university of groningen / faculty of science and engineering / artificial intelligence

Applying Hebbian Learning in System Control for Neuromorphic Computing

Bachelor's Project Thesis

Jochem Klaas Kerssies, s3677281, j.k.kerssies@student.rug.nl,
Supervisor: MSc J.J.M.A. Timmermans

Abstract: Last decades, substantial resources have been allocated to developing more capable AI systems, often resulting in larger models with an increased number of parameters. This trend has led to significant energy consumption, raising environmental concerns. Current machine learning models, based on artificial neural networks, rely heavily on energy-intensive training methods such as backpropagation. In contrast, the human brain can operate more efficiently, as it does not use backpropagation. Hebbian learning is a learning method that is biologically plausible and is inspired by the brain. The theory behind Hebbian learning is that the connections between simultaneously active neurons are strengthened. This report explores Hebbian learning as an alternative to backpropagation. In previous research, Hebbian learning has been used often in combination with Hopfield networks. However, there is not a lot of research focused on the use of Hebbian learning for neuromorphic computing. In this report, an adaptation of the standard Hebbian learning rule called Oja's rule is used, which normalizes weight changes. Utilizing this approach, a single-layer perceptron is trained to play multiple adapted versions of the Chrome Dino game, testing the feasibility of Hebbian learning for simple control tasks. The network proved to be capable to some simplified versions of the game, but it did not succeed in learning the more complicated versions. Investigation of this resulted in a mathematical problem that is inherent to the design of the network and the way the state of the game is observed.

1 Introduction

In the past decades, many resources have been dedicated to creating more capable AI systems. These systems have increased in performance, in part, by creating bigger networks with more parameters each year. According to Villalobos et al. (2022), model sizes have increased by around 0.1 order of magnitudes per year between 1952 and 2018. While these systems boast impressive capabilities, they come with a hefty energy cost, contributing significantly to environmental concerns due to their high energy consumption.

The multi billion parameter machine learning models mentioned above are similar to the brain in that they consist of many interconnected neurons. However, training these systems is often done with methods like backpropagation and other gradient based learning methods. These methods are very capable, but are not energy efficient. Compared to these types of learning, the brain functions a lot more on the principle of "neurons wire together if they fire together" (Löwel & Singer, 1992). This was first introduced by Donald Hebb in the book *The Organization of Behavior* (Hebb, 1949). His theory claims that the synaptic connection between two neurons increases if the pre-synaptic neuron repeatedly assists in firing the post-synaptic neuron.

The theory from Hebb is based on the order and grouping between neuron activations. The connections between neurons that often fire together is strengthened over time. A form of learning has been built upon this theory called Hebbian learning. In contrast to backpropagation, Hebbian learning is a biologically plausible and ecologically valid learning mechanism (Munakata & Pfaffly, 2004). Hebbian learning is based on neurons and the brain, and can be used to find correlations in the training data by strengthening the connections between neurons. Hebbian learning is used often in combina-

tion with Hopfield networks (Watson et al., 2009; Centorrino et al., 2022), but it can also be used to train single-layer perceptrons. In this report, a single-layer perceptron is used in combination with Hebbian learning to train the network. The basic Hebbian learning rule can be seen in equation 1.1.

$$\Delta w_{ij} = \eta \cdot a_i a_j \tag{1.1}$$

In this equation, Δw_{ij} defines the change of the weight between nodes i and j. η is the learning rate and a_i and a_j are the activations of the two connected nodes.

Although this equation describes the basics of Hebbian learning, there is a problem with this formula. Weights can increase indefinitely and therefore become infinitely large. To rectify this, I will be using Oja's learning rule in this report (Oja, 1982). Oja's rule is a variation on the basic Hebbian learning rule and is shown in equation 1.2.

$$\Delta w_{ij} = \eta \cdot a_j(a_i - a_j w_{ij}) \tag{1.2}$$

This equation will limit the weights to be in the same range as the activations of a_i and therefore solve the problem of the infinitely large weights.

One of the advantages of Hebbian based learning algorithms, is that all the learning is local in nature. Each weight updates its strength based solely on the activity of the two nodes it connects, without requiring information from other parts of the network. This locality is particularly advantageous for implementation in neuromorphic computing systems. To achieve these neuromorphic computing systems, memristors are needed. Memristors are an electrical component that could simulate a connection between two neurons, as it has a resistance that is changed by the past electrical flow through itself. The creation of nanoscale memristors opens up the possibility of creating these large-scale analog neural networks (Serrano-Gotarredona et al. (2013)). These systems would be biologically plausible and compared to the current multi billion parameter neural networks, have the potential to be a lot more energy efficient.

In this report, I use the Hebbian learning principle to train a single-layer perceptron on an adaptation of the "Chrome dinosaur game" (Google, 2018). This game was chosen as a toy problem for this research as it is a simple game with a small problem space. The goal of this research is to see whether Hebbian learning can be used for system control. If the network with Hebbian learning is capable of learning the game, it would be advantageous to see if it can be scaled up to larger networks with more complicated problems. If Hebbian learning could be implemented successfully on large scale memristor based networks, this could result in systems that are extremely efficient to train and run, as there is no separation between memory and computation. Also, there would be no heavy and costly gradient calculations needed, which will also result in faster training. The code and data used in this report is available and can be found in my Github repository*.

2 Method

2.1 Dino Game

The Chrome Dino game is a well-known endless runner game embedded in Google Chrome. The game consists of a player and different types of obstacles coming towards the player. The goal of the player is to avoid collisions with these obstacles as long as possible, either by jumping over them or by ducking under them. In the Chrome Dino game, the player is represented by a T-Rex. Furthermore, there are 9 types of obstacles, of which 6 are cacti and 3 are pterosaurs. The cacti obstacles are all on the ground and vary slightly in width and height. The pterosaurs, which are flying dinosaurs, are all the same, but can be at three different heights. The pterosaur can be flying close to the ground, in the middle of the game environment or high up in the game environment. The player has three possible actions: a short jump, a long jump and ducking. The jump actions are used to avoid all the cacti obstacles and the pterosaurs at the lowest position. The middle position pterosaur can be avoided either by ducking or jumping. The high flying pterosaur does not have to be avoided, because the "nothing" action results in the player going underneath it. All low obstacles can be avoided by utilizing a short jump, however, the timing becomes extremely critical. Therefore, most human players will use the long jump whenever it it pos-

*The Github repository containing all the code: https://github.com/JochemK1999/Hebbian-Learning-for-System-Control

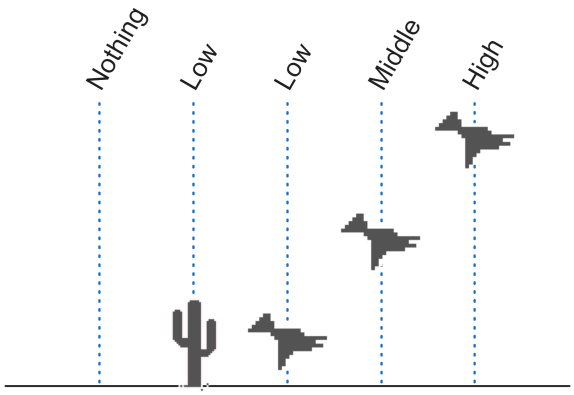

Figure 2.1: Schematic of 5 detectors showing the 4 different signals they can emit. The first detector is detecting nothing, the second and third detector are detecting a low obstacle, the fourth detector is detecting a middle obstacle and the fifth detector is detecting a high obstacle. The emitted signals are shown above the detectors

sible. The only time it is not possible to use the long jump, is when there are two obstacles following each other up in quick succession. This would result in the player landing on the second obstacle, ending the game.

To be able to control the Dino from a neural network, the network needs to be able to know the state of the game to determine the next action. To capture the state of the game, multiple detectors are used. These detectors are placed at certain distances away from the Dino. Each detector can emit 4 different signals, corresponding to if it is detecting an obstacle and at which height. This can be seen in figure 2.1, along with the signals that are emitted by the detectors when they detect an obstacle.

To simplify training and testing I extracted all possible problem states from the game. This way, I could start by training the network on the simplest versions of the game and increase the complexity if the algorithm could successfully learn the previous scenarios.

I started with 6 detectors divided over two clusters (see figure 2.2). The first cluster is close to the player, which would give information about the direct action that needs to be taken. The second cluster is further down the line, these detectors provide information on whether the player could do a long jump or that it would have to do a short jump to

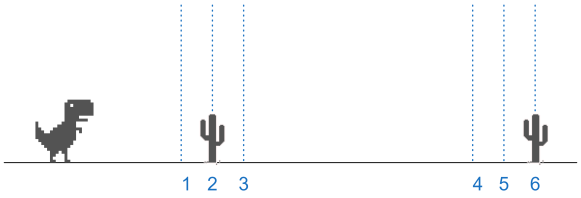

Figure 2.2: Schematic of the Chrome Dino game with 6 detectors. There are two cacti obstacles, one on detector 2 and one on detector 6

ensure landing in time to jump again. There can only be at most one active detector in each cluster as the spacing between detectors is greater than the width of the obstacles. Therefore a single obstacle cannot trigger two detectors simultaneously. On the other side, the distance between two obstacles is always greater than the width of a cluster, so therefore there will never be two obstacles in a cluster at the same time. This way of not watching the entire game but only looking at a couple important places is also something that human players do when judging fast moving objects. A study looking at cricket players has shown that they do not watch the ball the entire time, but instead focus on a couple of key points to predict where the ball will go (Land & McLeod, 2000).

When setting up for the experiment, I created two versions of the game, with different levels of complexity. The first version (see section 2.2) contains only low obstacles, the second version (see section 2.3) contains low and middle height of obstacles.

2.2 Low obstacles game

In the low obstacle game, I removed the pterosaurs, resulting in the game only containing cacti as obstacles. Therefore, the only action the player would need to avoid the obstacles would be the jump actions. Theoretically, the player would not even need the long jump action, as a system with perfect timing could use the short jump to jump over all obstacles just in time. Because there can only be one obstacle in each cluster at a time (see section 2.1), there are 16 possible inputs. There is 1 possible input for when there is nothing detected, 3 possible inputs for when there is a cactus in the first cluster, 3 possible inputs for when there is a cactus in the

second cluster and 9 for when there is a cactus in cluster 1 and a cactus in cluster 2.

For most inputs, the desired action follows logically from the game. However, in the training data I simulated having non-perfect human players by introducing some variability. This was simulated by having data points in the training data that might not cause a collision with a cactus, but would not be the optimal action as it would result in a near-miss. These data points are all the data points when there is a cactus detected at detector 3. A short jump would result in the player jumping over the cactus, but it would be very close. Because this jump does not result in a collision with the cactus, it could be in the training data when gathering data from actual humans. To represent the players variability in the training data, the action for when there is a cactus at detector 3 has a 25% chance of being "short jump" and a 75% chance of being "nothing". Similarly to how the human player might sometimes jump early, the player might also jump late. A late jump would happen when the player jumps when the cactus is at detector 1. If this were to happen however, the action would always be "short jump" as the action continues to be short jump even when the jump event has started. Therefore, if a short jump was initiated at detector 2 or 3, the action would still be "short jump" at detector 1.

For the low obstacle game, I created two different datasets. The first dataset contains only the "short jump" action. The second dataset contains both the "short jump" and "long jump" action. For the second dataset, the "long jump" action was selected when there was a cactus in the first cluster but not in the second cluster. When there is a cactus in both the first and the second cluster, the "short jump" action was selected, regardless of where in the second cluster the cactus was. For all cases though, if the obstacle in the first cluster was at position three, there would still be a 75% chance of the action being "nothing".

2.3 Low and Middle Obstacles game

For the low and middle obstacles game, the pterosaurs at the medium height were kept in the game. The network could avoid these by either jumping or by ducking, but for the purpose of validating if the network could learn to duck as well, "duck" was chosen as the desired action. Similar to the low obstacles game (see section 2.2), there can be at most one obstacle in each cluster at a given time. However, due to the possibility of this obstacle being at a "middle" height, the number of possible inputs increases to 49. The actions for each input again followed logically from the game, with the added premise that the network should predict the "duck" action if there is a pterosaur in the first cluster. The 25% - 75% distribution was kept when there was a "low" obstacle on detector 3.

In the training data for the low and middle obstacle game, all possible inputs occur only once. However, when training data would be gathered by observing a player playing the game, some inputs would have a lot bigger probability of occurring than others. This happens because of the way new obstacles are spawned in the game. To represent this, a new training set was created based upon the training set for the low and middle obstacle game. In this case, each input pattern was given a probability of occurrence in the real game. The game does not create obstacles equally, as there are more cacti spawned then there are pterosaurs spawned. A distribution of 80-20 was chosen to represent this in the training data. This means that for each cluster in which an obstacle is detected, there is an 80% chance of the obstacle being a "low" obstacle and a 20% chance of the obstacle being a "middle" obstacle. The probability of there being an obstacle in a cluster was 50%. The final probabilities for each input can be seen in table 2.1. Finally, the location of the obstacle in the cluster was randomized, with an equal probability for each location. With these values, the probability of an input occurring during the game for each of the 49 possible inputs was calculated. The training set was created by generating 2000 data points based on these probabilities, generating a more realistic training set that could be created by recording the moves of a human player playing the game.

2.4 Learning rule

As mentioned in section 1, The learning rule used in this report will be Oja's rule. Oja's rule ensures that all weights will be limited to be in the same range as the activations of a_i. For the experiments in this report that means that the values of the weights are limited between -1 and 1.

$$\Delta w_{ij} = \eta \cdot a_j(a_i - a_j w_{ij}) \tag{2.1}$$

In this equation, Δw_{ij} describes the change of the weight between node i and node j as mentioned in section 1. η is the learning rate and a describes the activation of the neurons i and j respectively. The term w_{ij} is the current weight between node i and j. In this report, a_i will always be -1 or 1, and a_j will always be 0 or 1. When a_j is zero, no learning for this output will take place. In equation 2.1, this can be seen by the first occurrence of a_j. When a_j is 0, $(a_i - a_j w_{ij})$ will be multiplied by 0, resulting in a Δw_{ij} of 0. This also results in the part between brackets to only be important when a_j is equal to 1. As a_j is multiplied with w_{ij} before the subtraction happens, the equation can be simplified to equation 2.2, as a multiplication with 1 does not change the value of w_{ij}.

$$\Delta w_{ij} = \eta \cdot a_j(a_i - w_{ij}) \tag{2.2}$$

From this equation and the knowledge that a_j will either be 0 or 1, we can deduce that if a_j is equal to 1, Δw_{ij} will become closer to zero the closer the weight, w_{ij}, comes to the activation value of i, a_i.

2.5 Network structure

Hebbian learning trains a neural network by adapting its weights. It updates these weights by using the activation values of the two connected neurons. The network used in this report consists of nodes, which represent the neurons in the human brain. Because of the chosen learning rule for this report (see section 2.4), the network can only learn weights between nodes of which the activations are present in the training data. Therefore the decision was made to use a single layer perceptron.

To calculate the activation of output node a_j, the sum of all input activations a_i are multiplied by their corresponding weights w_{ij}. This can also be seen in equation 2.3

$$a_j = \sum_{n=0}^{i} a_i w_{ij} \tag{2.3}$$

The state of the game is represented by 6 detectors, each of which can be in 1 of 4 states. The 4 states, except for the "nothing" detection state could be argued as being ordinal inputs because they describe the height of the coming obstacle, but it can also be seen as categorical input as they describe 3 different types of obstacles coming towards the player. For this report, the inputs have been used as categorical inputs. Because a single binary input cannot represent these 4 categorical inputs, 3 input nodes were used for each detector. When a low obstacle would be detected, the first of the three input nodes would activate, whilst the other two stayed inactive. With a middle or high obstacle, input nodes 2 or 3 would activate, whilst the rest stayed inactive. When no obstacle is detected, all nodes stayed inactive. With 6 detectors, this results in the network having 6*3 = 18 input nodes. The output of the network consists of 4 nodes which represent the 4 actions a player can take: doing nothing, making a short jump, making a long jump and ducking. A schematic of the network can be seen in figure 2.3.

When an input node activates, its activation value becomes 1. When it is inactive, this activation value could take on an activation value of 0 or -1. There is a big difference between these two options, which can be seen when we look at the learning rules. When an input node takes on an activation value of 0, it will multiply 0 with all weights. This would mean that the summand corresponding to this detector is zero in the forward calculation sum (see equation 2.3). Therefore is not really part of the decision of the network. This means that the

Table 2.1: The probability of each input occurring in the training data. The first and second column show the signal generated by one of the detectors in the cluster, the P_{input} column shows the probability of this input occurring in the game.

Cluster 1	Cluster 2	P_{input}
Nothing	Nothing	0.25
Nothing	Low	0.2
Nothing	Middle	0.05
Low	Nothing	0.2
Low	Low	0.16
Low	Middle	0.04
Middle	Nothing	0.05
Middle	Low	0.04
Middle	Middle	0.01

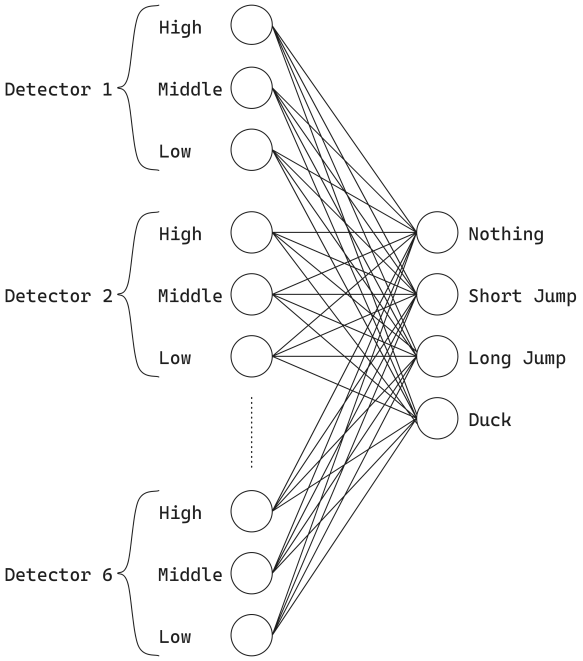

Figure 2.3: The network architecture of the single layer perceptron. On the left are the input nodes. There are 6 groups of 3 nodes, 1 group for each detector and 1 node in each group for each signal. On the right are the 4 output nodes, one for each of the possible actions

network would only be able to make decisions based on inputs that are active. For our game, it would mean that it would only be able to make decisions based on detected obstacles. This however would not work for the Dino game, as the decision between a short and long jump depends on whether or not there is an obstacle in the second cluster of detectors. I wanted the network to be able to act on both there being an obstacle, as well as there not being an obstacle. With the activation value of -1, the network multiplies the weights with -1, meaning the summand would not be constrained to zero and the network can make decisions based on the these inputs. This is also something that can be seen in the human brain. Besides the normal excitatory neurons in the human brain, there are also inhibitory neurons (Swanson & Maffei, 2019). These neurons do the opposite of excitatory neurons and inhibit other neurons from firing, therefore being similar to a negative activation value.

2.5.1 Linear separability

Single layer perceptrons are simple, but have a couple of drawbacks. The most prevalent is that single layer perceptrons are only capable of learning problems that are linearly separable (given linear inputs). This can be circumvented by having combinations of inputs as their own inputs, but having combinations increases the number of input nodes with $\binom{n}{2}$ where n is the number of the original inputs. With these added inputs, the number of weights increases quadratically with respect to the original number of inputs. Therefore it would be preferable that the data is linearly separable. To check if the data is linearly separable, I trained a support vector machine on the validation data for the low obstacle game (see section 2.2). This resulted in an accuracy of 1.0. An accuracy of 1.0 indicates that the data is fully linearly separable. The same was done for the validation set of the low and middle obstacle game (see section 2.3), which also resulted in an accuracy of 1.0.

2.6 Preliminary Testing

Before testing the network on the two game variants described in sections 2.2 and 2.3, multiple preliminary tests where performed. These tests included the same type of detectors as described in section 2.1. In contrast to what is described there, the versions used for preliminary testing used between 10 and 25 detectors, divided evenly across the playing field. The theory behind this was that it was the easiest space representation that also didn't depend on the specific placement of the detectors. Training data for these networks was gathered by recording the actual moves of a player whilst playing the game. The last datapoints were always removed from the training data to exclude the player actions that resulted in the collision of the player with an obstacle. The first test was using the standard Hebbian learning rule (see equation 1.1). For the activations, an activation of 1 was chosen when the detector detected an obstacle, and an activation of 0 when there was no obstacle detected. The game was setup in such a way that there were never two obstacles coming in quick succession. This meant that there was always at most one obstacle within the detectors. The network was able to learn this version of the game completely and jumped and

ducked at the right moments in all possible scenarios. For all further testing, the obstacles would be created in quicker succession, resulting in at most two obstacles being present within the detectors. With the standard Hebbian learning rule and the same activations, the network was not able to learn the game. When analyzing the weights of the network, the weights for the "nothing" action when there is a cactus far away, are a lot bigger than the weights for ducking when a pterosaur is close. This happens because the probability of the obstacle being a cactus is a lot higher than being a pterosaur. Therefore when there is a cactus far away, there will be an obstacle close by the player and the probability of the obstacle close to the dino being a cactus is higher. This results in the weights connecting the far away "low" detectors training more than the closeby "middle" detectors. Therefore, when there is only a pterosaur in the detection field, it will duck. However, when there is a pterosaur close by but a cactus far away, it will choose the "nothing" action, as the weight for the "nothing" action far outweighs that for ducking. To prevent these large weights, the switch was made to Oja's learning rule as Oja's learning rule constrains the weights in the same range as the activation values. This changed the results, but the network was still not able to play the game. In the scenario where it had to duck under a pterosaur, but there was a cactus far away, the network decided that the best action was jump. This resulted in the network jumping into the pterosaur and causing a collision. As this network was trained on a training set generated by recording the moves of a human player, I suspect this has to do with the probabilities in the training data. Because in the training data, when there is a cactus at the end of the game, the probability of there being a cactus close to the player is higher than the probability of there being a pterosaur. This correlation is stronger than the correlation with the pterosaur that is actually at the front, resulting in the dino jumping. Finally, the switch to a negative activation of -1 when there was nothing detected was made. This was because of the realization that when the activation is 0, these detectors are of no influence to the descision making process. I wanted the network to be able to make predictions on obstacles that were presenent, but also on the information that there were no detectors present. However, this ultimatly also didn't result in the network being able to play the game.

3 Results

3.1 Low obstacles game

The network was trained on the two datasets associated with the low obstacles game, as described in section 2.2. To test the performance of the network on these datasets, a validation set was created for each dataset. The validation set contains all possible inputs to the network, alongside their desired action. This is largely the same as the training data, with the exception of all inputs where the desired output was based on probability. For these data points, the output with the highest probability in the training data was chosen.

For the data set without the "long jump" action, the network was trained with a learning rate of 0.01 over 10 epochs. The network was used to predict the desired action depending on the inputs in the validation set and this was compared with the desired actions in the validation set. The accuracy of the predictions of the network on the validation set were recorded after each epoch and can be seen in figure 3.1. In this figure, you can see that the network starts with an accuracy of 0.5, which is expected as the "nothing" action is expected for half of the inputs in the validation set. The network's accuracy increases until after epoch 3, after which it stays the same again until epoch 6. After epoch 7, the accuracy increases to 1. This means the network has learned the entire dataset.

When the network is trained on the data set with the "long jump" action, the accuracy increased and decreased over time (see figure 3.2). The network was trained with a learning rate of 0.01, but over 40 epochs this time to reach a point where all weights had converged. After the first epoch the network's accuracy increases to 0.875. After epoch 23 this increases to 1. However, when the network continues training, the accuracy drops again to 0.625 after 33 epochs.

3.2 Low and medium obstacles game

For the low and medium obstacles game, the network was trained first on the dataset in which each input occurred equally often. A new validation set

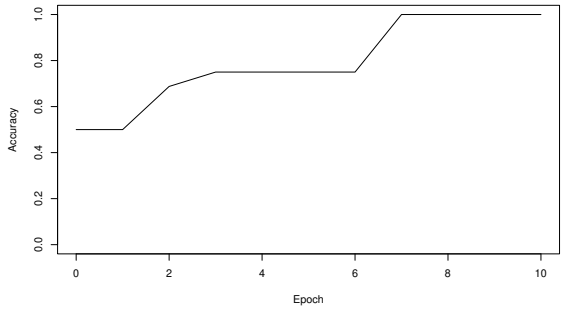

Figure 3.1: The accuracy of the network predictions on the validation set for the low obstacles game without the "long jump" action. The accuracy of the network was measured after each epoch. (learning rate = 0.01)

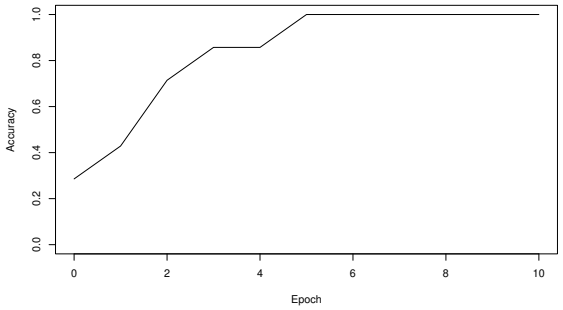

Figure 3.3: The accuracy of the network predictions on the validation set for the low and middle obstacles game with the balanced dataset. The accuracy of the network was measured after each epoch. (learning rate = 0.01)

was created for this dataset, in the same way as for the low obstacles game (see section 3.1).The network was trained with a learning rate of 0.01 over 10 epochs. The results can be seen in figure 3.3. The figure shows the accuracy of the model rising after epoch 1 until epoch 5, after which it reaches an accuracy of 1.

The network was also trained on the probability based dataset, as described in section 2.3. As this only changes the occurrence of all inputs in the dataset, the same validation set as the one for the low and medium obstacles game with the balanced dataset can be used. With this dataset, the network was trained with a learning rate of 0.001 over 10 epochs. The lower learning rate was necessary due to the change in the size of the dataset. Training with a learning rate of 0.01 results in unstable learning. With this data set, the network performs poorly and only manages to reach an accuracy of 0.57 (see figure 3.4).

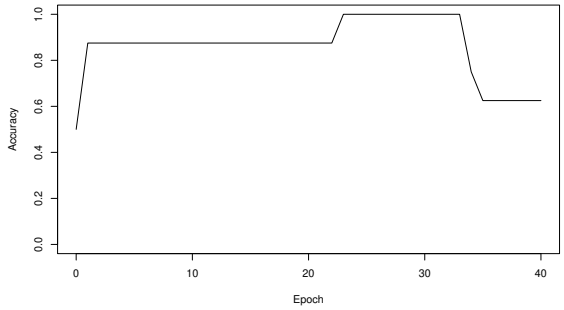

Figure 3.2: The accuracy of the network predictions on the validation set for the low obstacles game with the "long jump" action. The accuracy of the network was measured after each epoch. (learning rate = 0.01)

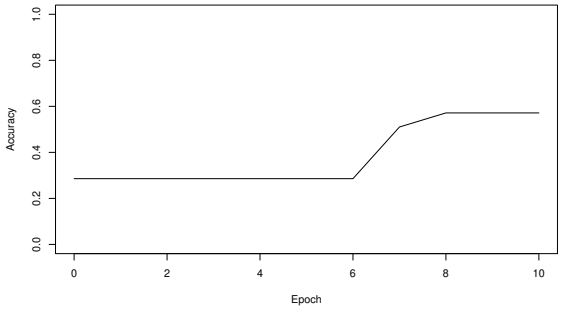

Figure 3.4: The accuracy of the network predictions on the validation set for the low and middle obstacles game with the probability based dataset. The accuracy of the network was measured after each epoch. (learning rate = 0.001)

4 Discussion

4.1 Negative weight values

Something noticeable about the weights of the network after training is that almost all weights have converged to negative values, and the ones that have not converged to negative values are close to zero. When inspecting the weights of the network after training the network on the low obstacle game using the dataset without the "long jump" action, all weight values are negative (see figure 4.1). These negative values can also be seen when looking at the other versions of the game. The graphs for these figures can be found in the appendix. (see figures B.1, B.2 and B.3). The negative values can be explained when looking at the datasets. Due to the structure of the game and the detector clusters, in each cluster there will always be at most one detector detecting an obstacle, whilst the other two are detecting nothing. When there is an pterosaur in cluster 1 and the optimal action would always be "duck", there will always be one detector training the relation of detecting an obstacle with the duck action, whilst the other two detectors will train the relation between the "nothing" signal and the duck action. This results in the weight between the detector and the duck action being negative. This problem can perhaps be solved by changing the way the game is observed, or by changing the network architecture and activation values of the input nodes.

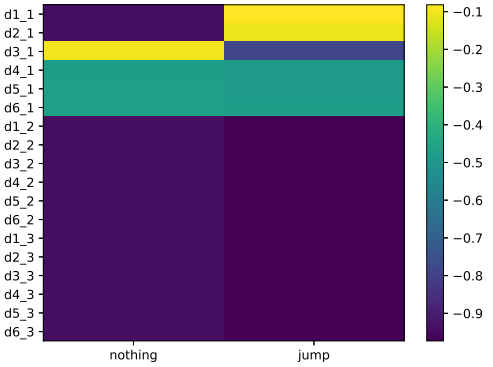

Figure 4.1: Weights of the network after training on the low obstacles game without the "long jump" action for 10 epochs with a learning rate of 0.01. The labels on the y-axis are the names of the detectors in the network. dx_y describes the input for detector x with the signal y. y=1 represents a "low" signal, y=2 represents a "middle" signal and y=3 represents a "high" signal. The labels on the x-axis are the actions the network can take.

4.2 Architecture limitation

The current network architecture and input design have a mathematical problem. In table 4.1, a selection of the weights of the network for the low and medium obstacles game can be seen. All weights of this game can be found in the appendix, see figure B.2. The weights in table 4.1 are the weights connecting the detectors of the first clusters to the actions "nothing" and "duck". If all other weights would be equal, these weights would make the decision on weather the right action would be "duck" or "nothing". When looking solely at the row of detector 1 in table 4.1, we can see that when the inputs are multiplied by the weights, the output for the duck action would be -0.3, whilst the output for the "nothing" action would be -1. This indicates that the network would predict that the optimal action would be duck. However, the other two detectors have the same weight values and their inputs are inverted compared to those of detector 1. This results in the output also being inverted. The outputs from detector 2 cancels out the result of the output of the first detector. This leaves the final output of the cluster to be determined by the third detector. As the third detector is also detecting nothing, the output would be the "nothing" action. This is not the desired output, as there is a pterosaur at the first detector.

The problem described above is a limitation of the network. Because of the lack of hidden layers, the network cannot learn to combine the outputs of the detectors within the clusters. The problem could probably be solved by combining the outputs of the detectors within each cluster together before passing it to the network. However, this would mean the clusters would have to be predetermined, making the network more specific towards this certain game.

Table 4.1: An example of an input calculated through part of the network. d1_2, d2_2 and d3_2 are the inputs of the network corresponding to "Middle" signal of the three detectors in the first cluster. The weights are based upon, but not the exact values of the trained weights from the low and medium obstacle game with the balanced dataset. The output is the result of the input multiplied by the weights of the network. The sum is the sum of the output values and the output with the highest value it the predicted action. In this table, the input corresponds to detecting a pterosaur at the first detector.

		Weights		Output	
	Input	Nothing	Duck	Nothing	Duck
d1_2	1	-1	-0.3	-1	-0.3
d2_2	-1	-1	-0.3	1	0.3
d3_2	-1	-1	-0.3	1	0.3
			Sum	1	0.3

4.3 Successful learning of the "short jump" action

In the low obstacle game without long jump, we can analyze the weights connected to the inputs of the "low" signal from detectors 1, 2 and 3. When the cactus is at detector 1 or 2, the action in the dataset will always be jump. However, due to only one of the two detectors having the positive value of 1, the negative value (-1) of the other detector will decrease the weight between itself and the jump output. Because both scenarios occur equally often in the dataset, these two cancel each other out, resulting in the weights between the detector and the jump output being around 0. When looking back at figure 4.1, we can see that the value of these two weights is indeed close to 0. The slight negative offset can be explained by the fact that when the cactus is at detector 3, the action is jump 25% of the time. At that moment, both detector 1 and 2 are negative, resulting in the weight between these detectors and the jump action lowering.

In the low obstacle game without the long jump action, the mathematical problem described in section 4.2 does not seem to be happening, as the network is able to fully learn the training data. This is because, in contrast to the ducking action, the jump action is not the desired action when a low obstacle is detected at detector 3. As described in section 2.2, this wouldn't result in a collision but would be a near miss and therefore "nothing" is the desired action in this case. As described in 4.2, the detectors 1 and 2 are still canceling each other. But because the weight between detector 3 and the "nothing" action is greater, the "nothing" action has a lower value. The lower value is enough to tip the scales in favor for the jump action, resulting in the network predicting the right action.

4.4 Class imbalance in the training data

When looking at the results of the low objects game with the long jump dataset, we can see that the network starts to classify everything correctly, but after more training the score decreases. The reason for this can explained by figure 4.2. The weight values connecting all detectors, with the exception of the weights connecting the "low" signals from detector 1, 2 and 3 with the long jump action, are all around -0.8. These weights haven't fully reached their steady state yet and will continue to decrease with more training, until they reach close to -1, just like all the other states that only train negative relations. The reason the weights connected to long jump take longer to converge to a stable value is because they are less represented in the dataset. There are only two or three (depending on the 25% chance) actions with long jump in the dataset, in comparison to the 6 to 9 short jump and 4 to 8 "nothing" actions. When the network is fully trained, it will always perform the action "long jump" when there is a cactus at detector 1 or detector 2. This is because the main difference in weights between between the "jump" and "long jump" action are in the weights connecting detector 4, 5 and 6 to these two actions. Here, the problem described in section 4.2 arises.

4.5 Balanced dataset and the mathematical problem

On the low and middle obstacle game with the balanced dataset, the network was able to learn when to duck at the right time was because the desired action when there was a cactus at the third detector was "nothing". With the balanced dataset, this resulted in the weight between the "low" signal of detector 3 and the "nothing" action being around 0. Because the weight between the "low"

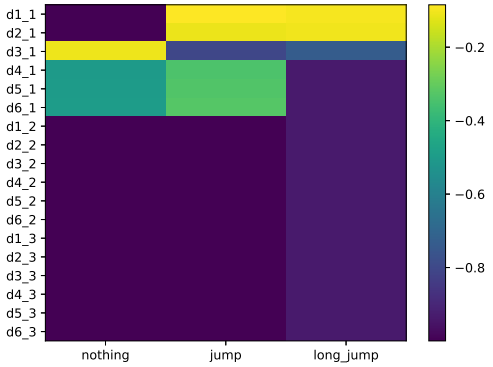

Figure 4.2: Weights of the network after training on the low obstacles game with the "long jump" action for 30 epochs with a learning rate of 0.01. The labels on the y-axis are the names of the detectors in the network. dx_y describes the input for detector x with the signal y. y=1 represents a "low" signal, y=2 represents a "middle" signal and y=3 represents a "high" signal. The labels on the x-axis are the actions the network can take.

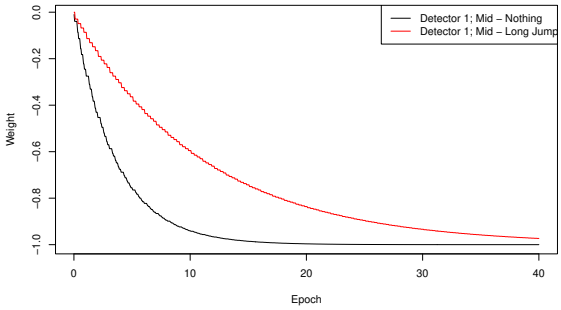

Figure 4.3: The values of two weights of the network after training on the low obstacles game with the "long jump" action for 40 epochs with a learning rate of 0.01. On the x-axis is the progress of training in epochs, the value of the weights was recorded after each training sample in the dataset.

signal of detector 3 and the "duck" action is equal to -1, this can be seen as giving the "duck" action a bias. This bias is enough to offset the problem shown in table 4.1, as the difference between the two is 0.7. For the network that was trained on the dataset with the probability based dataset, the weight value connecting the "low" signal of detector 3 and the "nothing" action is only -0.6. Therefore, this was not enough to offset the 0.7 from table 4.1, resulting in the network failing to classify the "duck" actions correctly.

4.6 Data distribution

In section 4.4, it is described that the weights of the "long jump" action take longer to reach their stable value compared to the weights of the other actions. When looking at figure 4.3, we can see the value of both weights are approaching -1, but because of the imbalance in the training data, the connection between the "middle" detector 1 and the "nothing" action trains a lot quicker than the connection between the "middle" detector 1 and the "long jump" action. With the current setup this is not a big problem as it can just be solved by training for a bit longer. This is however not very efficient and would cause problems for bigger networks. It could be solved however by oversampling the underrepresented classes. This would ensure that all classes are trained equally quickly, resulting in training with fewer epochs total.

4.7 Game environment

As discussed in section 4.2, the current layout with the detectors does not work because the network is not capable of grouping the inputs of the detectors together. This could be solved by a different design of the game environment. The most simple way would be to have 1 detector in each cluster. In this case however, the timing of the jump is not decided by the network anymore, but by the placement of the detector. Another way to solve the problem would be to link the detectors in some way. For example, if one detector in the network detects something, it's value becomes one, but the value of the other detectors becomes 0 instead of -1. Only when nothing is detected within a cluster, the values could become -1. This would solve the problem, but again, requires coupling of the detectors to a certain cluster by hand.

4.8 Application in Electronics

The promise of the learning rule used (see equation 2.2) for electronics is the local learning aspect of it. The training of a weight connecting node i and j requires only it's own weight value and the activations of i and j. When an analog network would be created with memristors as it weights, a small circuit around each memristor could potentially be used to train the network. Communication with all other parts of the network would not be needed, which could really simplify the circuit. The realisation of neural networks in analog electronics would mean incredibly efficient calculations at almost instantaneous speeds. All calculations wouldn't have to be performed sequentially by a central processing unit but would instead occur in parallel due to the physical properties of the electrical components involved.

4.9 Future research

The results discussed in section 3 leave some performance to be desired. To try and combat this, more research is needed on the network architecture and the input design. A future research could look into improving the Hebbian learning algorithm used in this report. This research could focus on finding ways to overcome the network architecture limitations described in section 4.2. This could be in the form of an adaptation to the network architecture or by looking at a different state representation of the game. Another approach could be to use time information. The current neural network only uses the current state of the game to make a prediction. A future research could try using different methods that can train a network in the time domain. This could be done by using a network capable of using spiking signals. A promising example of this is the use of spike time dependent training methods such as STDP (Bi & Poo, 1998). This method has shown great potential and is also considered biologically plausible. Finally, there has already been research done on implementing Oja's rule in memristors, such as the report by Li et al. (2022). However, more research is still needed in this space to create large scale neuromorphic computing hardware.

5 Conclusions

In summary, a single layer perceptron was trained using a Hebbian learning rule called Oja's rule. The network was capable of solving the problems for the low obstacle game without the long jump action. However, the network was not able to successfully train the low obstacle game with the long jump action due to a limitation of the current network architecture and the game state observation. The network did temporarily learn this version of the game, because the training of the different actions happened at different speeds due to a class imbalance in the training data. This resulted in the weights connected to the underrepresented classes converging slower to their final value. The network was also capable of learning the low and middle obstacle version of the game with the balanced dataset, but on the probability based dataset it did not succeed to correctly classify the states in which the correct action would be to jump, again due to the limitations of the network architecture. In general the network and learning algorithm showed promise, but more research is needed before it can be applied to more complex problems.

References

Bi, G.-q., & Poo, M.-m. (1998). Synaptic modifications in cultured hippocampal neurons: Dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of Neuroscience*, *18*(24), 10464–10472. doi: 10.1523/JNEUROSCI.18-24-10464.1998

Centorrino, V., Bullo, F., & Russo, G. (2022). Contraction analysis of hopfield neural networks with hebbian learning. In *2022 ieee 61st conference on decision and control (cdc)* (p. 622-627). doi: 10.1109/CDC51059.2022.9993009

Google. (2018). *As the chrome dino runs, we caught up with the googlers who built it.* `https://blog.google/products/chrome/chrome-dino/`. ([Accessed 07-07-2024])

Hebb, D. O. (1949). *The organization of behavior: A neuropsychological theory.* New York: Wiley. Hardcover.

Land, M. F., & McLeod, P. (2000). From eye movements to actions: how batsmen hit the ball. *Nature Neuroscience*, *3*, 1340-1345.

Li, M., Hong, Q., & Wang, X. (2022, 01). Memristor-based circuit implementation of competitive neural network based on online unsupervised hebbian learning rule for pattern recognition. *Neural Computing and Applications*, *34*. doi: 10.1007/s00521-021-06361-4

Löwel, S., & Singer, W. (1992). Selection of intrinsic horizontal connections in the visual cortex by correlated neuronal activity. *Science*, *255*(5041), 209-212. doi: 10.1126/science.1372754

Munakata, Y., & Pfaffly, J. (2004). Hebbian learning and development. *Developmental Science*, *7*(2), 141-148. doi: https://doi.org/10.1111/j.1467-7687.2004.00331.x

Oja, E. (1982). Simplified neuron model as a principal component analyzer. *Journal of Mathematical Biology*, *15*, 267-273.

Serrano-Gotarredona, T., Masquelier, T., Prodromakis, T., Indiveri, G., & Linares-Barranco, B. (2013). Stdp and stdp variations with memristors for spiking neuromorphic learning systems. *Frontiers in Neuroscience*, *7*. doi: 10.3389/fnins.2013.00002

Swanson, O. K., & Maffei, A. (2019). From hiring to firing: Activation of inhibitory neurons and their recruitment in behavior. *Frontiers in Molecular Neuroscience*, *12*. doi: 10.3389/fnmol.2019.00168

Villalobos, P., Sevilla, J., Besiroglu, T., Heim, L., Ho, A., & Hobbhahn, M. (2022). *Machine learning model sizes and the parameter gap.*

Watson, R. A., Buckley, C. L., & Mills, R. (2009, June). *The effect of hebbian learning on optimisation in hopfield networks* (Project Report).

A Dataset table

Table A.1: Number of student passes and fails per year.

	1	2	3	4	5	6	Action
No cacti	-1	-1	-1	-1	-1	-1	Nothing
Cactus on 4	-1	-1	-1	1	-1	-1	Nothing
Cactus on 5	-1	-1	-1	-1	1	-1	Nothing
Cactus on 6	-1	-1	-1	-1	-1	1	Nothing
Cactus on 1	1	-1	-1	-1	-1	-1	Jump
Cactus on 2	-1	1	-1	-1	-1	-1	Jump
Cactus on 3	-1	-1	1	-1	-1	-1	25% Jump, 75%Nothing
Cactus on 1 and cactus on 4	1	-1	-1	1	-1	-1	Jump
Cactus on 1 and cactus on 5	1	-1	-1	-1	1	-1	Jump
Cactus on 1 and cactus on 6	1	-1	-1	-1	-1	1	Jump
Cactus on 2 and cactus on 4	-1	1	-1	1	-1	-1	Jump
Cactus on 2 and cactus on 5	-1	1	-1	-1	1	-1	Jump
Cactus on 2 and cactus on 6	-1	1	-1	-1	-1	1	Jump
Cactus on 3 and cactus on 4	-1	-1	1	1	-1	-1	25% Jump, 75%Nothing
Cactus on 3 and cactus on 5	-1	-1	1	-1	1	-1	25% Jump, 75%Nothing
Cactus on 3 and cactus on 6	-1	-1	1	-1	-1	1	25% Jump, 75%Nothing

B Weights after training

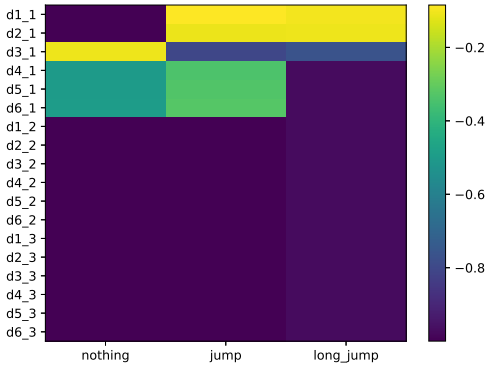

Figure B.1: Weights of the network after training on the low obstacles game with the "long jump" action for 40 epochs with a learning rate of 0.01. The labels on the y-axis are the names of the detectors in the network. dx_y describes the input for detector x with the signal y. y=1 represents a "low" signal, y=2 represents a "middle" signal and y=3 represents a "high" signal. The labels on the x-axis are the actions the network can take.

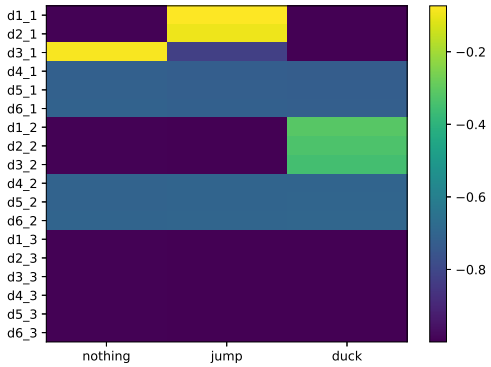

Figure B.2: Weights of the network after training on the low and middle obstacles game with the balanced dataset for 10 epochs with a learning rate of 0.01. The labels on the y-axis are the names of the detectors in the network. dx_y describes the input for detector x with the signal y. y=1 represents a "low" signal, y=2 represents a "middle" signal and y=3 represents a "high" signal. The labels on the x-axis are the actions the network can take.

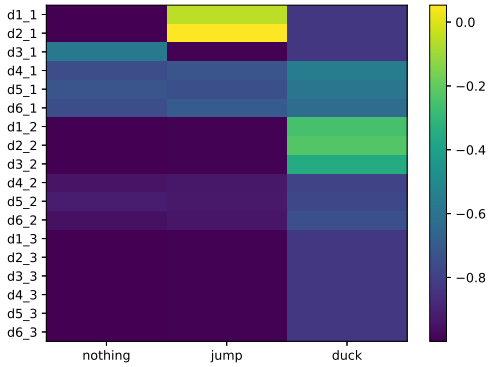

Figure B.3: Weights of the network after training on the low and middle obstacles game with the probability based dataset for 10 epochs with a learning rate of 0.001. The labels on the y-axis are the names of the detectors in the network. dx_y describes the input for detector x with the signal y. y=1 represents a "low" signal, y=2 represents a "middle" signal and y=3 represents a "high" signal. The labels on the x-axis are the actions the network can take.