10000 different images. Depending on how detailed and similar the objects are, training can be more or less successful [4]. Learning facial recognition, on the other hand, can be prone to different issues, such as identifying all the faces in one image, unique facial features, bad lighting, different pose, or position of the face in general.

![Figure 2.12 Application of a convolution neural network in image classification [4]](image-url)
Another important application is in the field of natural language processing. Since CNNs have from their inception been used for extracting information from raw signals, speech should not be that much harder to recognize or classify. Lately these networks have also been applied to the different tasks of sentence classification, topic categorization, sentiment analysis and many more [20].

When it comes to video analysis, it is much more complex than image classification, but nevertheless, there has been some development in trying to perform convolutions in both time and space [21]. It can be said that it is still very much work in progress.

CNNs have been also used in drug discovery, by predicting how molecules will interact with biological proteins to possibly identify potential treatments. Similar to how network can compose low-level shapes and forms into complex features at image recognition, here it can discover chemical features and also possibly predict biomolecules that can solve different diseases.

This last application is also a connection to the next chapter and the rest of the thesis, but it deserves to also be mentioned here. Game of Go (Figure 2.13) is one of the most complex board games in existence and in 2017 a couple of CNNs were used by a computer program called AlphaGo to beat (for the first time) the best human player in the world [22].
3. Artificial intelligence in the gaming industry

Artificial intelligence and the gaming industry have always been going hand in hand, since the development of one kept pushing the development of the other. Most often in digital games developers are using AI to try to simulate human-like behavior. That adaptive and responsive behavior is usually assigned to a game structure called non-player characters. Moreover, some fields of AI are used for things that the player does not come in direct contact with, such as procedural generation of game content and data mining.

3.1. Pushing the boundaries of game development

The main purpose of digital games is for them to be engaging and overall an enjoyable experience for a player. For them to be that, they need to have that special something that differentiates them from other entertainment medias, they need to be interactive and immersive as much as they can.

Being able to give players an obstacle in the game that requires them to be inventive and skillful at the same time often makes a difference between a good functionality and a great one. The field of AI makes it possible to implement different, adjustable, self-learning obstacles or behavioral patterns that do not follow traditional human-like logic.

Some of the most thrilling functionalities and game design ideas have come as a result of a certain field of research of AI. Since game developers are always being pushed to be innovative with their creations, the gaming industry of the future definitely will not suffer from the lack of new possibilities to try out.
3.2. Important historic benchmarks

Since the early start of computers and digital games, AI had played its part. In 1952 the game *Nim* was released, which included the first ever implementation of a computerized opponent [27]. Nim is a strategy game that consists of players removing objects from game heaps or piles. The computerized opponent could regularly win games even against the best human players.

At the start of the 1970s, games that are adjusted for a single player to play them started to be developed. That means that everything else had to be pre-programmed or use some sort of AI, mostly based on stored patterns. During that decade, microprocessors have been developed, allowing game designers to be more innovative with adding random elements into enemies’ movement patterns.

In 1978 *Space Invaders* (Figure 3.1) changed the perception of AI opponents in that time by quite a lot. It had different movement patterns, difficulty levels and in-game events [12]. In 1980 *Pac-Man* brought AI patterns to maze games and *Karate Champ* in 1984 to fighting games. In the 1990s the games began to use a tool that is among the most popular ones even today, *finite state machines*.

![Figure 3.1 Space Invaders](image)
3.3. Growing need of improved AI solutions

The gaming industry is currently following the steps of other parts of the entertainment industry in a way that is being rapidly developed and the money that is being invested in it is increasing every year. The market for digital games is greater than ever, but with that also the number of companies making games. That being said, it is very hard to be completely original when making a new product, so game designers and developers are either trying to improve and perfect the current popular game ideas and mechanics, or they decide to make use of new developing technologies for certain elements of their games.

Solutions from the field of AI can be used for both of those purposes, even though it is sometimes much harder to implement those functionalities, especially when it comes to debugging, testing, and quality assurance. It is after all, an industry like every other, so it is normal that some people working in the gaming industry often care more about the release windows, players’ feedback and profit, than the research and scientific part of trying out new, unproven technology and all other "issues" that come with it. So there will always be this kind of balance between new AI discoveries and possible ways of using it in new ideas and the pure industrial aspect, but with the whole market growing, both of those parts are developing at the same time.

3.4. Examples of artificial neural networks usage in virtual games

ANNs are used as functionality in games much less often than some of the other types of AI techniques, such as finite state machines or behavior trees. One, more simple reason for that is that, more often than not, the other simpler solutions are good enough for what they are necessary for. It is not that common to have functionalities in games that need to recognize patterns, but in those that do need that, they are more than likely to dive into usage of some kind of ANN.
The other reason is the technical aspect, because games, like any other project where the result needs to be a stable product at a given deadline, are not keen on using structures that can be changed or need to learn from end-user input to adapt behavior. That is certainly not a stable environment and would often lead to a nightmare for quality assurance teams. However, there are some examples of bigger games that did manage to implement ANNs to some of their functionalities.

In a real-time strategy game *Supreme Commander 2 (2010)* (Figure 3.2), a neural network was used to control the fight or flight response of non-player characters [26]. The other functionalities were controlled by more simplistic AI, but for the team that made the game it was that implementation that they were most proud of. The main benefit that they managed to get from it is that they did not have to come up with the weights by themselves, saving them a lot of time, but that did come at the cost of spending time on the training process of the network. The other important aspect that came out of the network was that of an abstract representation of a group of non-player characters, that did not change its composition even though the statistics on the individual units changed, as long as the group was functional, the networks were able to continue to make good game decisions.

Other interesting example is the one of *Forza Motorsport 6 (2015)* that collects driving data from the players. The data is collected in a cloud storage and then used as input in deep learning algorithms to make “Drivatars” that imitate how an individual plays (Figure 3.2).
4. Development of a CNN for Recognizing Objects in Digital Games

In this chapter a methodology for developing the functionality of drawing and recognizing objects is described, implemented in the Unity game engine. The chapter will go through the aspects that are needed to perform these tasks, from planning and constructing the scene and structure of the CNN, to training and testing both the network and overall game functionality.

4.1. Unity game engine

Unity software is a game development platform used for making virtual games for a wide variety of different platforms such as Android, iOS, Windows, Mac or Linux [23]. This kind of software is usually called a game engine because of the fact that it already offers game developers implementation of physics, collision detection, graphics rendering and more. Unity gives a developer full freedom of control over every aspect of the project folder structure. That being said, it is important to keep in mind that it is possible to generate and use a wide variety of different files when producing a virtual game, so having folders such as Scripts, Scenes and Prefabs in the main Assets folder is advisable. Scenes are a crucial part of every Unity project. They contain everything that a player sees, such as menus and user interfaces, and everything else that is happening in the game world such as solid objects, lightning, camera or sounds. Every object that is in the scene can have different properties, scripts, and a lot of other possibilities. These are called Components and can also be turned on and off at will.

The main scripting language in Unity is C#, so to be able to develop the script it is preferable to have Visual Studio with Unity add-on package installed. Since scripts are also components, they always need to be put on a game object. Every Unity script derives from MonoBehaviour class that lets it implement certain main functions that are called from the game engine. Some of those are Start() that activates after the game object has been initialized and Update() that runs once every frame while the game object is enabled.
4.2. Way to implement a functional CNN in Unity

Since the CNN will need to go through the process of training before it can be used in a game, there needs to be developed a special scene just for that purpose. Also, after the training, the newly calculated weights values need to somehow be transported together with the whole network structure to the main scene where it can be used in a game.

To save the numerical values of weights, it was decided that the best possible way would be to keep them all in textual files (Figure 4.1). That way, following network initializations, the files would be created, and during training the values for weights of each layer would be parsed from necessary files, calculated upon, had their value updated and then saved back in the files. When the network would be done with the training process and ready to export to the main game scene, the files would not need to be touched, except if the game is made in a different Unity project, then they should be copied to a certain place in that folder structure and have their path to read / write from them adjusted.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLayerWeights.txt</td>
<td>Text Document</td>
<td>34 KB</td>
</tr>
<tr>
<td>OutputLayerWeights.txt</td>
<td>Text Document</td>
<td>3 KB</td>
</tr>
<tr>
<td>ConvLayer2Filters.txt</td>
<td>Text Document</td>
<td>38 KB</td>
</tr>
<tr>
<td>InputLayerFilters.txt</td>
<td>Text Document</td>
<td>43 KB</td>
</tr>
<tr>
<td>ConvLayer1Filters.txt</td>
<td>Text Document</td>
<td>4 KB</td>
</tr>
</tbody>
</table>

*Figure 4.1 Files where the values of weights are being initialized, updated and saved to*

There are 5 files that are used in both training and testing phases and they each represent a certain set of weight values for a specific layer in the network. The other 6 files are network training datasets, which will be explained in detail in the following chapter.
There are a couple of more components that need to be explained which will be used for constructing the network in this thesis. They are generally used for player’s interaction with the game scene.

Camera component has the main task of rendering everything that the player sees in the scene. It can be moved and rotated through the scene so knowing exactly how it works and behaves is important when developing a functionality that deals with player input. It should be noted here that accidently different camera setup was used in training (2D perspective) project and game project (orthographic – 3D setup) and that led to some issues that needed to be fixed when implementing the trained network into the game project.

Line Renderer component is the main component that is used to obtain player input, to draw objects on screen. It can be added manually (Figure 4.2) to the game object, or through scripts, which was used for this thesis and will be described later. The main idea behind is that it has an array of points in 3D space and it draws a line between adjacent array members. Its width, color, and material can be changed, and also the shape and size of the edges and corners.

![Line Renderer component]

*Figure 4.2 Example of manually adding Line Renderer component on an empty game object in a scene*
Last thing to mention is the user interface. Unity uses a component called *Canvas* to automatically gather all UI elements that can be found on the scene. Canvas can render itself in different modes, which mainly depends on whether we want its position to be static on the player’s screen, no matter the camera position, or static in the scene’s world space. From UI components we will use *Button* and *Text* for testing.

### 4.3. Construction of convolutional neural network parts

The general idea is to implement an input module with all the scene components that are needed to communicate the raw input data to the neural network, which will be constructed with C# scripts that will be added to an empty game object. Every layer of the CNN will be programmed separately as a standalone module so that they can be easily adjustable. This allows to change the network structure more easily.

The first issue that had to be dealt with is how to exactly draw an object and present that into data that the network can work with. The most commonly used for tasks of drawing, and the only tested, option was to use the Line Renderer component that updates every frame while the mouse button is clicked (Figure 4.3), by adding new point positions at the end of the array. That way the player gets the feeling of drawing and the positions of object edges are being remembered for future use.

To restrict drawing to a certain area of the screen, the panel components have been put in the background to indicate to the player where he/she can draw. The positions of line renderer are then updated only if the mouse button is clicked and the mouse position is in the restricted area.

*Figure 4.3 Area where Line Renderer is being updated while drawing*
Canvas was added so that the buttons with different functionalities (Figure 4.4) can be implemented. Their tasks are around drawn line renderer, like rotation by a certain angle, scale to fill out the whole restricted area, clear drawing area, show background and a few more that will be explained in training and testing chapters 4.4. and 4.5. Since the camera is static in the training scene, and these buttons (except for the clear area one) are used only for that scene, visually they are very simplistic, but they do their tasks.

![Figure 4.4 Canvas buttons that can change made line renderer and other communication with neural network](image)

To get the necessary raw input that has been described in the introduction, the line renderer array positions need to be transformed into something else. For this we are also keeping in mind that there might be a need for more detailed image outputs so we will call that network input parameter \textit{resolution}. The 2D input array for the first convolutional layer has a size of \textit{resolution} \times \textit{resolution}, and the ones used and tested the most are the values of 32, 64, 128 and 256. Even though the higher resolutions were tested, the final version of the structure had either resolution of 32 or 64, because they were good enough for the object that had to be drawn.

To transform the line renderer positions to raw network input, an additional boolean 2D array (later in the thesis referred to as \textit{visited}) was used. Depending on the resolution, the coordinates of restricted drawing area were mapped to a certain member of the array, and during line renderer updating it checks if the new position is in a part of the grid where there were no points before. If it is, then it marks the necessary value as
true, which means it has been visited / entered and it will act as a pixel input to neural network.

This is very important because of 2 reasons. One is very obvious and that is because this keeps track of the drawing line, which is controlled by mouse movement whose position is being monitored every frame in Update() method. To optimize the line being rendered, there is no need to keep adding mouse positions that are already there, for cases that the user did not move the mouse for longer than few frames. That way, if the mouse position coordinates match the value of true in 2D array that keeps track of visited input pixels, it does not need to be added to Line Renderer coordinates component.

The other reason is on the opposite side of the spectrum. If the user moves the mouse fast enough between the frames the distance the mouse covers is large enough that 2 cells in visited 2D array are not adjacent. So those two values would be true, but all the ones between them, that also need to be marked as true, would be false. To counter that, there needs to be a way to find out which values need to be set to true if that happens. That is why on every frame update, if there is a need to update the visited array, the last two Line Renderer coordinates are checked and if the distance is large enough a new method is called which sets the necessary values in visited between those to true.

The panels on the grid that visualizes visited array are set to a different color than the surrounding background and are activated with clicking on a “B” button (Figure 4.5). This feature helps significantly with debugging the network because it offers an exact with of the input data that enters the network.
Another feature that was designed for this project is the ability to scale lines and objects that have been drawn. Like previously explained, this is not necessary to have for a functional CNN, as it should be able to recognize objects no matter the size. Nevertheless, for this project the ability to scale all objects to the same size was implemented because the game is still in an early development stage. Hence, there is a need to be able to work with lower resolution of input data and to shrink the possible amount of different versions of the same objects. This helps with network testing time consumption, because the testing dataset can be smaller.

Scaling was achieved in a way that the object increases by same multiplier on both axis to fill out the whole drawing area. It is done by re-calculating the coordinates of the Line Renderer positions array. The method that does that goes through all the current coordinates, locates the topmost, bottommost, leftmost and rightmost ones, decides whether the correct multiplier is for x or y axis so that the scaled object will not go out of bound, calculates the offset for which leftmost and bottommost positions need to be moved to be positioned on the edges of the area and finally adjusts every other coordinate to fit those values of multiplier and offsets.

Same as for the background, scaling can be activated by click on the button “S” after the object has been drawn (Figure 4.6). Moreover, this button only exists for debugging purposes, because during both training and testing processes object scaling method is being called in a different way, which will be explained in the following chapters.
Last, but not least, rotating was a finally feature added to the ways of controlling user input data. CNN should be able to recognize objects that are positioned under an angle also, which means that the drawn objects in training datasets need to be that way. For that, method for rotation is being used, which goes through all the coordinates of Line Renderer and calculates their new positions. It does that by rotating them around the center of drawing area for a certain angle that is defined by the method input parameter.

Again, for the purposes of debugging the network, UI button “R” was added which simulates a rotation of 5 degrees (Figure 4.7). Since this feature was developed as last one before starting to generate the testing datasets, to test if all other necessary features are working as intended, clicking on this button also activates other two previously described methods, scaling and background visualization.

Figure 4.6 Object scaling being activated by clicking UI button

Figure 4.7 Example of drawn object before and after clicking the rotation UI button
With that, the main methods for managing the drawn object have been explained. The other part are the structures of CNN itself. That being said, the classes for convolution layer, filter of convolution layer, max pooling layer and fully-connected layer have been constructed. They were all built from ground up while keeping two things in mind, to structure them as separate entities with inputs and outputs so they can be defined and used in different kinds of CNNs, and with all necessary properties and methods for forward pass and backpropagation. It can be seen how they are connected between themselves on Figure 4.8 in the following chapter 4.4.

Convolution layers are represented by a class called `ConvolutionLayer`, and its main properties that need to be set in the constructor are value of input size, number of inputs, number of different filters, value of filter size, and name of the file with filter values for the layer and its location in the folder structure. As for the methods, they are used for setting the input data, calculating and fetching the output data (different ones for forwards pass and backpropagation), calculating the values for weights update, and updating the files in which the weight values are written.

Convolution Layer filters are represented by a class called `Filter`, and its main properties that are set during the network initialization are string value of the file name with its current weights and its location in the folder structure, value of its size (resolution), value of its depth, value of its layer input size (for forward pass), and value of its layer output size (for backpropagation). In the filter constructor, the fetching of its weights from files is also being done. Filter class has only one extra method, which transforms weight values from an array to a necessary string form that can be written in the file.

Max pooling layers are represented by a class called `MaxpoolLayer`, and since it does not have filters with weights it is much more simplistic than `ConvolutionLayer` one. Its main properties are value of input resolution, value of input depth, and value of the size of the area that decides from which values max value has to be found. It has methods for generating results for forwards pass and backpropagation, and its main one that calculates the maximum value from a 2D array of numbers.
Since the outputs from convolutional and max pooling layers are 2D array values, and the input of fully-connected layer should be one dimensional vector, a new structure needs to be implemented. Its job is to flatten the output of the last max pooling layer (transform it from 2-dimensional to 1-dimensional) and it is represented with a class called `InputLayer`.

Fully-connected layers are represented by a class called `FullyConnectedLayer`, and its main properties are value of input size, value of output size, and name of the file with layer weight values. It has the similar methods for calculation of forward pass and backpropagation data as `ConvolutionLayer`, but since these layers do not have filters, it also has methods for fetching from the file, calculating, updating, and saving weight values to the file. The output layer, which has the size of the number of objects that can be drawn, is also represented by `FullyConnectedLayer`, and on its output the loss function is being calculated.

The same structures are used for both the training and the testing processes, only the activation of certain layers’ methods is different. Because of backpropagation and how it is being calculated, all layers need to keep track of both the input parameters and the input data. Filters in convolutional and input layers are kept in `LinkedList` structure because it makes it faster to iterate through them that way. There was an idea to use `List` or `LinkedList` for weight values, but it was quickly scraped away because of the performance issues. All weight values through the network are of type `double` and are kept in 1D, 2D or 3D arrays.
4.4. Training process of the constructed network

After the initial construction and setup of the network parts, the training process can begin. As explained in previous chapter 2.2, it consists of forward pass of the input data, calculation of loss function, propagation of loss backwards through the network, and update of the weight values.

Before the training can start, there is a need to decide on the network structure itself. Previously described layer components were designed in a way that they can easily be connected to one another, while adjusting necessary parameters like resolution size of the inputs of each layer and filters’ numbers and sizes. This offers the possibility to design different versions of the network, while not having to worry about the functionality of every layer. That being said, for the game functionality of the network that is being developed, and which is the focus of this thesis, only one structural version has been created. That does not necessarily mean that it is the optimal or the best one, but it serves as a foundation that can later be built upon. As for why exactly this version was chosen, it involves 2 parts of each of the main layers (convolutional, max pooling, fully-connected) so that the importance of each type can be shown and tested.

The first part of the network consists of the first convolutional layer, the first max pooling layer, the second convolutional layer, and the second max pooling layer (Figure 4.8). The input data at the start is a 2D array of values of the drawn object, which was explained in the previous chapter 4.3 (array called visited – but values of type double, instead of boolean), but all the outputs and inputs to other layers are feature maps. The first convolutional layer has 6 different filters, all of which are applied on the same input data, because there is only one input array. Because of those 6 filters, there are 6 feature maps as output of that first convolutional layer, and which serve as input to the first max pooling layer. Its output then consists of 6 pooled feature maps, which are the input data to the second convolutional layer. That layer has 16 filters, but all of which are used on different input arrays and have different depth also. Consequently, there are 16 feature maps as input to the last max pooling layer, which then creates 16 pooled feature maps.
Figure 4.8 Construction of the first part of the CNN
Before serving as input to the last part of the network, those feature maps need to be flattened to a 1D vector array. There are 3 fully-connected layers, with the last one serving as output layer of the network (6 nodes for 6 drawable objects). Furthermore, to get the values that are easier to work with, softmax function (Figure 4.9) is used. It normalizes the outputs in a way that it converts them to values between 0 and 1, with their overall sum of 1.

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

![Figure 4.9 Mathematical definition and example of the softmax function](image)

Now comes the part that is very important for the training process. Learning of this network is supervised, which means that, during training, we need to “tell” the network which object is being presented to it. To do that, a 1D array called goal is used, whose values represent each of the possible objects, and are being changed depending on the input image. If the object is presented to the network, the value that marks it in the array is set to 1, while all others are set to 0.

The Goal array is used to calculate loss in the loss function part of the network. Since that is the value the network should be converging to, the “real” value, the one from the output layer, has to be subtracted by the values in the goal array. That way the resulting values that are closer to 0, and all the weights in network that participated in its calculation, should be changed less, and the ones further from 0 and its weight should be changed more.
Calculating loss function derivations for each layer will not be explained in length as a part of this thesis, but some further explanations will be given in chapter 6.1. about different network parameters. That being said, one of the main problems that was encountered during the practical work of this thesis was how to understand correctly which values are being calculated with what, and to design the structures that are capable of accomplishing those tasks. That was solved with further extensive research and debugging.

The last part of the training process is weights update. Since the values of all the weights of the network are kept in textual files, accessing them and saving new values takes a certain amount of time (around half a second). Which means that if the weights would be updated after every iteration of a single data input (one drawn object), the training process which includes 1000-5000 inputs would take some time. That is why it was decided not to update weights after every input, but after a batch of certain size. This will be explained in more details later.
4.5. Testing process of the trained network

CNN testing is much simpler than the training, since it only includes input data forward pass. When the network calculations reach the final part (here it is the last fully-connected layer), the value that is the largest in the output 1D array marks the object that the network “thinks” is shown on the drawn image.

There are multiple ways of checking whether the network has been trained well. The easiest way in Unity is through a console window, by writing out the necessary message (Figure 4.10). In this example, there are 5 possible objects that can be recognized, the values in the array percentages are calculated directly from the output of final layer (before softmax function), and they show exactly that, how sure is the network that the drawn object is that specific object.

```csharp
Debug.Log("Binoculars: " + percentages[0].ToString("#.00") + ", ", " +
         "Ladder: " + percentages[1].ToString("#.00") + ", ", " +
         "Lasso: " + percentages[2].ToString("#.00") + ", ", " +
         "Closet: " + percentages[3].ToString("#.00") + ", ", " +
         "Pick: " + percentages[4].ToString("#.00") + ", ");
```

![Figure 4.10 Example of testing the trained CNN through console window](image)

The other testing method that will be discussed in this chapter is basically a test of whether the recognition functionality works how it is intended in the game that it is being produced for. As it was noted in chapter 4.2, the Unity scene for the purposes of training the network has been developed separately from the main game. To be able to test it in a game scenario, the necessary structures from the testing scene need to be implemented in the game scene, together with scripts that are used to communicate with the network, and the network structures themselves.
Moreover, another important thing to note is, since those two scenes have been developed in different Unity projects, that the files with weight values need to be transferred also. Furthermore, the files’ path values in scripts had to be adjusted, since they are not the same as in the testing project.

Since the network should be fully functional and free of bugs, in game scene there is no need for UI buttons from the training scene. Those have been replaced with only two (Figure 4.11), but which have different tasks, one for activating and deactivating the drawing area, and the other for clearing what has been drawn. The network output values here are not displayed in the console window, but are sent to another component which only accepts one value, that being the index of an object with the highest calculated value in the output layer. It then displays the correct image on screen for the player.

![Figure 4.11 Testing the trained CNN as game functionality:](image_url)

a) shows activated drawing area, b) example of the image drawn by user, c) picture of the recognized object appears

One thing to note here is that there was one serious issue that was encountered. In the training scene the camera was not moving and drawing area, background panels, and every other component that is being initialized on scene never had to be repositioned to adjust to the camera movements. In the game scenario, that was instantly a problem because everything had to follow camera movement, on all 3 axis. It took a lot of time to fix only this, which basically had nothing to do with the network itself, but had to be done to be able to test the network. It was a design flaw that our project team did not anticipate, but the lessons were learned for further work.
5. Dataset Collection

The training and testing datasets are a very important part of any kind of AI learning process. They are sets of inputs (Figure 5.1) which are used to calculate network outputs. In case of the CNN that is being developed as part of this thesis, datasets consist of images that represent 6 items which the network has to be able to recognize.

The training and testing sets do not necessarily differ that much from one another. The training datasets are inputs that are used to train the network, and testing datasets are used to test whether the network is trained well enough. However, it is important to note that if the inputs were used as part of the training dataset, they should not be used in testing one. This is because it could provide a false validation that the network is working correctly when in fact it is not, but it had just “seen” those samples already and adjusted the weights accordingly.

Figure 5.1 Example of a MNIST dataset for number recognition [16]

Previously in chapter 2.2., the difference between sample, batch and epoch was explained. In this chapter those terms will be used for training and testing the CNN which is a part of this thesis.
5.1. Training datasets

At the early stages of development of this CNN the training was done only by using single samples at the time. That was performed with the goal of finding out how the network reacts with the current set of parameters. Moreover, while there were still issues with backpropagation and initialization of weights, it was helpful to see how that single input, and its network output from final layer, affect the loss function and weights update.

After those issues were dealt with, and the network was seemingly working as intended, a new feature was developed to allow easier CNN training. The problem being faced with was that for the purposes of training there needs to be hundreds or thousands of unique samples. Given the fact that, with how the input to the network has been designed, the only way to supply the network with samples is through drawing area, this looks like a very tedious task to train the network. Moreover, if half way through the training we find out that there is an error, all those inputs and work that was put in training would go to waste.

That is why the new feature was introduced that enables the training samples to be gathered beforehand, saved in a file, and used all together to train the network. By clicking the UI button “A”, after the object has been drawn, the scaled input is saved to the file (Figure 5.2).

There are as many files as the number of recognizable objects, 6 in this case. There are specific variables, only one of which is always set to true, to know which of those 6 files needs to be updated. This dealt with an issue of losing samples, but the number of samples is still a problem.
The solution to that comes from a feature that has been explained in chapter 4.3, and that is rotation. Even though the object being rotated is the same, for network, after each of those rotations, it is a different object because the network input 2D array has different values. This solves the problem of needing lots of samples because with just one drawing, and a rotation angle of 1 or 2 degrees, it is possible to get hundreds of samples.

Although, unless we want to make a database containing a couple of tens of thousands of samples, it is not wise to take a rotation that small because an object rotated by 1 or 2 degrees is still almost the same object to the network, especially if lower network input resolution is used, as in this case. For the purposes of obtaining a training dataset for this CNN, many different options were tried (different angles, number of rotations), and at the current state it was decided that 5 rotations per one drawn object with an angle between 5 and 10 are suitable to get the training dataset (Figure 5.3).

*Figure 5.3 Example of creating multiple inputs using only one object drawing and object rotation*
5.2. Testing datasets

Compared to the CNN training, which involves the need for hundreds or thousands of samples in the dataset, testing one does not necessarily need to be quite as big, or even close to that size. It all depends on what kind of testing is being performed.

As for the CNN which is the focus of this thesis, the number and the source of testing samples has varied a lot during the development process. At the early stages, even before the whole network was completed, test samples were used to check the functionality of forward pass. Those were very important pieces when it came to fixing the initial network structure.

After the network structure was successfully finished and trained, there was the question of how to see if it was trained long enough and well enough. A simple feature was developed that does only a forward pass of the input data from the drawn object (Figure 5.4). By clicking in the UI button “Te”, the data forward pass is calculated and the result from output vector is shown in the console. This is also the equivalent of in-game testing that was previously described in chapter 4.5.

Figure 5.4 Testing the trained CNN network with UI button
An important thing to note when discussing testing is that, if the functionality is in its early development stages and being developed by only one person, as is the case in this thesis, and most of the samples from the training dataset were created by the same person, it is advisable to give one or more other people a task of creating test samples for testing dataset. Every person has a different style of writing and drawing, and if the CNN should be able to recognize objects that come from different sources (players), then different people should also be involved in creating the training and testing datasets.

So far in this chapter, only testing datasets of one sample at the time were mentioned. There was an idea to create a system that would collect multiple testing samples from multiple users, much like the one for training datasets, and store them together so they could be used later. Because of the lack of development time, this still remains as one of the top priorities in future work on this functionality.
6. Performance Analysis

This chapter will give a performance analysis of the created CNN structure. The purpose of the analysis is to find a way to optimize the training and testing processes of the network, with the goal of improving the overall functionality.

6.1. Network testing parameters

So far, the whole CNN structure, creating datasets, the training and testing process, Unity scenes and its components have been thoroughly explained. We now focus on optimization of the current CNN setup.

At the start of this project, there was a plan to try out different layers to see how they mix with one another to try to optimize the CNN in that way. However, due to the length of time required for the development process of every one of those CNN structures, this was not achieved in the scope of the thesis and remains as an idea for future work. The connections between layers would need to be different, same with resolutions of inputs and outputs of every layer, backpropagation would need to be calculated differently, and even after all that, the debugging process and fixing all the other CNN parameters would still take time. That being said, even developing 3 or more of this kind of structures would take so much time that there probably would not be enough left to test them thoroughly enough to extract valuable optimization information.

That is why it was decided to focus on one specific CNN structure and tweak its changeable parameters. This way the functionality of the network would be secured and would leave enough time to focus on optimizing certain elements, with the goal being to find out which of them are the most important for the CNN to be able to function well enough.
In this chapter, the parameters that have been worked with will be mentioned and explained, and in the following chapters there will be more details how and in what way were they changed and their influence on the CNN itself. The addressed parameters are as follows:

- weight values initialization
- drawn object input resolution
- padding
- activation function
- number of nodes in fully-connected layers
- learning rate
- training dataset epoch size
- number of epochs

There has been lots of discussion about the importance of weight values in this thesis already, since they are the most dynamic part of the whole network. However, they too need to have certain starting values. Those values are being defined by a programmer at the network initialization, when they are written in their files for the first time.

Drawn object input resolution defines the size of the 2D visited array with boolean values, and furthermore also the size of input to first convolutional layer of the CNN. It also defines the size of background panels that are shown in the game scene, but that is not important for the optimization of the CNN. The lowest resolution value is 32 (32 x 32 array), and the other tested ones are multiplications of 32.

Padding has not been mentioned so far in this thesis because it is an optional feature that can be added if necessary. It is quite intuitive (Figure 6.1) to understand and the function that adds zeros on the positions in the array around the original image is very simple. Padding shown on the figure 6.1 is of value 1, but it can be more, depending on the need.
Activation function is a function that does an additional calculation over any output value of any convolutional or fully-connected layer. Depending on the function, its results can vary, but all of them do the same task, which is to not let the values get too high through the network calculations because that would make the output not viable.

Learning rate is a very important parameter in the CNN training process. As described in chapter 2.2., it is used for calculation of weight value derivative and has a maximum value of 1. It defines how fast the network “learns”, or more precisely, how big are the steps that the weight values are changed for.

6.2. Multiple solutions of the network for data analysis

All of the changeable parameters explained in the previous chapter in some way offer a unique solution of the network. The following chapter will give a detailed look at how these parameters were utilized.

When it comes to initializing the weight values of convolutional layers’ filters and those of fully-connected layers, there are many ways it can be done: they can all be set to the same value, such as 0 or any other number; they can be set to a random value between two numbers; or they can be calculated using a function. The initialization that has been tested for this CNN, and worked well enough that there was no need to try out different solutions, was Xavier initialization [28]. With it, the initial values are affected by the input size of the layer which has that filter or weight values (Figure 6.2).
It was noted that the drawn object input resolution of 32 was used the most. That is mostly because the first convolutional layer was created to accept input array of value 32, and all other layers after that one had their values fixed to follow that first input layer. Nevertheless, the other thing that can be done if the resolution is increased is to transform and recalculate all the values from the bigger array (because of higher resolution) to an array of size 32. That is why all the resolution used were a multiplication of 32.

As for padding, at start of the development of the CNN there were no plans to even start using it, but because of the scaling feature, and after further research it was decided to implement it. Padding is very useful to have when the values are near the edges of the input array. That is a logical conclusion when we look at how the filter is convolving over the input data, the values on the edges are calculated by filter way less times than the values closer to the middle. Padding solves that issue by putting worthless values on the edges.

Another interesting parameter that had not yet been discussed is the number of nodes in fully-connected layers. There is no real indication of how many nodes there should be, so that also offers us different network solutions that can be tested. Adding or subtracting nodes from those layers is quite easy for the CNN structure that was built, the only thing that needs to be adjusted are the sizes of certain input, output and weights arrays. This parameter had been changed when there were moments during the development when the number of objects dropped down from 6. Since the final output layer is also defined as fully-connected in this structure, there are usually 6 nodes for 6 objects. If the objects are decided to be removed from recognition, so should the nodes be. Moreover, then we do not need as many nodes in previous fully-connected layers, so those must be adjusted also.
Last, but not least, is the learning rate parameter. Obtaining the best value of this parameter was approached by testing and keeping track of the weight values in the files to see how much they changed for certain parameter values. If it seemed that the changes were too sudden, learning rate would be lowered, and if the changes would be too small, the parameter would be increased.

6.3. Final network analysis

After it was explained how every parameter influences the network, it is important to come up with certain conclusions. The CNN structure that has been developed as a part of this thesis can be defined by the files in which the values that are being trained are saved, the network input, and the scripts which are used for network calculations. Since the two latter parts are components in the scenes from which the network is being instantiated, the only part that can actually “save” the network state are the generated files.

Through the development process many different network solutions have been saved (Figure 6.3). In a single version, more subversions were created and tried out, with different amount of training datasets, number of epochs and learning rate values to understand what kind of effect they can have. First version was the initial working network, in the second version the activation function was changed from ReLU to hyperbolic (Tanh function). For third version activation function was changed again, this time to LeCun’s Tanh, and the function for weights initialization was changed to Xavier initialization. In the forth version padding was added, and the last and final version was made for the game implementation, but it included only 5 objects, not 6, because one was not necessary at that moment.
First and second versions were hardly useful, from the output after the training it looked like the network does not work at all. Testing results of those initial versions turned out to be abysmal and it was not really clear why. Moreover, additional research has been done and a breakthrough finally happened when different weight initialization function and activation function were added.

As for the initial weight values, Xavier initialization was shown to be invaluable. The solutions in first and second version of the CNN often had either too sudden growth or decrease of certain values (exploding gradient) or they converged to 0 (vanishing gradient) during the training, which resulted with a faulty network. After this initialization was added, those values after the training were changed, but still normalized (Figure 6.4).
Even though ReLU and normal hyperbolic activation functions were performing well for this CNN, it seemed that slightly better output results are being created using LeCun’s Tanh function. That is why this activation function is another optimal parameter that is advised on being used for this CNN.

As the last important part that was added to the network, because of padding in the forth version of the network, the functionality in the game itself that uses the CNN finally seemed like it is performing well enough for this initial version of the game. Players that tried out the object recognition managed to get their object recognized enough times that they kept playing the game, and reported not being annoyed if it sometimes was not recognized.

The testing group through all of the network versions consisted of 6 project team members and 5 to 10 other people. In the first two versions, the network was tested in the training scene, so the output result was shown in a console window, and not a single tester managed to obtain more than 50% of correct recognition with a testing dataset of around 20 samples. That test was repeated for the other version also. Third one’s recognition successfulness was 50-60%, forth one was 70-80%, and the final one had over 80%. The last three versions were tested in the game scene and the unsuccessful recognition was shown by a picture of the wrong object appearing. It has to be noted that the biggest amount of unsuccessful samples were the ones that deviated the most from the ones that were shown to testers of how to draw them.
7. Conclusions and Future work

Even with all the unexpected issues that arose through the development, the main goal of the thesis has still been achieved, and that is to discover the crucial elements and structural parts of the CNN that need to be focused more on during the development process. A self-made convolutional neural network has been designed and implemented as a part of the game “Me and SuperBoy”. Its parts, which had been explained in length, were created and connected together. The network was trained, tested, and went through enough iterations for us to be able to understand which parts were optimized, and can be optimized even further.

The main conclusion that can be drawn from this work is that creating a convolutional neural network from the ground up is a very tedious task. The optimization part of this thesis would have likely been easier to conduct if a framework for creating a network would have been chosen, such as Tenserflow. However, an issue with that would maybe arise when it would come to implementing it with Unity, which is why it was decided not to use it in the first place.

Once the foundation has been laid, it will be much easier to continue the development process while knowing every specific network feature inside out. There are many possibilities of continuing the work on this CNN structure, and out of which many will hopefully come to fruition as a part of the ongoing development of the game.

As for the network parameters, the crucial ones that, when changed, improved the network the most were, activation functions, padding and a way to initialize weights. Out of those, the biggest improvements were caused by setting the padding to input image drawn by player and changing the weight initialization to Xavier initialization. This will provide much more clarity in the next stages of the development of the game functionality. It is important to note that the ones that were explained are suitable for this structure and purpose of a CNN, which does not necessarily need to be true for other similar projects.
The two different Unity scenes in which the network has been trained and tested proved to be quite a struggle, but that was mostly the fault of the project team, the programmers, that did not plan for certain issues that happened or agreed upon certain design aspects that had to be adjusted when transporting the network from training scene to the testing one (the game version).

As for future work, there are couple of key elements that will be focused on. It was already mentioned in chapter 5.2 that the testing datasets and testing process have not been utilized completely, so that will be the main focus in the continuation of the work. Testing datasets need to be collected from players that will play the game, so a structure that gives them the opportunity to draw their objects and save them for further use is necessary. With that data, a more detailed analysis can be performed and the network will be able to become even more optimized as a result.

Even further goals will be oriented towards experimenting with different CNN structures, but before that the current scripts will need to be cleaned up and commented in a right way to allow easier work in the future or if other people will continue the work. It is not necessary to experiment with changing the current layer structure, but more experimentation with inputs and outputs in Unity, as well as the parameters is advisable.
Bibliography

URL: https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6 (last accessed April 10\textsuperscript{th} 2019)


URL: https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/ (last accessed April 15\textsuperscript{th} 2019)

URL: https://blog.goodaudience.com/convolutional-neural-net-in-tensorflow-e15e43129d7d (last accessed May 20\textsuperscript{th} 2019)


URL: https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening (last accessed May 22\textsuperscript{nd} 2019)


Bhandare, Ashwin; Bhide, Maithili; Gokhale, Pranav; Chandavarkar, Rohan. “Applications of Convolutional Neural Networks”, IJCSIT, Vol. 7 (5), 2016.


AlphaGo
URL: https://deepmind.com/research/alphago/ (last accessed June 21st 2019)

Unity game engine
URL: https://unity.com/ (last accessed June 21st 2019)


[31] “Games market expected to hit $180.1 billion in revenues in 2021”. Takahashi, Dean. April 30th 2018.


Abstract

This thesis covers the development and analysis of a convolutional neural network and its function in a digital game. The purpose of the network is the recognition of 6 different objects that a player can draw during gameplay. The thesis describes the motivation for this work, problems that were faced, and a description of possible solutions. The emphasis is given on why certain choices have been taken, and why certain technologies were used to try to solve the issues.

Through the main part of the thesis, the theoretical background for those technologies is given, focusing on explaining the convolutional neural network, together with their use in the digital games industry, and their actual implementation in the Unity game engine software. All the structures of the CNN, how they are connected, and the processes of network training and testing are explained. The reasons are given as to why the network structure is decided to be implemented in a certain way and all parts of the implementation are explained in length. Finally, an explanation is provided of datasets collected for the training and testing of the network.

The last part of the thesis covers the network parameters, how they can change the network, and the analysis which results in finding the optimal values or structures of those parameters. At the end, the conclusions are drawn and the plan for the possible future work is set.
Sažetak

Ovaj diplomski rad obuhvaća proces izrade i analizu konvolucijske neuronske mreže koja se koristi za funkcionalnost u digitalnoj igri. Cilj mreže je prepoznavanje 6 različitih objekata koje igrač može nacrtati. U uvodu je objašnjena motivacija za projekt, problemi s kojima se suočavamo, te završava s prikazom mogućih rješenja. Naglasak je stavljen na to zašto su donesene određene odluke i zašto se koriste određene tehnologije za rješavanje problema.

Kroz glavni dio rada dan je teorijski pregled tih tehnologija, uz stavljanja konvolucijske neuronske mreže u prvi plan i njihovu uključenost u industriji digitalnih igara, te njihovih stvarnih primjena u programskom alatu za izradu digitalnih igara Unity. Objašnjene su sve strukture mreže, kako su međusobno povezane, te procesi treniranja i testiranja mreže. Navedeni su razlozi zašto je odlučeno za određenu strukturu mreže i svi njeni dijelovi su detaljno objašnjeni. Glavni dio završava s poglavljem koje opisuje sakupljanje podataka za treniranje i testiranje mreže.

U zadnjim su poglavljima objašnjeni parametri mreže, kako njihova promjena utječe na mrežu te analiza koja rezultira pronalaskom optimalnih vrijednosti i struktura tih parametara. Na kraju su opisani doneseni zaključci te je postavljen plan za budući rad.