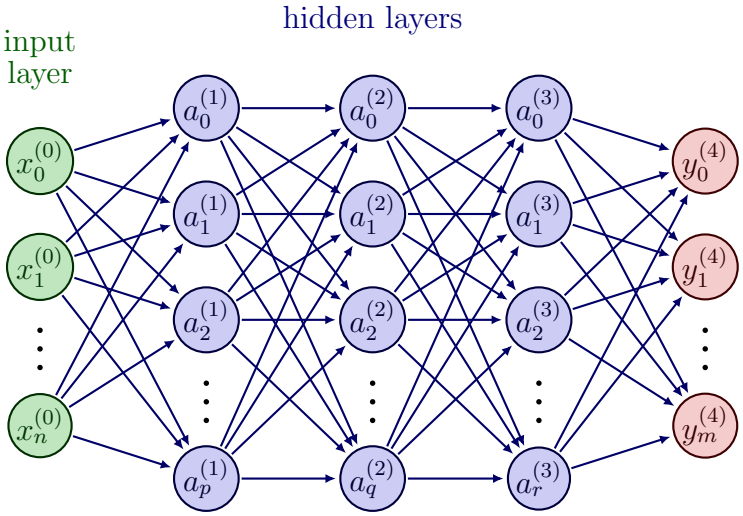
Exploring Neural Networks in C++

First Edition

Toby Davis

October 9, 2023



Abstract

With so many powerful machine learning libraries available for free, almost anyone can develop an advanced AI application in a few hours. Unfortunately, this often comes with a lack of true understanding of the inner workings of the neural network being developed. In this document, we explore what neural networks are, how they work and how they are trained to become experts at almost any task. The explanations are supported with C++ code written using LibRapid for mathematics and linear algebra, and Surge for visualising the results. Having read this document, I hope anyone interested in the field of artificial intelligence will have a deeper understanding of neural networks beyond the usage of black-box machine-learning libraries that are so common today.

Summary

1 A Brief Introduction to Neural Networks

Artificial Intelligence (AI) refers to any computer program capable of performing seemingly "intelligent" actions given some number of inputs. Often, however, when one sees AI mentioned in the news or popular media, the term is generally used to refer to a specific subset of AI called Machine Learning or, more specifically, Deep Learning. These concepts relate to *artificial neural networks*, which are the focus of this document.

Artificial neural networks are the foundation of almost all machine learning techniques. These are collections of *nodes* and *synapses* arranged in *layers*, designed in such a way that information can flow through the network of nodes, being manipulated at each layer and eventually forming a useful response at the output layer.

The power of neural networks, however, arises from their ability to learn incredibly complex behaviours and recognise almost imperceptible patterns in data. By carefully altering how the layers and nodes manipulate information, it is possible to train a sufficiently large network to perform practically any task. Neural networks can be used to solve problems that would be almost impossible with traditional programming techniques due to the complexity of the relationships that need to be modelled.

AI is now incorporated into almost everything we do, from research and medicine to shopping and entertainment. Complex algorithms analyse your search patterns and show more relevant information closer to the top of your search results. At the same time, neural networks pick apart DNA sequences, allowing scientists to find new cures to prevalent diseases and design new vaccines that may save humanity as we know it.

1.1 Key Terminology and Concepts

The field of machine learning has a myriad of complex terminology that can seem confusing at first. This section contains some short descriptions and explanations of common terms you may come across while researching or implementing an AI program. Many of these terms will be expanded in more detail later in the document, so check the glossary for more information.

1.1.1 Supervised Learning

Supervised learning is a method of training neural networks which involves giving the neural network an input with a known output and comparing the model's answer with the "correct" answer. With some complex mathematics (which will be covered later), it is possible to slightly change the weights and biases of the neural network in such a way that, given the same input, it will output something closer to the desired answer. When repeated enough times, the neural network can learn arbitrarily complex behaviours relatively quickly.

Supervised learning has many benefits over alternative techniques, with the most significant being the rate of convergence for the network. Because the neural network can be trained with known inputs and outputs and is altered directly to perform better on the provided data, networks trained in this manner tend to give better results more quickly. Supervised learning may be used to recognise number plates from traffic cameras, for example, since there is an objectively correct answer for the neural network to output.

Unfortunately, supervised learning can only be applied in very specific use cases where there is a "correct" answer for any given input. For example, it is incredibly difficult to use supervised learning when training an AI program to play a video game since hundreds of moves could be equally good at any point in time.

1.1.2 Unsupervised Learning

Unsupervised learning is another technique for training neural networks, relying on data without labelled data. Most unsupervised learning techniques involve many hundreds or thousands of agents all performing the same task. The agents that perform the best are given a higher score than those that perform poorly, and a new generation of agents is produced from the more talented ones in the previous generation. Repeating this process over many generations progressively refines the agents, resulting in an AI model capable of performing the desired task extremely well.

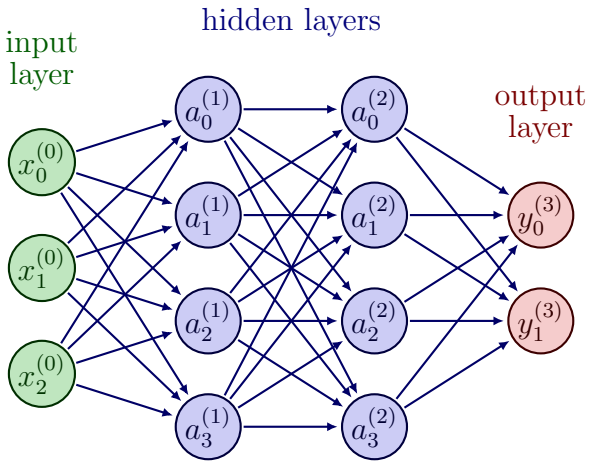
Unsupervised learning is not without its downsides, though. For this approach to work effectively, the programmer must be able to describe the agents' goals incredibly well. While the objective of a game may seem obvious to humans, artificial models often find loopholes in the reward function and find a way of "cheating" the game, increasing their score rapidly but not achieving the game's objective. Furthermore, the quality of the reward function can have a huge impact on the rate at which the agents learn; giving a particular action a score of +3 rather than +2 may completely change how they approach the task.

Training artificial neural networks with unsupervised approaches often takes longer than with supervised approaches since the agents can only improve by small, random changes to their weights and biases. Increasing the number of agents used in the simulation increases the chances they will improve more quickly, but also requires more compute power.

1.1.3 Neural Network Diagrams

Neural networks often need to be represented visually for the purpose of explanation or demonstration, and they're almost always drawn with circles representing nodes and arrows between the nodes representing weighted connections. Many of these terms will be expanded on later, but having a diagram to reference is helpful.

The image below shows a neural network with three inputs, two hidden layers with four nodes each and an output layer with two nodes. The layers are also fully connected.



1.1.4 Input Layer

The input layer of a neural network is its very first layer. This is where data is given to the network to process it. Since neural networks operate entirely with numbers, the data provided to the network must be processed into a form the computer can work with. Often, this involves mapping large numbers into the interval $[-1, 1]$ or looking up values in a dictionary which maps information to numeric values. The original data could be the readings from sensors, the pixels in an image or text from a search query, depending on the task the neural network is intended to perform.

1.1.5 Hidden Layers

Hidden layers in a neural network refer to the layers between the input and output of the network, which the programmer does not directly interact with. They often perform the vast majority of the "thinking" performed by the network, and a substantial amount of time is often spent optimising the number of hidden layers and nodes in each layer.

Generally speaking, increasing the number of hidden layers or nodes in each layer increases the

neural network's capacity to learn. However, it increases training times and can also lead to overfitting.

1.1.6 Output Layer

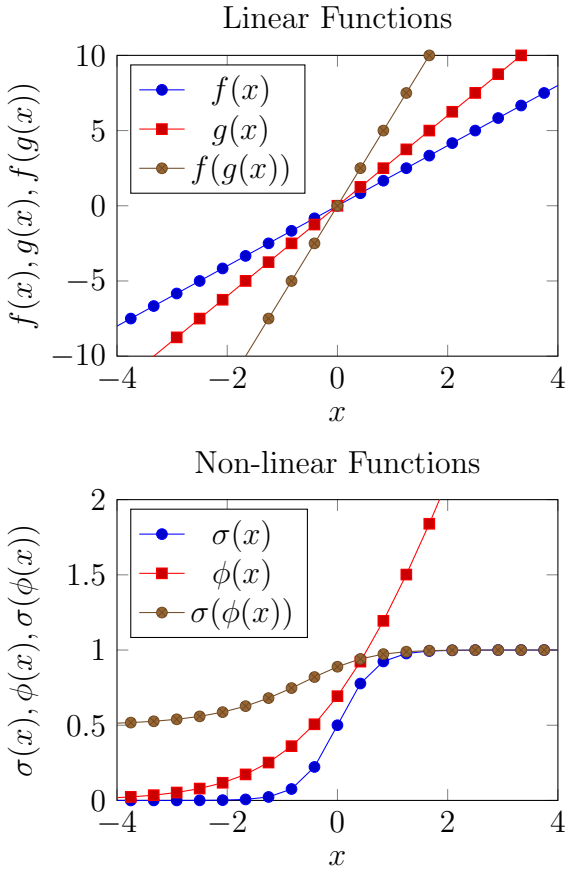
The output layer of a neural network produces the final result, having fully processed a set of input values. Since the output values are also numeric, they generally have to be converted back into something more useful for the main program. For example, if the first output value represents the angle of a motor in a robot, a value from $[-1, 1]$ might be mapped to $[0, 2\pi]$. When a Boolean output value is needed, programmers often set a threshold value over which the output is considered "on" (the Flappy Bird program implemented later in this document uses a threshold of $\tau = 0.5$ to determine whether the bird should jump).

1.1.7 Weights and Biases

The connections between nodes in a neural network are *weighted*, meaning the value passed along is multiplied by some value (the specifics of this are covered in more depth later). The scaling factor used on a particular connection is the weight, w, of that connection. The bias value, b, is another scalar value added to the node after the multiplication stage. Bias values shift the activation function left or right, allowing the neural network to learn more complex behaviours and patterns.

1.1.8 Activation Function

An activation function is a *non-linear* function applied to the value on each node before passing it onto the next layer. The role of the activation function is covered in a lot more depth later on in the document, but, in short, if the activation function were linearly dependent on the input (i.e. $\sigma(x) = kx : k \in \mathbb{R}$), the neural network would essentially be one large, linear function. This is because a linear function of another linear function is, itself, a linear function.



Imagine a neural network with two inputs, x_0 and x_1, and one output, y_0, which is designed to classify one cluster of points from another, where the position of the points depends on some pair of values a and b, where $a, b \in \mathbb{R}$. Imagine, as well, that the output of the neural network is a value in the interval $[0, 1]$, and any output $y_0 \leq 0.5$ means the network believes the point to be in *Group A*, and any $y_0 > 0.5$ in *Group B*. Given a plane where one axis represents a and the other represents b, one could colour different parts of the plane depending on how the neural network classifies the specified point (a, b). If the network acted as a linear function of x_0 and x_1, the intersection between the two classified regions would be a straight line, which, you can imagine, severely limits the neural network's ability to perform the task well (you can imagine two clearly defined sets of points that a straight line

cannot separate). However, with a non-linear activation function, the network can form far more nuanced outputs and perform the desired task with significantly better accuracy.

1.1.9 Learning Rate

The learning of a neural network, also commonly known as the mutation rate, is a hyperparameter which affects the magnitude by which the neural network is altered during any given training step. For unsupervised algorithms, the learning rate often defines the probability that any individual weight or bias in the network is altered. For supervised-learning techniques, the learning rate generally acts as a scaling factor during the backpropagation stage to prevent the neural network from jumping over local or global minima.

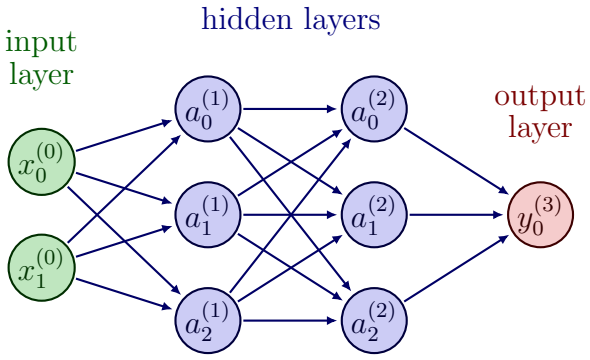
2 Forward Propagation in Neural Networks

2.1 What is Forward Propagation?

Arguably the most important process in deep learning, forward propagation involves passing data through the various layers in a neural network and obtaining an output. Without this, it would be impossible to utilise neural networks to perform the complex tasks we use them for today.

In addition to producing useful outputs, the mathematics behind forward propagation is fundamental to understanding the process of backpropagation, which enables neural networks to be trained on existing data.

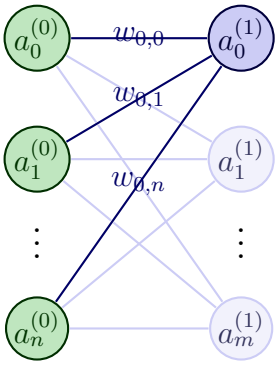
2.2 The Mathematics Behind Forward Propagation



2.2.1 Understanding the Process

Before delving into the mathematics behind the forward pass, it is important to understand what is happening and why it is done that way. With that initial intuition, the equations will make more sense and be easier to process.

The forward feed begins with the input data being passed to the neural network. This data comprises a set of numbers, generally confined to the interval $[-1, 1]$, which come from elsewhere in the program. You can think about these values being "placed" on the nodes in the neural network's input layer.



To compute the value for a single node in the next layer, the values on the input nodes are multiplied by their respective weights on every connection, summed together and then passed through an activation function. This means performing n multiplications and additions[1], where n is the number of nodes in the layer.

This process is repeated m times so that every node in the next layer has a value. The same process can then be applied again, except using the new set of "input" values. The values are multiplied by the connection weights, added together and passed through the activation function. The process is repeated until the output layer is reached, and the programmer can then use the values on the nodes to make decisions or perform actions.

2.2.2 Understanding the Mathematics

Let $\{y_0, y_1, y_2, ..., y_m\}$ be the set of output values from a given layer, $\{\{w_{0,0}, ..., w_{0,m}\}, ..., \{w_{n,0}, ..., w_{n,m}\}\}$ be a set of weights for each node in the selected layer and $\{\beta_0, \beta_1, ..., \beta_m\}$ be the set of bias values for the layer. With σ representing the activation function and $\{a_0, a_1, .., a_n\}$ being the set of input values, the output of a single node is given as follows.

[1] Assuming the layers are fully connected. It is possible to have a pair of layers where not every node is connected to every other.

$$\begin{aligned} y_i &= \sigma\left(a_0 w_{i,0} + a_1 w_{i,1} + \cdots + a_m w_{i,m} + \beta_i\right) \\ &= \sigma\left(\sum_{j=0}^{m} a_j w_{i,j} + \beta_i\right) \end{aligned}$$

Applying this to each node in the layer, one may start to notice a pattern in the calculations being performed. This calculation can be rewritten in matrix form as follows, using the rules of matrix multiplication to simplify some of the calculations.

$$\begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_m \end{pmatrix} = \sigma\left[\begin{pmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,m} \\ w_{1,0} & w_{1,1} & \dots & w_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n,0} & w_{n,1} & \dots & w_{n,m} \end{pmatrix}\begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix} + \begin{pmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_m \end{pmatrix}\right]$$

$$\mathbf{y} = \sigma\left(\mathbf{w} \cdot \mathbf{a} + \boldsymbol{\beta}\right)$$

This final equation is a heavily simplified version of the original ideas behind the forward pass, though it still shows the overall logic behind it. The output of a layer, $\mathbf{y}$ is the result of applying the activation function, σ, to the cumulative sum of the inputs, $\mathbf{a}$ multiplied by the weights, $\mathbf{w}$, and the biases, $\boldsymbol{\beta}$.

2.3 Activation Functions

As mentioned earlier, the activation function is a non-linear function used to find the output of each layer. It ensures the neural network is capable of learning more complex behaviours and is often used to ensure the values in large neural networks do not explode[2].

Some of the most common activation functions are described here, though many more exist with different properties and characteristics.

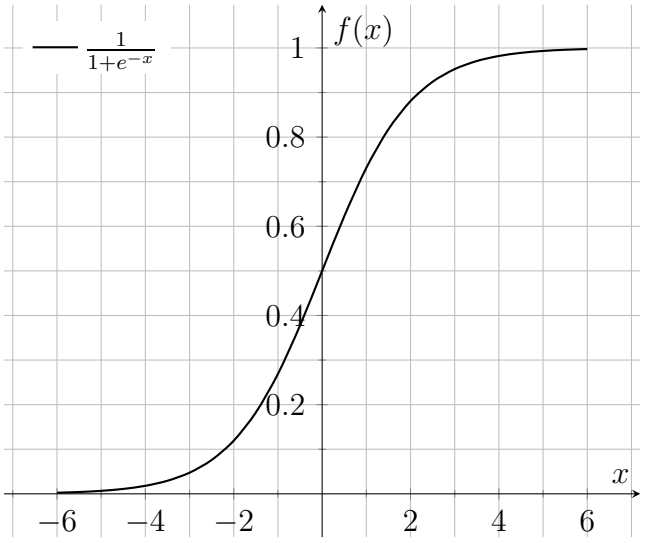
2.3.1 Logistic Curves

Arguably the most well-known class of activation functions, logistic curves, also known as sigmoid curves[3], have the form

$$f(x) = \frac{L}{1 + e^{-k(x - x_0)}}$$

and map values into the interval $[0, L]$. However, the most common equation to use is centred at zero and maps values from 0 to 1. The standard logistic function is often denoted by $\sigma(x)$.

Standard Logistic Activation



2.3.2 Hyperbolic Tangent

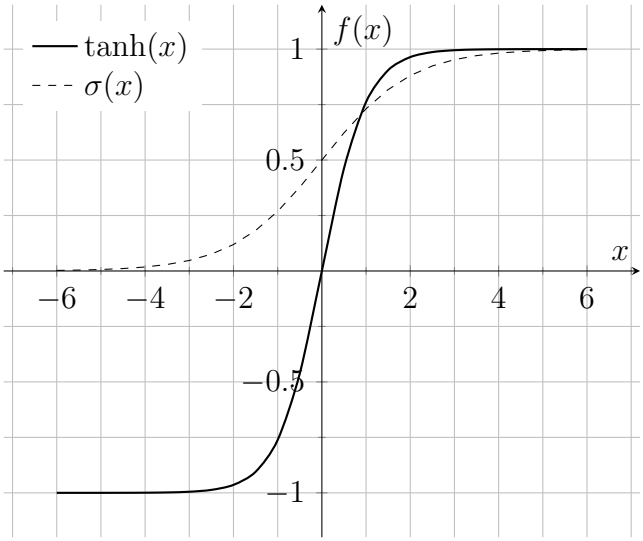
Another sigmoid curve commonly used as an activation function is the hyperbolic tangent operator, since it maps values from the interval $(-\infty, \infty)$ to $(-1, 1)$, following an almost identical "S" shape as the logistic function mentioned previously. In fact, with some algebraic manipulation, it is possible[1] to show that $\tanh(x) = 2\sigma(2x) - 1$.

[2] In large neural networks, the values may become increasingly large when moving from layer to layer, eventually overflowing and simply becoming NaN or ±Inf

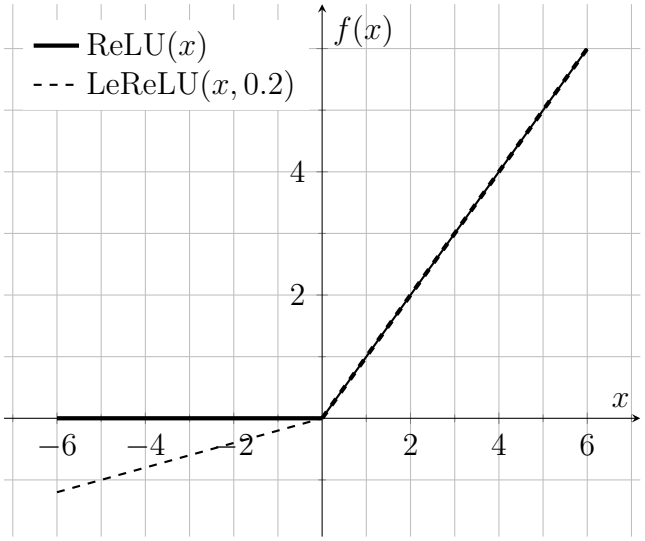
[3] Technically, logistic curves are a subset of sigmoid curves, which are curves with an "S" shape to them. In the context of machine learning, these two names are often used interchangeably.

The graph below shows $f(x) = \tanh(x)$ and $g(x) = \sigma(x)$ to emphasise the similarities between the functions.

Standard tanh Activation



Rectified Linear units



2.3.3 Rectified Linear Units

A rectified linear unit (ReLU) is a piecewise function that outputs zero for negative inputs and the input itself otherwise. ReLUs are often used due to their computational efficiency and their ability to reduce the effects of the vanishing gradient problem within large neural networks.

A variation of the traditional ReLU, known as "Leaky ReLU", has a small, positive coefficient α applied to negative values, such that the gradient is non-zero for negative numbers. The value of α is often set to 0.01.

$$\text{ReLU}(x) = \begin{cases} x & x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\text{LeReLU}(x, \alpha) = \begin{cases} x & x \geq 0 \\ \alpha x & \text{otherwise} \end{cases}$$

3 Genetic Algorithms

3.1 A Brief Introduction

As the name suggests, genetic algorithms aim to model the idea of natural selection and survival of the fittest. A group of agents, each with a neural network 'brain', are allowed to operate within the context of the problem being solved. Each agent can receive inputs and produce outputs, and the agents which perform the best are allowed to reproduce and pass their 'genetics' onto the next generation. Over time, the neural networks 'evolve' and improve at the task, and the process ends when some termination condition is reached.

Genetic algorithms are a type of *unsupervised learning* technique since they do not require a known output for any given input (unlike supervised techniques). This makes genetic algorithms incredibly powerful and enables them to solve a huge range of problems, from finding patterns in large datasets to playing video games.

3.2 Components of Genetic Algorithms

Genetic algorithms have many moving parts that must fit perfectly for the training process to work correctly. If any part is not functioning correctly, it is unlikely that the resulting networks will perform the desired task.

3.2.1 The Environment

The environment of a genetic algorithm is the simulated space in which the agents operate. For example, imagine a game of cat and mouse, where a mouse is trying to get some cheese without being caught by a cat. The environment is where the cat and mouse can move and interact and can contain added features, such as the cheese, doors, traps, etc.

For the agents to perform the desired task, they must be able to interact with sense the environment. For the cat and mouse game, this may involve the mouse knowing its location within the room, the cat's location and any mouse traps within the room.

3.2.2 Agents

An agent is an individual capable of receiving inputs from the environment, making a decision, performing an action and having that action reflected in the environment.

Typically, each agent will have a neural network acting as a brain, which takes inputs and produces outputs via the feed-forward algorithm described earlier in this document. For example, the mouse may receive its x and y coordinates in the room as values mapped to the interval $[-1, 1]$ (with $(0, 0)$ being the centre of the room). The neural network may then output a target coordinate in the same way.

3.2.3 The Population

The population is a collection of agents aiming to perform the same task. The population may consist of many hundreds or thousands of agents, depending on the complexity of the task and the available compute power.

3.3 Processes in a Genetic Algorithm

3.3.1 Random Initialisation of the Agents

The first generation of agents normally starts with a neural network full of randomly initialised values. There are many ways of doing this, and the exact method can depend on the activation function being used, but the simplest is to use a uniform distribution in the range $[-1, 1]$.

The resulting agents are, unsurprisingly, almost entirely useless at performing the desired task. This said, some will, by pure chance, perform better than others. This is the core idea behind the algorithm,

and the small, random improvements that occur within agents between generations eventually lead to a valid solution to the problem.

3.3.2 Processing a Generation

Once the agents have been initialised, each one must be allowed to sense its environment and make a decision, which is then reflected by the agent moving or interacting with something.

This process is repeated until a desired state is reached or no more agents are capable of moving. For example, this may be when all the mice have been caught by the cat or after a certain number of time-steps[4].

3.3.3 Producing the Next Generation

There are a range of approaches for producing the next generation of agents in the population, each with its own advantages and disadvantages. Some of these methods are explained in more detail below, but the overall goal is to produce a new generation of agents which perform better than the previous one.

Normally, the number of agents remains constant throughout the training process, though it is not uncommon for this number to be altered. For example, starting out with many agents enables them to find a coarse solution very quickly but is very computationally intensive. Reducing the number of agents reduces strain on the computer, allowing more generations to be run more quickly, resulting in finer adjustments to the agents.

3.4 Algorithms to Select Parent Agents

3.4.1 Elitism

Since there is a significant amount of randomness involved in the production of the next generation, it is possible that no improvements will be made and that the population will decrease in quality.

To mitigate this issue, the best-performing agents are often copied directly into the next generation without alteration, ensuring that, at the very least, progress is not lost.

Elitism selection must be used in conjunction with other approaches to select the remaining agents to populate the next generation.

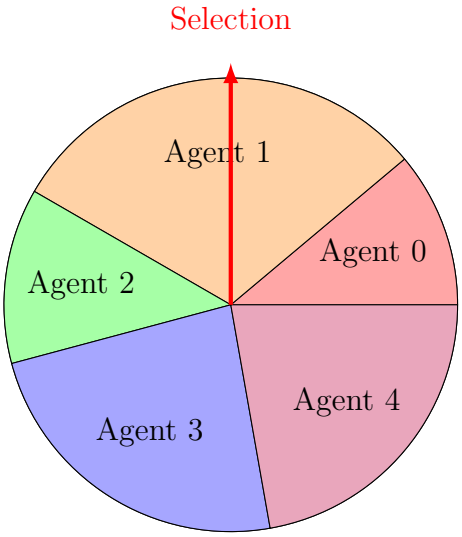
3.4.2 Truncation Selection

This is the simplest way of selecting parents for new agents, where the agents are sorted based on fitness, and only the best n agents are selected as possible parents. Some amount of crossover or repetition will be required to maintain a constant population size, but this ensures that only the strongest agents are able to influence the next generation.

3.4.3 Roulette Wheel Selection

The roulette wheel selection method of picking a parent for an agent involves assigning each agent a segment on the wheel whose size is proportional to the agent's fitness. The wheel is then spun, and whichever agent is selected is the chosen parent.

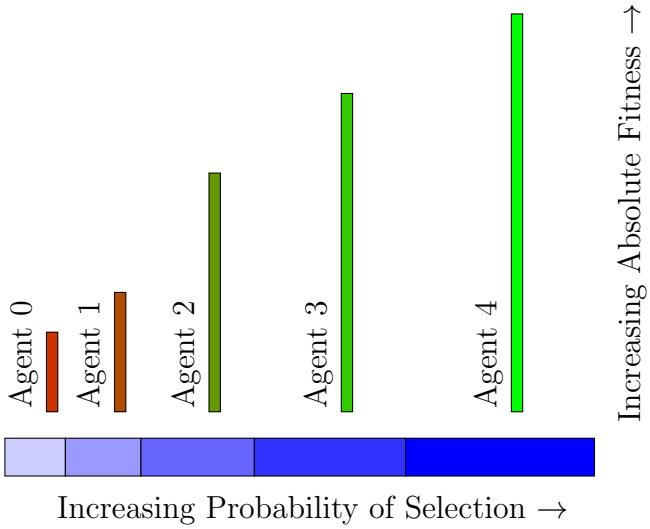
[4]Sometimes it is difficult to encourage an agent to make any useful progress – especially at the start of the training process. By limiting the number of population updates and slowly increasing it over each generation, the agents can be given a 'helping hand' to move in the right direction, dramatically reducing the required training time.



This selection method biases the results in favour of the most capable agents but does not rule out the selection of weaker ones. Counterintuitively, this sometimes produces a better next generation than if only the best-performing agents are selected. It increases diversity between generations and allows more evolutionary paths to be explored.

3.4.4 Rank Selection

Rank selection is similar to roulette wheel selection in that agents are picked randomly, with a bias towards those who perform better. Where rank selection differs, however, is in how the biasing is implemented. With rank selection, the probability of an agent being selected is determined by its rank in the population when ordered by absolute fitness. This differs from roulette wheel selection, which uses only the absolute value.



One benefit of this technique over the roulette wheel approach is that it is not biased in favour of agents with significantly higher or lower fitness scores since the absolute value is abstracted away. This can make it useful in situations with a significant range of fitness scores.

3.5 Producing Children from Parent Agents

Once a parent has been selected, there are a number of different ways to produce a novel child agent which can take part in the next generation. Most methods of producing a child involve copying and then modifying the weights and biases of the parent agent. A *learning rate* hyperparameter represents the probability that any given weight or bias will be modified.

Common methods of altering the weights and biases include *Random Setting*, *Gaussian Pertubation* and *Crossover*, among others.

In *Random Setting*, the learning rate determines the probability that a given weight or bias is altered. It is set to a (uniformly distributed) random value, often in the interval $[-1, 1]$.

Gaussian Perturbation involves adding a normally distributed value to the weight or bias. Since $X \sim N(0, \sigma^2)$ is symmetrical about $x = 0$, it is

equally likely that the value will increase or decrease. This means the weights and biases will generally remain centred around zero. Smaller standard deviations result in smaller changes to the weights and biases, potentially reducing overfitting and sensitivity to input changes.

Crossover merging requires selecting two or more parent agents. The exact method used to merge the 'genes' depends on the implementation used, but the resulting child agent contains genes from each parent. One common method is to pick a random parent to supply the gene for each weight and bias in the agent's brain.

3.6 Fitness Functions

The fitness function is used to determine the 'score' of each agent in the population and is the thing the agents will eventually optimise.

Designing a fitness function for a given problem can be a challenging task. If the programmer fails to *perfecty* describe the target of the agents, they may end up solving a different version of the problem than was originally intended.

For example, refer back to the cat and mouse game detailed earlier. Alice defines a fitness function $f(t) = t$ for $t \in R$, where t is the time before the mouse gets caught. Using this fitness function, the agents will be given a higher score the longer they survive. Intuitively, this will result in agents who expertly avoid the cat but do not target the cheese.

Bob decides that mice who get the cheese should have a higher score than those who do not. Bob defines $g(t, c) = t + c$ for $t \in R$ and $c \in \{0, 1\}$. Here, c is a Boolean equal to 1 if the mouse has gotten the cheese and is equal to 0 otherwise.

Charlie also wants to prioritise mice who get the cheese and defines $h(t, c) = t + 100 \times c$.

Somewhat counterintuitively, Alice and Bob's fitness functions perform almost identically and fail to produce agents which value the cheese. However, Charlie's produces agents who excel at avoiding the cat and getting the cheese. This is because Bob's mice must only survive for 1 time unit more to receive the same benefit as a mouse which gets the cheese. On the other hand, Charlie's mice get a significant bonus for getting the cheese and, hence, are more strongly encouraged to do so.

3.7 Genetic Algorithms in Summary

Genetic algorithms are relatively simple to implement, can work on unlabeled data, often find highly efficient solutions to problems and can be fine-tuned to converge on a solution more quickly.

That said, genetic algorithms require the programmer to describe their goal *perfectly*, leaving no room for ambiguity. They often require many agents in the population before any progress is made, and substantial computing power is needed to simulate them.

4 Implementing a Genetic Algorithm to Play Flappy Bird

4.1 Introduction

Flappy Bird is a well-known single-player game where the player must instruct a bird to flap at the correct time to fly through gaps in pipes. The game ends if the bird hits a pipe, the floor or the ceiling. The objective of the game is to fly as far as possible.

This document guides you through the process of implementing the game, writing a simple neural network framework and integrating the two together so that an AI program can learn to beat the game.

We will implement the program in C++, using LibRapid for numeric calculations and Surge for the graphical interface. This implementation also requires CMake and a modern C++ compiler (C++20 or later).

4.2 Project Setup

Before implementing the program, the project must be set up correctly. This involves setting up the `CMakeLists.txt` file and the necessary folders and files.

4.2.1 Directory Tree

This project requires the directory tree shown below. You can customise this to fit your needs, but this is the directory we will use.

```
FlappyBirdAI/
├─ build/
│  ├─ debug/
│  └─ release/
├─ include/
│  ├─ bird.hpp
│  ├─ brain.hpp
│  ├─ configuration.hpp
│  ├─ generation.hpp
│  ├─ utils.hpp
│  └─ wall.hpp
├─ CMakeLists.txt
├─ main.cpp
└─ surge/ (GitHub Repo)
```

The `build/*` directory will contain the executable binaries for the project. `include` contains the C++header files for our implementation and the configuration settings. The entire project is controlled by the `CMakeLists.txt` file, and `main.cpp` contains our main program loop. The `surge` directory is the GitHub repository containing the code for the graphics and also includes LibRapid for the calculations.

4.2.2 Initial Project Setup

Starting with the directory structure shown above, clone Surge into your project. This can be done by opening a command line in the project directory and running the following command:

```
1  git clone --recursive
       https://github.com/Pencilcaseman/surge.git
```

Next, we can write the CMake file for our project. In `CMakeLists.txt`, write the following:

```
# CMakeLists.txt

# Set the minimum CMake version and
# the project's name
cmake_minimum_required(VERSION 3.16)
project(FlappyBirdAI)

# Set the C++ standard to C++23
set(CMAKE_CXX_STANDARD 23)

# Create the executable and add main.cpp
# as the source file
add_executable(FlappyBirdAI main.cpp)

# Configure LibRapid. See more options at
# https://librapid.rtfd.io/en/
    latest/cmakeIntegration.html
set(LIBRAPID_OPTIMISE_SMALL_ARRAYS ON)
set(LIBRAPID_FAST_MATH ON)

# Add surge as a subdirectory and link it
add_subdirectory(surge)
target_link_libraries(FlappyBirdAI PUBLIC
    surge)
```

Test this works correctly by writing a simple program in `main.cpp`:

```
// main.cpp
#include <surge/surge.hpp>

int main() {
    fmt::print("Hello, World\n");
    return 0;
}
```

Compile the program you just wrote using CMake in the command line (starting from the project root `FlappyBirdAI`).

```
cd build/debug
cmake ../.. -DCMAKE_BUILD_TYPE=Debug
cmake --build . --config Debug

# Windows
./Debug/FlappyBirdAI.exe

# macOS/Linux
./FlappyBirdAI
```

If everything is working correctly, this will compile and run correctly, outputting "Hello, World" to the console.

Glossary

Activation Function A non-linear function which is applied as the final step of computing the output of a node. 6

Agent A common term in unsupervised learning techniques, referring to an instance of a neural network capable of taking actions in its environment. 5

Artificial Intelligence A of programming aimed at getting computers to make "intelligent" decisions. 4

Bias A scalar added to a node before being passed through the activation function, used to shift the function left or right. 6

Convergence Rate A measure of how quickly a machine learning model arrives at a viable solution to the problem. 4

Deep Learning A subset of machine learning that aims to model the way the human brain works. 4

Hidden Layers Layers which the programmers do not directly interact with. 5

Hyperparameter An external parameter set by the programmer which determines certain characteristics about the neural network. 7

Input Layer The first layer in a neural network. 5

Learning Rate [supervised learning techniques] A scaling factor applied during backpropagation to prevent the model from overstepping local or global minima. 7

Learning Rate [unsupervised learning techniques] The probability that any given weight or bias is altered. 7

Machine Learning A of subset of artificial intelligence focusing on artificial neural networks. 4

Node [in a neural network] A fundamental unit of a neural network which receives input from other nodes or an external source, computing an output. 5

Output Layer The layer which produces the result from the neural network. 6

Overfitting When a neural network "learns" its training data and performs exceptionally well on the data it was trained on, but very poorly on real-world data. 6

Supervised Learning A method of training neural networks when inputs have known output values. 4

Synapse A weighted connection between two nodes in a neural network. 5

Unsupervised Learning A method of training neural networks where the exact output for a given input is not necessarily known. 5

Weight A scaling factor applied to values being passed along a synapse. 6

References

[1] Sebastian Raschka. Is the logistic sigmoid function just a rescaled version of the hyberpolic tangent (tanh) function?, Jul 2023. URL https://sebastianraschka.com/faq/docs/tanh-sigmoid-relationship.html.