Moonens Laurens

Neural networks applied to game AI

Graduation work 2018-19

Digital Arts and Entertainment

Howest.be

CONTENTS

ABSTRACT

This paper will discuss the fundamental structure of neural networks, what they are, how they are used, and more specifically, how they can be trained to become a simple game AI. This paper is about the process of making a neural network framework in C++, using a simple game engine as basis.

The two main learning algorithms will be discussed, being back propagation and a genetic algorithm. These methods will be completely dissected, to their mathematical foundation.

The result is a working pacman AI, that is trained using both back propagation and a genetic algorithm. These algorithms can be applied separately, or combined to reinforce each other.

No prerequisites of programming or neural networks are needed, but a basic knowledge of math, linear algebra to be specific, will definitely come in handy.

INTRODUCTION

In recent years, neural networks have become the staple of computer AI's. New advancements are made yearly, resulting in more complex and performant AI's. Most of these systems are built on the fundamental structure of neural networks. This paper will attempt to give you a better, in depth understanding of this structure: what neural networks consist of, what they can be used for, how they are used, … This paper will also dive deeper into the calculus behind neural networks, and how these obscure intelligences can be reduced to a lot of numbers and equations.

The paper will explain 2 main ways of training neural networks: back propagation and genetic algorithms. Their main differences, advantages and uses will be considered, as well as their detailed mathematical meaning.

After deepening our knowledge into the theoretical essentials of neural networks, the paper will discuss a C++ framework, used to create and train neural networks, built using a small game engine. These neural networks can be used as a simple game AI.

Based on the results of making this framework and these training cases, an attempt will be made at understanding why neural networks are not often used in games (yet), and possibly how they could become more useful in games.

RESEARCH

1. BASIC NEURAL NETWORK STRUCTURE

1.1. NEURONS AND WEIGHTS

Neural networks consist of two main elements: neurons and weights. Neurons can be simplified as an element that holds a number, nothing more, nothing less. Weights are connections between neurons, and these weights hold a value as well. These values (both for neurons and weights) most commonly are between -1 and 1, but this is no strict rule. We can interpret a neuron having a value of 1 as being very positively active, a value of -1 being very negatively active, and a value of 0 not active at all. The same applies for weights, but rather then talking about active or inactive weights, we talk about weights with a strong positive / negative connection between two neurons, or a weak connection between them.

As mentioned, a neuron is simply a cell that holds a number, the neuron's output value. This output is the result of a lot of inputs. These inputs are from weights, connected to other neurons. This means that the neuron's output is affected by each weight that this neuron is connected to, as well as the individual neurons on the other sides of these weights. The way we calculate the value of a neuron Y, is by multiplying the values of all the connected neurons X_i, with the weight W_i that connects those two neurons, and adding al the results together. The result of this sum is then pushed through an activation function (see next chapter). The result of this function is the output of the neuron. Figure 1 visualizes this calculation very well.

$$Output = f(x_1 * w_1 + x_2 * w_2 + \dots + x_n * w_n)$$

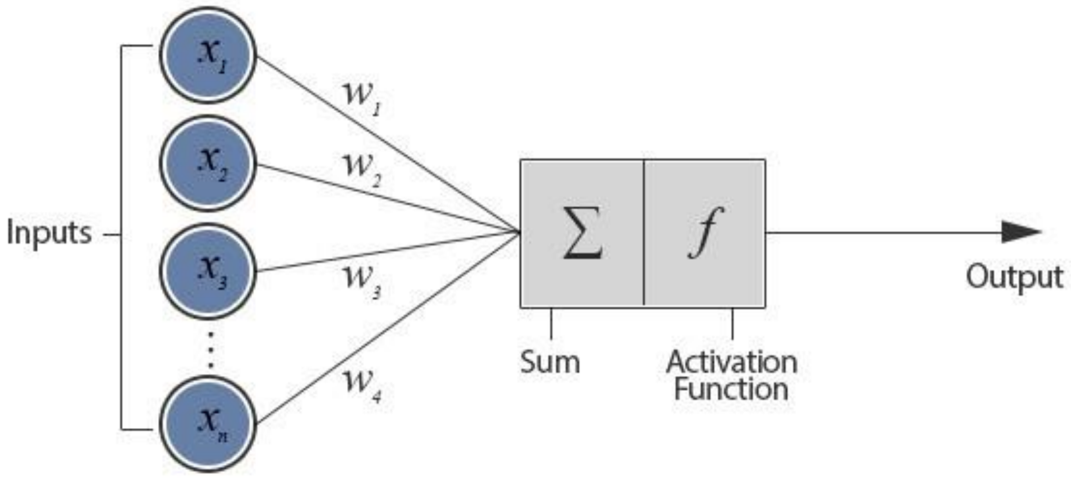


Figure 1: The most basic structure of a neuron

1.2. ACTIVATION FUNCTIONS

The activation function of a neuron is a function that is used to keep the neuron's output under control. It takes the sum of the neuron's input (the weighted sum of the connected neurons), and applies a specific function to it. There are multiple options for this function, and it is up to the designer of the network to choose an appropriate activation function, based on the type of neural network and the application of the network. Below, the most commonly used activation functions are summed up, as well as their derivative. The derivative of a function at a given point, can be described as the slope of the function at that given point. If the derivative is positive at a certain point, that means that the function's slope is uphill at that point (going from left to right). Accordingly, if the derivative is negative, the function's slope is downhill. (See figure 2)

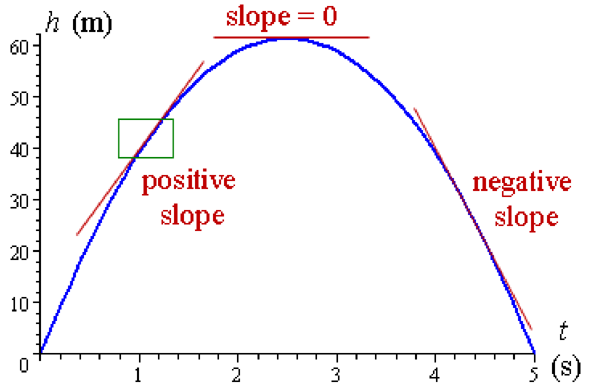


Figure 2: Visualization of the derivative of a function

1.2.1. LINEAR

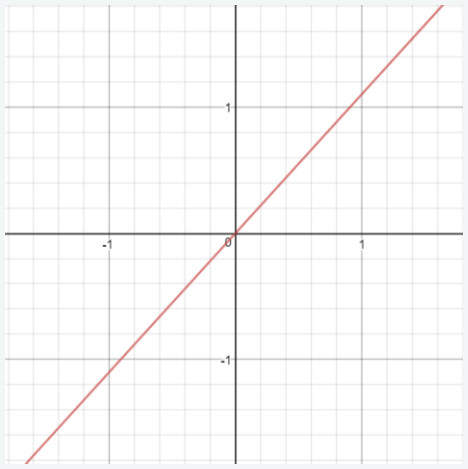
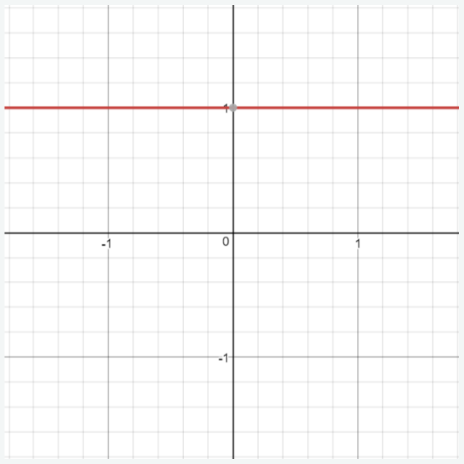
Function	Derivative
$R_{(x,m)} = m * x$	$R'_{(x,m)} = m$
	

Figure 3: Linear function and its derivative

1.2.2. RELU

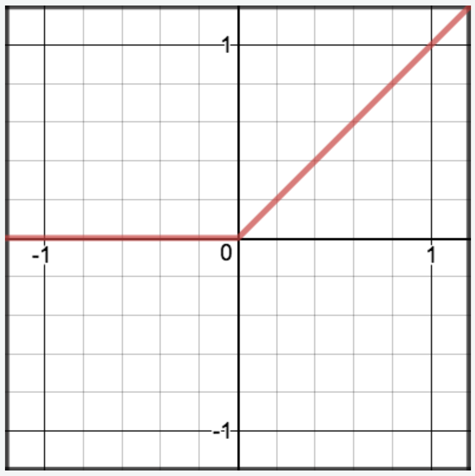
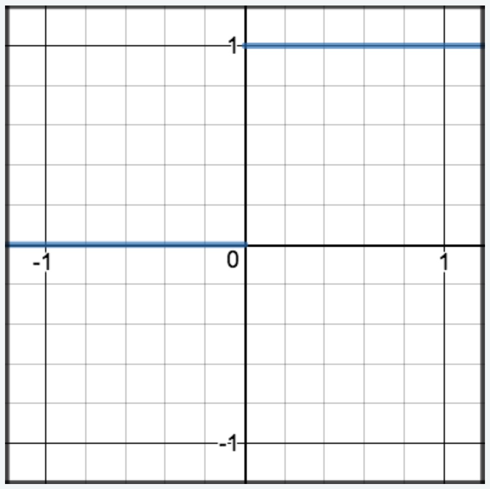
Function	Derivative
$R_{(x)} = \begin{Bmatrix} x > \ 0 => x \\ x \leq 0 => 0 \end{Bmatrix}$	$R'_{(x)} = \begin{Bmatrix} x > \ 0 => 1 \\ x < 0 => 0 \end{Bmatrix}$
	

Figure 4: Relu function and its derivative

1.2.3. PRELU

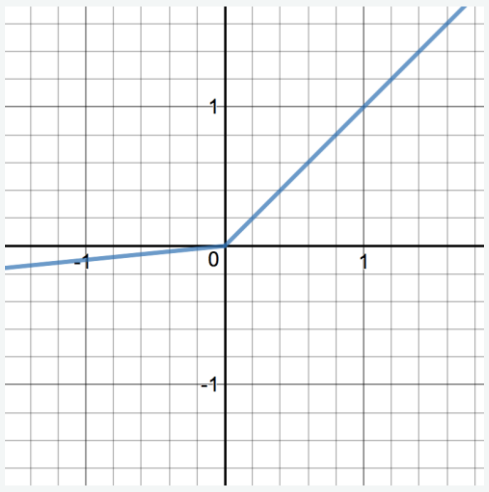
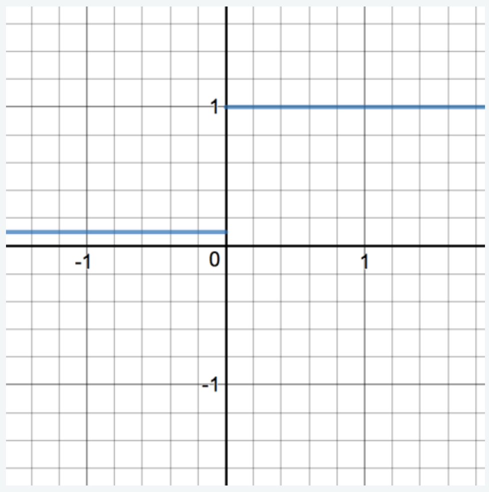
Function	Derivative
$R_{(x)} = \begin{Bmatrix} x > 0 => x \\ x \leq 0 => \alpha x \end{Bmatrix}$	$R'_{(x)} = \begin{Bmatrix} x > 0 => 1 \\ x < 0 => \alpha \end{Bmatrix}$
	

Figure 5: Prelu function and its derivative

1.2.4. TANH

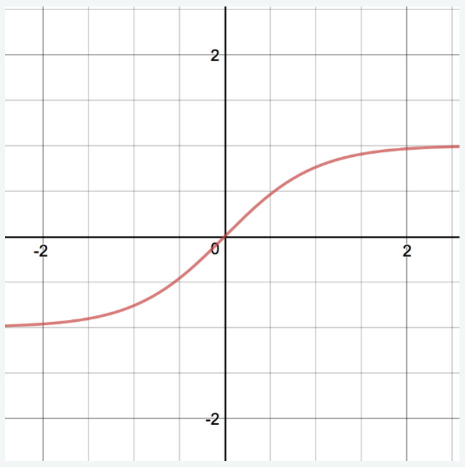
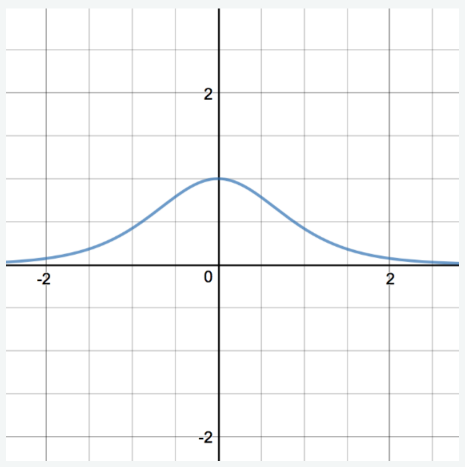
Function	Derivative
$tanh_{(x)} = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$	$tanh'_{(x)} =$
	

Figure 6: Tanh function and its derivative

1.3. LAYERS

One neuron on its own isn't very useful or interesting, but combining a lot of these together can have a whole plethora of applications. These neurons can be arranged in a lot of different ways, some neurons can even loop back in itself as input. The possibilities are endless, but every architecture has its strengths and weaknesses. In this paper, the deep feed forward architecture (DFF) is the one that will be discussed. This is by far the most well-known architecture and forms the basis for a lot of more advanced architectures.

In a DFF, the neurons are grouped in layers. A layer contains a certain amount of neurons, which can get input from the previous layer and send its output to the next layer.

The very first layer is the input layer, used to send the initial input to the network. This input is then send through the network, and ends up in the last layer, the output layer, where the output can be used for whatever application the network is trained for.

In between the input and output layers, we have a certain number of hidden layers. The first hidden layer takes the values from the input layer, and sends it to the next layer. Accordingly, the last hidden layer sends its output to the output layer. The smallest feed forward network that can be considered "smart" has at least one hidden layer.

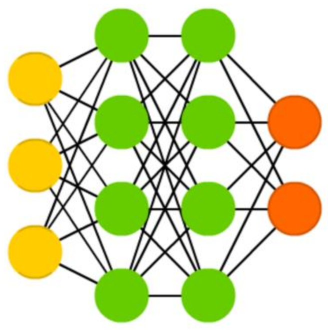


Figure 7: The deep feed forward structure

In a deep feed forward network, we assume that all the neurons of a layer are connected to the previous layer, but there are other network structures that don't follow this rule.

1.4. BIAS NEURON

A very common addition to the layers of a neural network, is a bias neuron. Bias neurons are an additional neuron for every layer. In contrast to normal neurons, bias neurons are not connected to the previous layer, and have a constant output value of 1. Bias neurons give the neural network the ability to offset certain neuron's activation sensitivity, by being connected to a bias neuron with a strong positive/negative weight. If a neuron is connected to a bias neuron from the previous layer with a strong negative weight, this means that the other inputs have to be stronger positive values for the neuron to activate.

1.5. FEED FORWARD

A neural network can be seen as a big function, taking input through the input layer, processing that input in the hidden layer(s), and returning a result through the output layer. This process of a neural network taking input, processing it and returning something is called feed forward.

We can split up the feed forward pass of the network, in the feed forward passes of each layer, and those in turn can be split up in feed forward passes of each neuron.

A neuron's feed forward pass is pretty simple, and is explained in *1.2 Neurons and weights*. To summarize once again, a neuron takes the outputs from the connected neurons, and multiplies those individual values with their connecting weight value. These results are summed up and pushed through the neuron's activation function.

A layer's feed forward pass is simply the feed forward pass of all its neurons. The neurons take the previous layer as input.

Subsequently, a neural network's feed forward pass, is the input layer's feed forward pass, followed by the hidden layers, and finally the output layer's feed forward pass.

1.6. LEARNING

The real potential of neural networks, lies in their ability to "learn". There are a lot of different algorithms used to achieve this. The main thing to understand here, is that the only elements that can change about a neural network to make it "learn", are the weights. It's these connections that influence the behavior of the network. The difficult part is knowing which weights to change, and how to change them.

2. BACK PROPAGATION

2.1. TRAINING DATA

The first training algorithm that will be discussed is back propagation. This algorithm uses training data for the neural network. This data is a collection of input values for the network, and the expected output for that input. For example, if we want to train a network to be a XOR gate (2 inputs and 1 output), the training data would look like this:

INPUT	EXPECTATION
0 0	0
0 1	1
1 0	1
1 1	0

2.2. ERROR FUNCTION

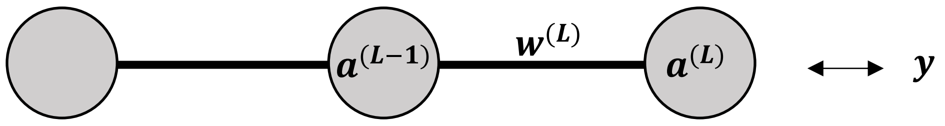
When training the network, we randomly pick input from the training data, and give this input to the neural network. The network will in turn give an output value. By comparing the network's output to the expected output, we can calculate the network's error. The formula for this is as follows. We take the individual differences between an output neuron's value, and the expected output, and do this for all the output neurons. We then sum up all these values, and square the result:

$$Error = \left((x_1 - y_1) + (x_2 - y_2) + \ \dots + (x_i - y_i)\right)^2$$

where x_i is the network's output from a certain output neuron, and y_i is the expected value for that neuron.

2.3. NEURON'S ERROR

As mentioned, the feed forward pass of a neural network, can be seen as all the feed forward passes of its individual neurons. This also applies to back propagation. The back propagation pass of a neural network can be seen as all the back propagation passes of the individual neurons. Let us use this convenience, and start by looking at the back propagation pass of a very simple network:



where:

$a^{(L)}$	= the output from the current neuron we are evaluating
y	= the expected output we are evaluating the neuron against
$a^{(L-1)}$	= the output from the neuron from the previous layer
$w^{(L)}$	= the weight connecting $a^{(L)}$ and $a^{(L-1)}$

Using these values, we can calculate

C_0 = the cost of neuron $a^{(L)}$ => $C_0 = (a^{(L)} - y)^2$

As discussed, the output of a neuron is influenced by its activation function σ.

=> $a^{(L)} = \sigma(z^{(L)})$ where $z^{(L)}$ is the pre-activated output of the neuron, or in other words, the weighted sum of the connected neurons, without passing through the activation function.

As said before, a neuron's output is also affected by the previous neuron and the weight connecting them.

=> $$\boldsymbol{z^{(L)} = w^{(L)} * a^{(L-1)}}$$

In the next image, you can find all of these formulas together, along with a diagram. This diagram represents which values contribute to other values. For example, $a^{(L-1)}$ and $w^{(L)}$ both contribute to the value of $z^{(L)}$.

$$\begin{tikzcd}
\boldsymbol{w^{(L)}} \arrow[dr, no head] & & \boldsymbol{a^{(L-1)}} \arrow[dl, no head] \\
& \boldsymbol{z^{(L)}} \arrow[d, no head] & \\
\boldsymbol{y} \arrow[dr, no head] & \boldsymbol{a^{(L)}} \arrow[d, no head] & \\
& \boldsymbol{C_0} &
\end{tikzcd}$$

$$\boldsymbol{z^{(L)} = w^{(L)} * a^{(L-1)}}$$

$$\boldsymbol{a^{(L)} = \sigma(z^{(L)})}$$

$$\boldsymbol{C_0 = (a^{(L)} - y)^2}$$

In the learning progress of this network, our goal is to affect the weights of the network, in this case being $w^{(L)}$. How this weight should be changed, directly depends on the effect it has on the final cost C_0. We want to understand how the cost value changes, if we apply some change to the weight. This relationship between values is the derivative $\boldsymbol{\partial}$. We are looking for:

$$\boldsymbol{\frac{\partial(C_0)}{\partial(w^{(L)})}}$$

As seen on the diagram, there are some values in between that also have their influence on the final cost. This means we have to split up the relationship between $w^{(L)}$ and C_0, into their smaller, indirect relationships:

$$\frac{\partial(\boldsymbol{C_0})}{\partial(\boldsymbol{w}^{(\boldsymbol{L})})} = \frac{\partial(\boldsymbol{z}^{(\boldsymbol{L})})}{\partial(\boldsymbol{w}^{(\boldsymbol{L})})} * \frac{\partial\left(\boldsymbol{a}^{(\boldsymbol{L})}\right)}{\partial(\boldsymbol{z}^{(\boldsymbol{L})})} * \frac{\partial(\boldsymbol{C_0})}{\partial(\boldsymbol{a}^{(\boldsymbol{L})})}$$

Calculating these relative derivatives, will allow us to calculate the relation between $w^{(L)}$ and C_0. Again, the diagram can be quite helpful to understand where these relations come from.

$\frac{\partial(\boldsymbol{C_0})}{\partial(\boldsymbol{a}^{(\boldsymbol{L})})} = \boldsymbol{2}\left(\boldsymbol{a}^{(\boldsymbol{L})} - \boldsymbol{y}\right)$ => $2(x)$ is the derivative of x^2, which is our error function.

$\frac{\partial(\boldsymbol{a}^{(\boldsymbol{L})})}{\partial(\boldsymbol{z}^{(\boldsymbol{L})})} = \boldsymbol{\sigma}'\left(\boldsymbol{z}^{(\boldsymbol{L})}\right)$ => σ' is simply the notation for the derivative of the activation function σ.

$\frac{\partial(\boldsymbol{z}^{(\boldsymbol{L})})}{\partial(\boldsymbol{w}^{(\boldsymbol{L})})} = \boldsymbol{a}^{(\boldsymbol{L-1})}$ => the effect $w^{(L)}$ has on $z^{(L)}$ depends on the output of $a^{(L-1)}$.

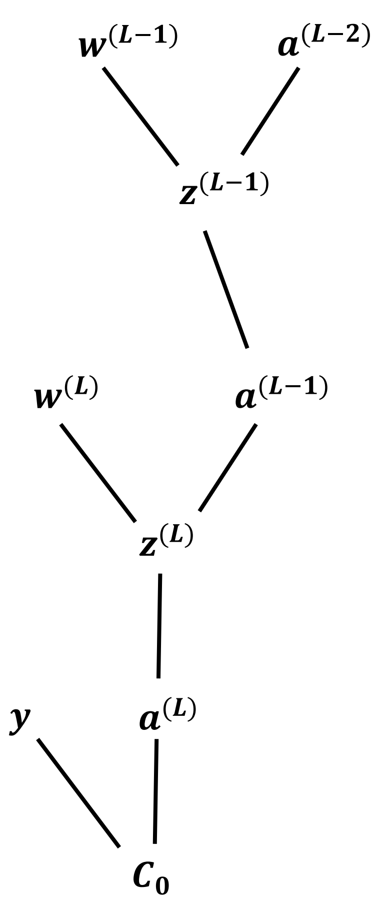
The resulting value, describes how much the network's error is affected by this specific weight. To decrease the network's overall error, we have to subtract this weight's error from its initial weight value. In practice, it is a good idea to store this change in weight in a delta weight value. This is because the final change in weight might also be affected by other neurons.
It is also a good idea to keep track of the neuron's error $\frac{\partial(C_0)}{\partial(a^{(L)})}$, which comes out to be $\frac{\partial(a^{(L)})}{\partial(z^{(L)})} * \frac{\partial(C_0)}{\partial(a^{(L)})}$. This value is used to propagate backwards to previous layers.

2.4. PROPAGATING BACKWARDS

So far, we only calculated the error of the weight, $w^{(L)}$, connecting $a^{(L-1)}$ and $a^{(L)}$. This weight is directly connected to the output layer. To recap, that weight's effect on the final error is influenced by:

- the expected output (and in turn the error function)
- the neuron's activation function
- the previous neuron connected to this weight

Back propagation is the operation of applying this formula to previous layers, while taking into account the effect of the next layer. This means that the effect of previous layers on the final cost, depends on the effect of next layers on that final cost. This can be better understood by expanding the diagram, with the previous layer.



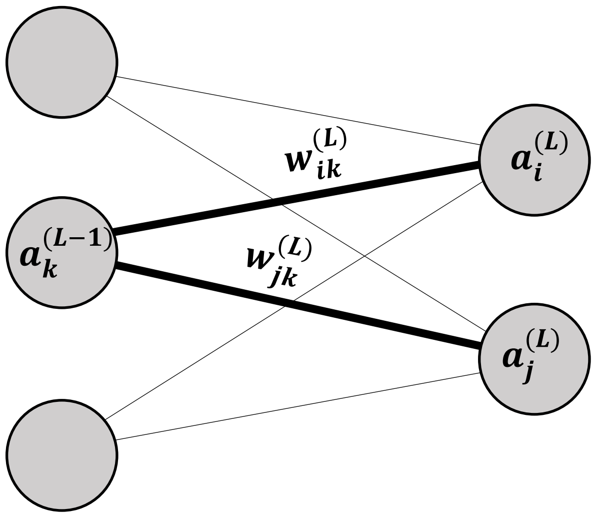
As you can see, the effect that $a^{(L-1)}$ has on the final cost, depends on the effect of $w^{(L)}$. Even though we cannot directly influence $a^{(L-1)}$, it is wise to keep track of. What we can influence, is the value of $w^{(L-1)}$. And for this we need to know its relation to the final cost, which can be calculating by, again, splitting that relation up into its relative relations:

$$\frac{\partial(C_0)}{\partial(w^{(L-1)})} = \frac{\partial(z^{(L-1)})}{\partial(w^{(L-1)})} * \frac{\partial(a^{(L-1)})}{\partial(z^{(L-1)})} * \frac{\partial(C_0)}{\partial(a^{(L-1)})}$$

These individual relationships are very similar to the first layer we evaluated. The main difference is the last element $\frac{\partial(C_0)}{\partial(a^{(L-1)})}$, the relation between $a^{(L-1)}$ and the final cost, or in other words, the error of $a^{(L-1)}$. Luckily, we already calculated this value in the previous equation. By keeping track of this neuron's error, we can easily apply this formula over and over again, propagating backwards through the network.

2.5. BIGGER NETWORKS

The network we looked at so far, only has one neuron per layer. This made the calculations a bit simpler, since every neuron only influences one neuron, and only gets influenced by one neuron. Despite this, the calculations for bigger networks don't really become a lot more difficult.



An additional identifier is added to every neuron and weight, since there are multiple neurons per layer. Despite this, the formula for a weight's effect on the network's cost, is essentially the same:

$$\frac{\partial(C_0)}{\partial\left(w_{jk}^{(L)}\right)} = \frac{\partial\left(z_j^{(L)}\right)}{\partial\left(w_{jk}^{(L)}\right)} * \frac{\partial\left(a_j^{(L)}\right)}{\partial\left(z_j^{(L)}\right)} * \frac{\partial(C_0)}{\partial\left(a_j^{(L)}\right)}$$

What does change, is the effect a neuron from the previous layer has on the final cost. This is because this neuron is connected to multiple neurons in the next layer. Neuron $a_k^{(L-1)}$ for example influences the cost both through neuron $a_i^{(L)}$ and neuron $a_j^{(L)}$. Because of this, we have to calculate the error for neuron $a_k^{(L-1)}$, both with respect to $a_i^{(L)}$ and $a_j^{(L)}$, and these have to be summed up, to get the overall cost of neuron $a_k^{(L-1)}$.

2.6. WEIGHT UPDATE

Because neurons are (most of the time) connected to multiple neurons, their error also depends on these multiple neurons. This also applies to the weights. Not because they're connected to multiple neurons, but because the neurons they connect to are. This means it is a good idea to not directly change the weight value, when doing the neural network's back propagation pass. It is much more preferable to store a delta weight for every weight, and affect that value. After the back propagation pass, all these delta weights are applied to their respective weight.

2.7. BATCH TRAINING

As mentioned, back propagation relies on training data, consisting of input and expected outputs. Ideally, the neural network would loop over all training examples, storing their desired delta weights, and averaging them. However, this is computationally slow, especially the more data you have. Because of this, the network trains in data batches. These are a random selection of training samples that the network will train on once, average their desired delta weights, and then applying those delta weights to the actual weights. The side effect of this is that the learning will not happen in the most optimized way, but by selecting a good batch size, the increase in computation speed makes up for this.

2.8. LEARNING RATE

The learning process of back propagation, is the attempt to minimize the networks cost. This can be visualized as a valley in the graph, where the bottom of the valley is the lowest cost. The goal is to get as close to the bottom of that valley, by using the derivative, or gradient, of the graph.

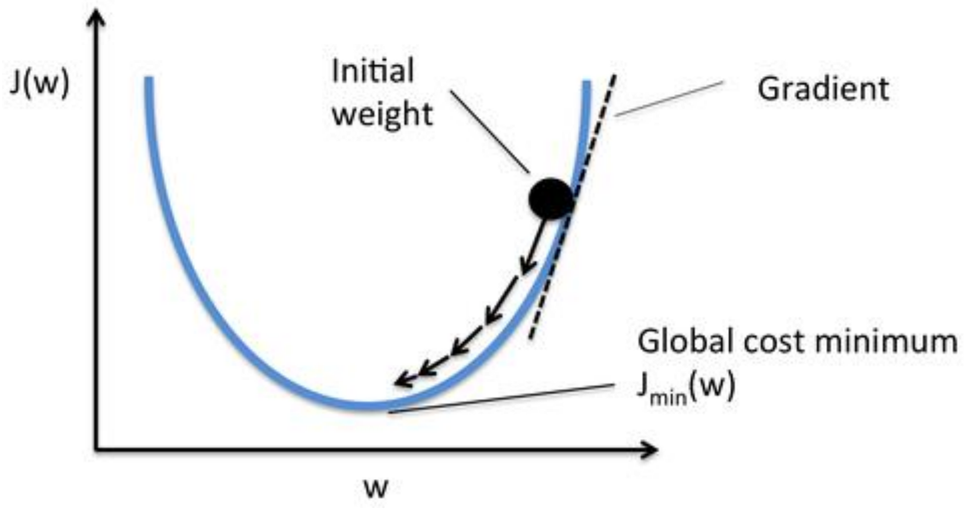


Figure 8: 2D visualization of the back propagation process

When training using back propagation, the training can be influenced by a learning rate. This is a value that is multiplied to the delta weight, every time right before the delta weight is applied to the weight value. For example, with a learning rate of 0.1, the change in weight is always divided by 10. This is helpful to avoid overshoot.

Overshoot is the event where the change in weight is too big, causing the cost to go past the lowest point of the cost graph. Because of this, it is best to lower the learning rate, when approaching the function’s valley. This increases the accuracy of the learning process.

3. GENETIC ALGORITHM

3.1. ORIGIN

The second approach to training neural networks that is discussed in this paper is the genetic algorithm. As the name implies, this algorithm is very much inspired from Charles Darwin's evolutionary theory, as well as the stages of chromosomal crossover (the exchange of genetic material between 2 chromosomes).

3.2. GENERAL GENETIC ALGORITHM

The genetic algorithm can be applied to a lot of structures, not only neural networks. The core of the genetic algorithm involves "survival of the fittest" and crossover. This means that no training data is required. The solution to a certain problem is simply found through a lot of trial and error, and random chance.

In figure XXXXX you can see the general structure of the genetic algorithm. It starts off with a random population with a certain population size. This population is evaluated for the given problem. A fitness function determines how well a certain individual of the population is doing.

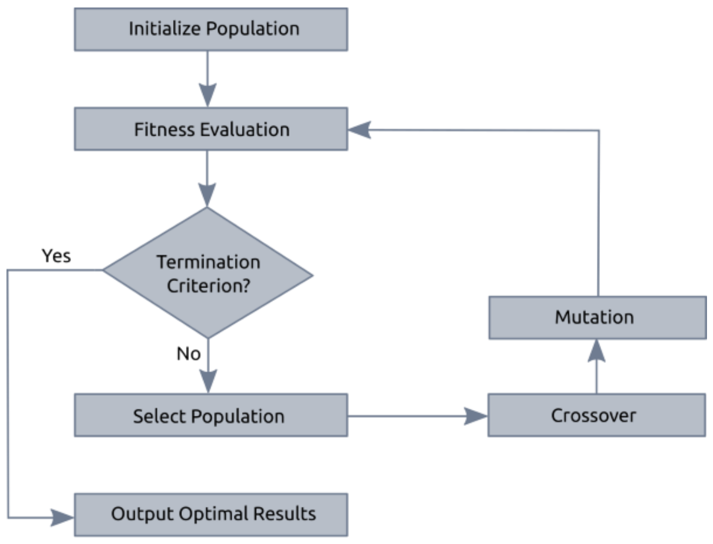


Figure 9: The main steps of a genetic algorithm

After evaluating all individuals, the population is sorted based on their fitness. Next, the "natural selection" process happens. This involves eliminating a certain percentage of the population, starting with the individuals with the lowest fitness. The surviving individuals continue to the next generation.

Because some individuals were eliminated, the population has to be filled again, to ensure a steady population size. This is done by using the surviving individuals, and creating new individuals from those. These new individuals also get a slight mutation, encouraging random trial and error.

This process is repeated with the new generation over and over again, until a termination criterion is reached. This can be a certain fitness value at which the individuals can be considered trained, or a sign that the learning rate has slowed down too much, resulting in no further improvement, or something else as criterion.

3.3. POPULATION

In case of neural networks, a population consists of multiple neural networks. This means that one individual is one neural network. A randomly generated neural network is a network with random weight values. Some models also allow for random amounts of layers and neurons, but that will not be discussed in this paper.

3.4. EVALUATION

The evaluation process of neural networks strongly depends on their application. Some neural network only need to be fed forward once to evaluate the network, in the XOR gate network for example. In other applications, like games for example, the network needs time to be evaluated, since it’s not possible to tell how well it is doing, only based on one action of the network.

Evaluating an individual simply means giving it a score, or fitness. This fitness is calculated using a specialized fitness function. This function strongly depends on the application as well, and should be carefully thought out, to ensure efficient learning progress, and a desired result.

3.5. SELECTION

The individuals of the current generation are then sorted based on their fitness. Next a certain percentage of the generation is eliminated. The optimal selection percentage can vary a lot depending on the problem. The selection can also vary from a hard selection to a gradient selection, where some lucky bad performers still live on to the next generation, while sacrificing some better performers.

3.6. CROSSOVER

Crossover is a process that allows us to combine two different individual’s properties into a new individual. These properties depend on the application of the genetic algorithm, but for neural networks, these are the weights of the network, since these are the randomly generated values from the initial population. As mentioned before, there are models that incorporate random amounts of neurons and layers as well. The crossover process should also take this into account. The rule is that every aspect of an individual that can vary compared to other individuals, is a candidate aspect for crossover, as well as mutation, but more on that later.

Before being able to cross over anything from two parents, we need to select two parents from the surviving individuals from the previous generation. This selection can either be completely random, or it can have a bias towards the individuals with higher fitness values.

Next, both parent’s aspects that are candidates for crossover, have to be lined up in a bit-string. For neural networks, these strings are simply a collection of their weight values. These strings can be considered the DNA of the individual. Very similar to the actual biological crossover process, these two DNA strings are combined. There are 3 ways to combine these.

The first method is called one-point crossover. Both parent bit-strings are split at the same point. The child individual is then built up from the first part of the first parent and the second part of the second parent.

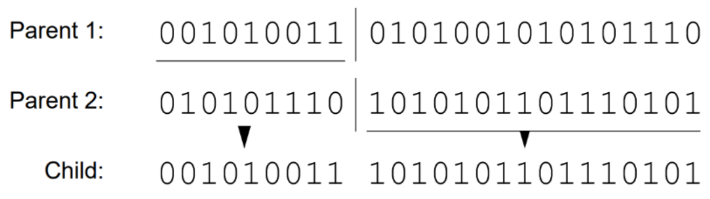


Figure 10: One-point crossover

The second method is two-point crossover. This method is very similar to one-point crossover, just (as the name says) splitting the bit-strings in 2 random positions. The 2 outer parts are copied from the first parent, and the middle part from the second parent.

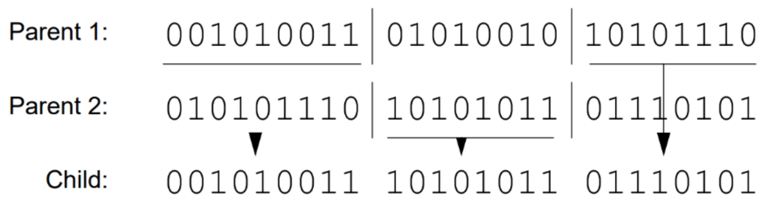


Figure 11: Two-point crossover

The final method is uniform crossover. Here, for every bit in the string, it is randomly decided from which parent it should come.

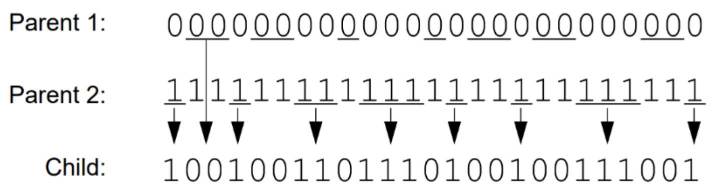


Figure 12: Uniform crossover

Despite general trends, crossover is not a mandatory step. In some cases the genetic algorithm can perform better by ignoring the crossover stage, and simply replenish the population by cloning random individuals from the previous generation. For neural networks, this can be the case, because of the nature of neural networks, and their interleaved structure. This means that certain behaviors cannot be assigned to a specific set of weights all the time. Neural networks are a very delicate structure, where mixing weights from two parents is not the same as mixing those parents behaviors.

Despite this, even for neural networks, using crossover is not a bad idea. This is because crossover also contributes to the random trial and error, and encourages more variety in the approach to the given problem.

3.7. MUTATION

The newly created individuals also get some slight mutations. This is to increase the random attempts at finding a solution to the given problem. These mutations can be applied to every variable aspect of the individuals. In neural networks these are the weights again. And again, some model incorporate the number of neurons and layers into this as well.

Most of the time, only newly created individuals (as the result of crossover) get mutated. This is to ensure that good candidates from the previous generations don't get mutations that potentially lower their performance. Again, this can vary from application to application.

The amount an individual gets mutated, is called the mutation rate. This rate can be a fixed amount, or a variable amount. It can vary depending on the overall performance of the generation, or it can vary on an individual bases. This means that individuals that originate from well performing parents, get less mutations then individuals with bad performing parents.

For neural networks though, there are two factors that determine the mutation rate: the amount of weights that get mutated, and the amount those weights get mutated. These mutation rates also should be carefully thought out, to suite the application and the problem to solve by the neural networks.

4. GAME AI APPLICATION

4.1. NEURAL NETWORK STRUCTURE

When applying neural networks to a game AI, there are some very important decisions to be considered, that might easily be overlooked or undervalued.

It all starts with choosing an appropriate network structure. How many layers should be used? How many neurons should the network have? Which activation functions are most efficient? Most of the times, there isn't a reliable way to know these parameters right from the beginning. More often than not, a lot of testing is required to find the ideal network structure, and even then, a lot depends on the randomly generated weights it starts with.

4.2. INPUT

For the network to be used as a game AI, it needs to have control over that AI, and have a way to observe the game setting it is in. These factors are represented in the networks output and input respectively.

Choosing the correct input for the network is probably one of the most important decisions to make, since the whole AI depends on that input to function. It is important not to overestimate the power of neural networks. If you, as the user and creator of the game and AI, can help the network a bit, don’t hesitate to do so. Of course this shouldn’t force you to do complicated calculations yourself, in turn defeating the whole purpose of a self-learning AI. For example, if the game AI gets as input its position, and some enemies’ positions, your expectation is that the neural network will eventually recognize the relative position of the enemies, according to his position. This is not a very difficult calculation to do yourself, but it actually is for a neural network. Giving the relative position of the enemies as input, is a lot more efficient, and usable for the neural network.

It is also a good practice to find a proper balance between too many inputs and too few. Give the network too many, and it is too difficult to learn for the network what inputs to pay attention the most, and what inputs to combine into more general data. In the pacman game for example, you could give as input the location of every single pill, but this makes it way too difficult for the network to know what to do with all of that input. It is way more efficient to only give the closest pill as input, making the network a lot smaller and more efficient.

4.3. OUTPUT

Luckily it is quite a bit easier to determine the necessary output of the network. Most of the time, it is possible to directly convert every possible button that a human player can press, to an output neuron for the network. If the output value for a specific button is close to 1, that button can be considered pressed, and if it is close to 0, it is not pressed.

4.4. BACK PROPAGATION

Back propagation is not a popular approach to training a neural network for games. This is mainly due to the need for training examples. Getting training data from games would mean recording every button input from a player, for a given game state. This would mean that a person has to play the game for hours, in order to get enough training data. Another downside, is that the network will learn exclusively from the way that person plays, and be limited by it. Back propagation can never result in a game AI that is better than the examples it is given.

4.5. GENETIC ALGORITHM

The genetic algorithm, applied to neural networks, is a way more popular method of training a game AI. This is because it avoids the need for training data. Also, after many generations, and with carefully thought out parameters (fitness function, mutation rate, …), the genetic algorithm is able to give some exceptional results.

CASE STUDY

1. INTRODUCTION

The best way to learn something, is by doing it. Because of this, I made a neural network framework in C++. By not using any existing libraries, related to neural networks, AI or genetic algorithms, I forced myself to fully understand the logical and mathematical steps, involved in neural networks and AI.

I then applied this framework to a pacman game I made earlier. The goal was to make an AI that was capable of playing pacman, using neural networks, and train it using both back propagation and the genetic algorithm.

2. SCENES

There are 4 scenes, which can be accessed with the function keys F1 to F4.

2.1. NEURAL NETWORK VISUALIZATION

The first step was to make a neural network framework and editor. This allowed me to quickly create, edit, save and load neural networks. The networks also got an intuitive visualization. White colors represent values close to 1, and black represents values close to -1. This both applies to the weights and the neurons.

The save destination and load source files are chosen, by editing an application settings text file. This eliminated the need for a runtime text editor of some kind, which would divert me from the core purpose of this project.

2.2. TRAINING

To train created networks, I created two independent interfaces. One for back propagation training and one for the genetic algorithm.

2.2.1. GENETIC ALGORITHM

The genetic algorithm scene allows the user to load a population of networks, based on a certain

load file. The current individual’s network is displayed, as well as its score. The pacman game is also drawn, but it is run at a much faster rate, to decrease the time it takes to evaluate an individual. Luckily this has no effect on the resulting AI, because no important game states are skipped whatsoever.

There is also a graph displaying 3 values for every generation, being the maximum score, minimum score and average score of that generation.

2.2.2. BACK PROPAGATION

The back propagation scene has the option to load different networks at runtime, as well as loading different training data at runtime. There is a graph, displaying the learning progress. The user has the option to train one step at a time, to be able to evaluate the network, or have a closer look at its performance. Of course there also is an option to train every frame.

2.3. FINAL RESULT

The last step of course is to see the AI in action. In the last scene, the game runs at its usual speed. The AI's network and score are shown.

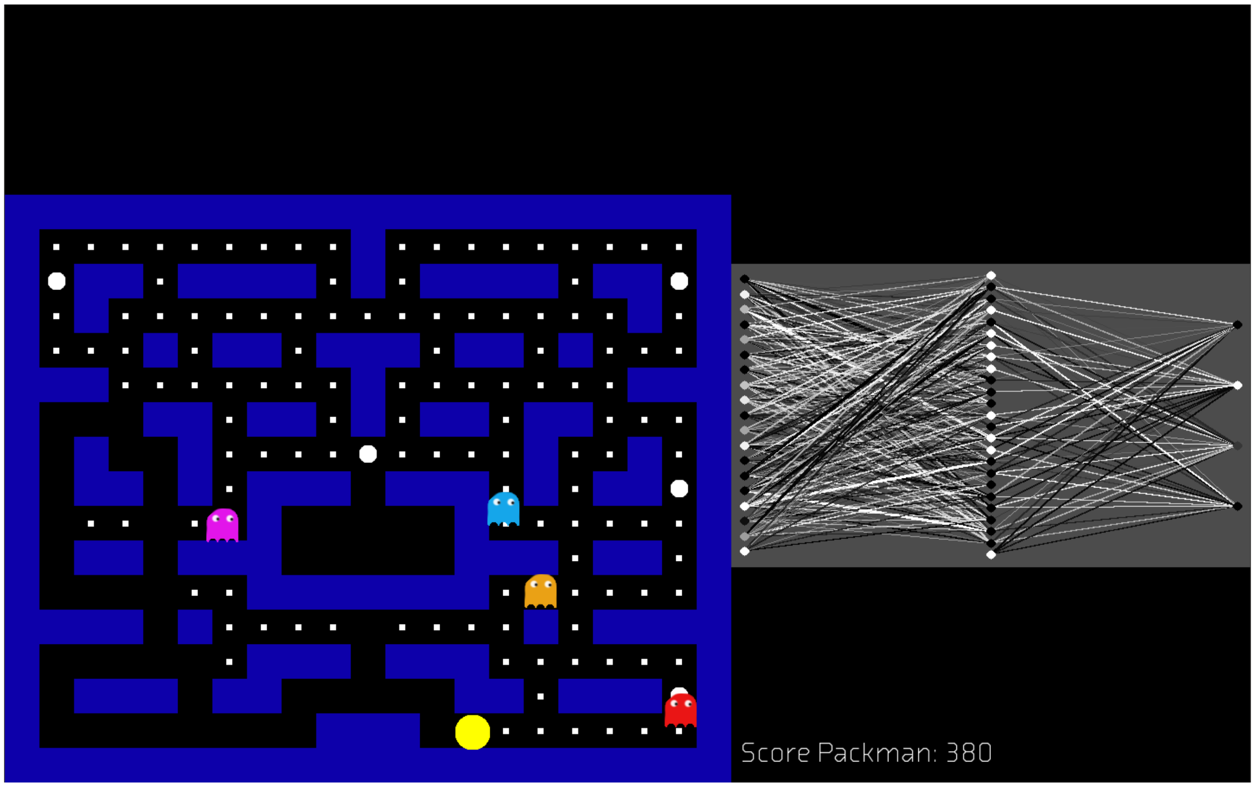


Figure 13: The working AI, controlled by the neural network (right)

3. TECHNICAL APPROACH

3.1. NETWORK-GAME CONNECTION

One the most important decisions to make was what to input from the game to the neural network, and how to interpret the network's output.

The network has 18 input neurons (plus a bias neuron). The first 2 represent the X and Y coordinate of pacman. Next, for every ghosts there are 3 inputs. The X and Y coordinate, relative to pacman, and the state that ghost is in (fleeing / hunting). The final 4 inputs represent what types of tiles pacman is surrounded by.

The network has 4 output neurons, representing the 4 directions pacman can decide to go in. Every frame, the output neurons are checked, and the one with the highest value is considered the desired direction.

3.2. GENETIC ALGORITHM

The genetic algorithm starts off with a random population of 50 individuals. By default, they are loaded from a file, creating a network with one hidden layer with 24 neurons, but this can be changed in the application settings file.

Every generation, the 60% worst individuals are eliminated. The remaining individuals are cloned to the next generation, without mutation. The population is then replenished with new individuals. These are created, by crossing over 2 randomly selected parents. Next, the individual is mutated, taking into account the mutation rate. The number of mutated weights is a constant 100, and the amount of mutation a weight can have, depends on then average score of the two parents.

The graph helps the user decide on how well the AI is doing, and decide when to stop the training.

3.3. BACK PROPAGATION

When working on the back propagation algorithm, it can be hard to know if you're doing a good job, without testing it. Because of this, I started by training a small network to become a XOR gate. This worked very well, proving the functionality of the algorithm.

3.4. COMBINING THE TWO ALGORITHMS

Almost always, neural networks are trained, using either back propagation or genetic algorithms, but not often are these two algorithms combined. My idea to do so, came from the fact that I needed a lot of training data for the back propagation algorithm. For a long time I assumed this had to be done by playing pacman for days. Not the most efficient approach.

A much better approach is to automate the gathering of this data. But for that it is needed to automate the playing of the game as well, pretty much making a game AI. As luck would have it, the genetic algorithm can do just that. After creating a functional AI using the genetic algorithm, this AI can be used to generate training data for the back propagation algorithm.

This might seem quite pointless, training an AI based on an existing AI. Still, there are some benefits and logical reasons for this.

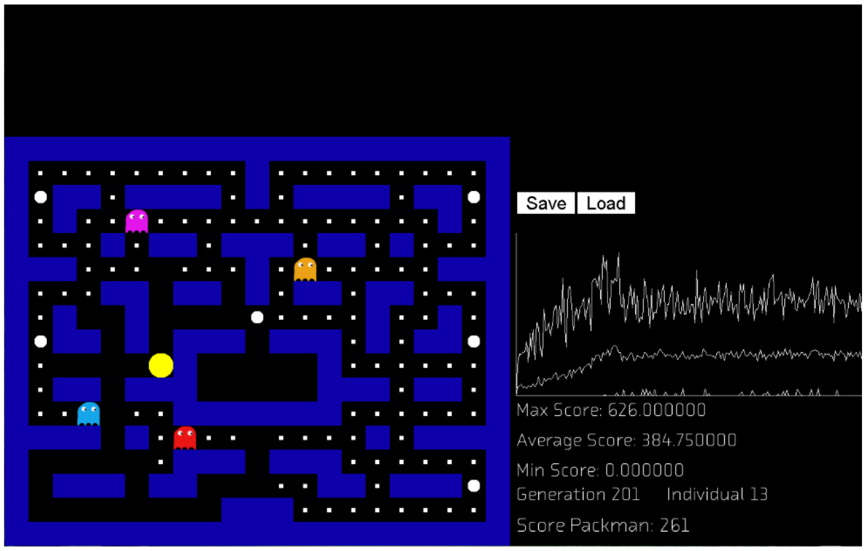
The AI that was trained with the genetic algorithm was not the best player. A lot of the time I bumped into walls, or ran back and forth. When saving the movement from the movement from this AI, I had the ability to discard useless movements, such as movements in a wall's direction. This increased the quality of the training data quite a bit.

Another benefit of combining these two algorithms, is to improve a genetically trained AI even more. The training data I gathered, allowed me to train a new AI from scratch. But it also allowed me to train the existing AI, on the improved training data. This gave the AI a major head start, as well as an increase in quality.

4. RESULT

The result is a working AI. Of course, this can be considered quite subjective. Even though the AI makes some very senseless movements, it is able to perform quite good as well. Sometimes it even reaches scores that I personally cannot beat.

The graphs shown also objectivize this claim, proving the effectiveness of these two algorithms.



Max Score: 626.000000
Average Score: 384.750000
Min Score: 0.000000
Generation 201 Individual 13
Score Packman: 261

Figure 14: The learning progress of the genetic algorithm

CONCLUSION

This project unraveled a lot about neural networks, their core structure and mathematical meaning. The two discussed training algorithms also demonstrated the potential of neural networks.

Even though this project might seem quite complex already, it is barely scratching the surface of what neural networks are capable of. This project gave me a lot of new insight in their function and potential, motivating me to continue their development.

1. IMPROVEMENTS

This was the very first time I made a game AI using neural networks, as well as writing these two learning algorithms. This means I learned a lot on the way as well. Looking back, I notice a lot of possible improvements and additions. For example, both learning algorithms still have a lot of parameters that can be made to adapt on the AI's performance. These include the learning rate, number of mutating weights, and many more. The genetic algorithm also has the potential to become a lot more complex, and possible more efficient.

2. EXTENSIONS

Since the first research, in the late 20th century, neural networks have come a long way. In a way, this project uses quite old technology, but most of the current AI models are heavily based on these "simple" feed forward networks.

Moving forward, there still is a lot to be discovered on my part. There are a lot of different network structures, including recurrent neural networks, and deep convolutional networks. Lately, a lot of new advancements have been made on ways to train neural networks, and a lot of those algorithms, I don't fully understand yet.

3. POTENTIAL IN GAMES

Neural networks are not a popular choice for AI in games, apart from projects that are specifically made to demonstrate neural networks, such as this project. Working on this AI, the reasons for this unpopularity became more clear.

For a start, neural networks require a lot of development and testing, without a lot of certainty for success. Again, success can be quite subjective, but still, it can be very difficult to create an AI using neural networks, that has exactly the behavior the developer are looking for.

These factors make it hard for neural networks to compete against existing, well developed methods of creating a game AI, such as behavior trees or state machines.

REFERENCES

INFORMATION

- Grant Sanderson (3 November 2017), https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- No single writer (Open source website) (August 18, 2018), https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
- Fjodor Van Veen (September 14, 2016), http://www.asimovinstitute.org/neural-network-zoo/
- Prakash Jay (April 20, 2017), https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c
- Mariya Yao (March 22, 2017), https://www.topbots.com/14-design-patterns-improve-convolutional-neural-network-cnn-architecture/
- Jiri Stastny (April 2005), https://www.researchgate.net/publication/292636184_Neural_networks_learning_methods_comparison
- Suryansh S. (March 26, 2018), https://towardsdatascience.com/gas-and-nns-6a41f1e8146d
- Vijini Mallawaarachchi (July 8, 2017), https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3
- Philipp Koehn (December 1994), http://homepages.inf.ed.ac.uk/pkoehn/publications/gann94.pdf

FIGURES

1. Harsh Pokharna (July 26, 2016), https://medium.com/technologymadeeasy/for-dummies-the-introduction-to-neural-networks-we-all-need-c50f6012d5eb
2. Gyan Ranjan (December 20, 2015), https://www.quora.com/What-does-taking-the-derivative-of-a-function-actually-do

3.-6. No single writer (Open source website) (August 18, 2018), https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

7. Fjodor Van Veen (September 14, 2016), http://www.asimovinstitute.org/neural-network-zoo/
8. Andrew Hu (February 4, 2016), https://stackoverflow.com/questions/35192628/use-of-activation-derivative-in-back-propagation
9. Aaqib Saeed (January 4, 2018), https://www.kdnuggets.com/2018/01/genetic-algorithm-optimizing-recurrent-neural-network.html

10.-12. Philipp Koehn (December 1994), http://homepages.inf.ed.ac.uk/pkoehn/publications/gann94.pdf