Extended Essay

Mathematically Modeling a Biological Neural Network

Research Question: To what extent does a mathematical model of a *biological neural network* (BNN) outperform a conventional *feed-forward neural network* (FFNN) in terms of resource utilization and prediction accuracy?

Word Studies: Science, technology, and society

Word Count: 3890

Table of Contents

Introduction 2
Investigation 3
Understanding Biological Neurons 3
Neuron Structure 3
The Synapse 5
Types of Neurons 6
Artificial Neural Networks 8
The Perceptron 9
The Activation Function 9
Learning 9
Synaptic Strength 10
Weights 10
Operant Conditioning 11
Deriving the Model 11
Base Model 11
Corrective Stimulus 15
Testing the Model 16
The Experiment 16
The Results 19
Experimental Analysis 22
Conclusion 23
References 24
Appendix A 26
BANN Source Code 26
FFNN Source Code 29
Input Configuration Source Code 29

Introduction

With the recent development of artificial intelligence (AI) such as ChatGPT, many new technologies have emerged. Artificial intelligence systems are learning to write code, play games, or maintain conversations. These innovations are all based on the concept of the brain's neurons and their corresponding networks.

However, these AIs require a lot of time, processing, and data to operate at their current levels. In fact, just training OpenAI's GPT-3 model used about the same energy required to power 130 homes in the US (Vincent, 2024). Considering the trillions of requests these models process daily, they can consume roughly 1 GWh of electricity per day—enough to power over 30,000 households (Moazeni, 2023).

This essay aims to answer the question, "To what extent does a mathematical model of a *biological neural network* (BNN) outperform a conventional *feed-forward neural network* (FFNN) in terms of resource utilization and prediction accuracy?" To achieve this, a model of a biological neuron will be constructed, the neurons will be chained together to form a network, and the resulting network, dubbed the *biologically approximated neural network* (BANN), will be compared to an FFNN and BNN when learning to play the computer game, *Pong*.

Investigation

Understanding Biological Neurons

A neuron can be thought of as the biological equivalent of a mathematical function. In a BNN, the network is given an input, and multiple “functions” process it to provide a result. Since each behaves differently, they must be modeled accordingly.

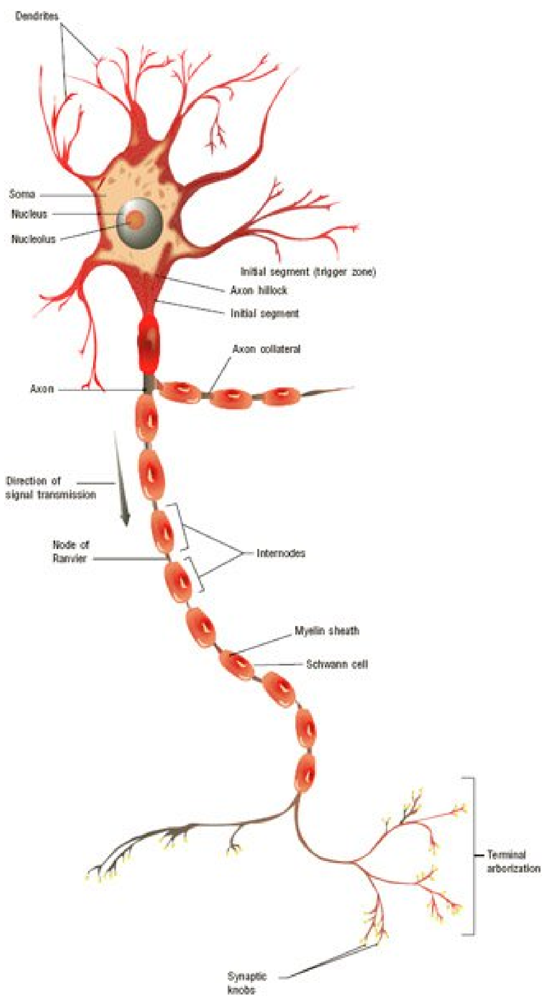
Neuron Structure

According to Hoehn & Sibler (2016), a neuron has four basic parts: the *dendrites*, *cell body*, *axon*, and *synaptic terminals* (see Figure 1) (THE STRUCTURE OF A TYPICAL NEURON, para. 1). The dendrites are the receptors. They pass the input to the cell body, which processes it, and decides whether to fire. If the neuron fires, the signal travels down the axon and to the synaptic terminals, which give stimuli to the dendrites of the following neurons.

This parallels nicely with a mathematical function. A function is given a set of inputs, processes them, and then returns a result. Neurons follow that concept but replicate it on a biological scale, receiving stimuli and firing as a result. However, the crux of the issue is understanding what goes on within the function.

Figure 1

Parts of a Neuron



Note. From *Biology*, by K. Hoen and K. W. Sibler, 2016, Macmillan Reference USA (https://link.gale.com/apps/doc/CX3629800295/SCIC?u=spot51331&sid=bookmark-SCIC&xid=f82cc33e). Copyright 2016 by Macmillan Reference USA, a part of Gale, Cengage Learning.

Since neurons work in a biochemical context, they rely on chemical reactions to process stimuli. In the brain, a neuron is surrounded by positively charged fluids, while its interior is negatively charged, around – 70 mV (Hoehn & Silber, 2016, ELECTRICAL SIGNALS IN NEURONS, para. 2). The cell’s membrane keeps this difference in charge by allowing certain ions to flow to and from the inside of the neuron (Dowling, 2018, p. 25). It uses *channels* and *pumps* that allow certain charged particles to travel across the membrane, thus altering the neuron’s internal voltage.

By regulating the difference in charge, a neuron can send electrical signals to and from each other by briefly building up the electrical tension, and then rapidly discharging it, sending an electrical impulse down to the next neuron. This discharge happens when the neuron reaches a voltage of -50 mV (Hoehn & Silber, 2016, ELECTRICAL SIGNALS IN NEURONS, para. 2).

This is the reason neurons fire. The discharge is an all-or-nothing response to stimuli. Neurons fire when a threshold is reached, and do not fire before then.

This concept can be represented mathematically with a piecewise function that outputs 1 when the threshold is reached and 0 otherwise. Based on this information alone, $N(x)$ represents the neuron.

$$N(x) = \left\{x \le A_p: 0, x > A_p: 1\right\} \quad (1)$$

If A_p represents the threshold needed to reach the action potential, $N(x)$ only returns 1 when the input, x, is above the threshold.

The Synapse

There are two types of synapses, *excitatory*, which instigates more electrical activity in the following neuron, and *inhibitory*, which reduces the electrical activity (Dowling, 2018, p. 32). When an excitatory neuron's synapses fire, it causes the following cell to intake more positive ions. This disturbs the carefully regulated charge imbalance on the cell membrane, increasing the chance of the neuron firing.

Inhibitory synapses are the opposite. They allow negative ions to enter the following neuron, making it more negative and less likely to fire (Dowling, 2018, Chapter 2).

> These synapses make a neuron less likely to fire at an action potential. … Channels at inhibitory synapses typically allow Cl- to pass into the cell; Since Cl- is negatively charged, the cell becomes more negative inside when the inhibitory synapses are activated. (Dowling, 2018, Chapter 2)

This means that there needs to be more positive ions to flow into the membrane to reach the threshold, thus reducing its likelihood for action potential.

As previously stated, a neuron's synapses are either excitatory or inhibitory. Mathematically, this can be shown as a synapse outputting 1 if excitatory or -1 if inhibitory. If *S*, is a matrix of the synapse inputs for neuron *i*, then the following is deduced:

$$S_{ij} = \left[S_{i1}, S_{i2}, \ldots, S_{ij}\right] \qquad (2)$$

$$N\left(\sum_{j=1} S_{(i-1)j}\right) \; i \neq 1$$

The input for any given neuron is the sum of the previous neuron's synaptic outputs. The first set of neurons, however, will just receive the raw input from its environment.

Types of Neurons

There are three primary types of neurons: *sensory neurons* (Swanson, 2012, p. 29), *motor neurons* (Swanson, 2012, p. 32), and *interneurons* ("Types of Neurons", n.d., Interneurons, para. 1). Although each has similar internal behaviors, they differ in how they give and are given stimuli.

Sensory Neurons. Starting with sensory neurons. They are the gateway to the environment. Like Swanson (2012) says:

> It is a *bipolar cell*, with a *detector*, *sensory*, or *input* end directed toward or into the environment and an *effector*, *motor*, or *output* end going to a group of responsive cells like myocytes. … they can be highly localized in various parts of the body, like at the ends of tentacles. This provides a restricted and specialized source of inputs to effector cells or, as we will see, to other types of neurons. (p. 29)

They are responsible for the senses and dictate the difference between a light touch and intense pain.

Motor Neurons. If sensory neurons are an input, then motor neurons are the output. After processing by earlier neurons, they stimulate other cells, such as muscle cells in the human body. Instead of sensory neurons controlling responses to external stimuli, nervous systems are also composed of another type of neuron to respond to stimuli: the motor neuron (Swanson, 2012, p. 32). These neurons take input from sensory neurons and trigger *effector cells,* such as muscles, glands, or other cells to start doing their work.

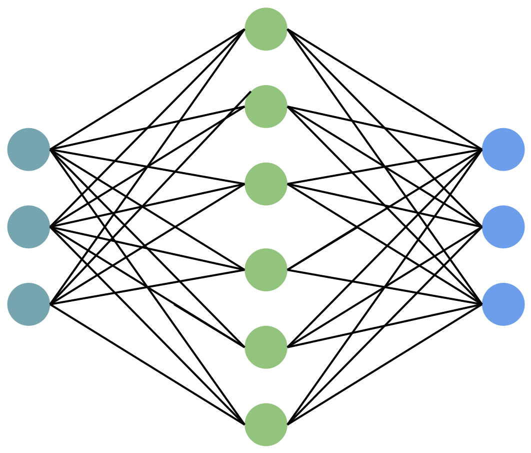
Interneurons. Interneurons relay processed stimuli from the sensory to motor neurons (“Types of Neurons”, n.d., Interneurons, para. 1). Apart from this, they communicate with each other, which allows for complex behaviors to form. They are closely related to the type of neurons modeled in (2).

Putting it Together. Biologically, sensory neurons receive input from the outside world, interneurons process it, and motor neurons act upon the environment. Mathematically, a similar concept can be achieved. Since all neurons fundamentally have the same biochemical reactions underneath, the model doesn’t have to be changed, but the placement of where neurons are within the network matters.

As shown in Figure 2, the network will have three general layers. The *input layer* is akin to the sensory neurons, receiving and processing input from the outside world. The *hidden layer* contains the interneurons, adding additional processing to the stimulus from the input layer. Lastly, the *output layer* is the final layer, and gives an output to be used however wished.

Figure 2

Visual Representation of a Network



Note. Created by the author.

Artificial Neural Networks

Artificial neural networks, ANNs, are what powers the AI used today. From ChatGPT to Tesla’s self-driving cars, they all have some form of ANN within the system, rendering them dynamic and adaptable. The most straightforward form is a *feed-forward neural network* (FFNN).

The Perceptron

The first, popularized form of the FFNN was the perceptron (Graupe, 2007, p. 18). It was a simple state machine that was slightly more advanced than a linear combinator. It takes a set of inputs, x_{ij}, and multiplies it by a set of weights, w_{ij}, then sums the results.

$$z_i = \sum_{j=1}^{m} w_{ij} x_{ij} \qquad (3)$$

The perceptron is the simplest building block for modern FFNNs. It was designed around a biological neuron and attempts to mimic its behavior. It takes the sum of an input multiplied by the weight (or strength) of that input, notated as z_i. Its successor, the Madaline model is, in essence, a multi-layered perceptron with a binary *activation function* of -1 or 1, and is still the backbone behind current FFNNs (Graupe, 2007, p. 37-39).

The Activation Function

The activation function, $f(z_i)$, is a "nonlinear function yielding the ith cell's output y_i to satisfy $y_i = f(z_i)$" (Graupe, 2007, p. 18-19). The activation function limits a neuron's output so that it's within a specific range. This effectively normalizes the output of said neuron so that every neuron works on the same scale within the network. This activation function can be anything, but its purpose is to stop the neuron from just outputting a direct copy of the sum of its inputs. This is why the activation function is generally nonlinear.

Learning

The key thing about BNNs is their ability to learn and adapt quickly. Given an image of a car, humans can identify cars of all shapes, colors, and sizes, based solely on a couple of experiences. An FFNN can also accomplish the same task, but by using more training data than the human mind needs. Both these systems accomplish this by adjusting the strength of connections between the neurons.

Synaptic Strength

Biological neurons follow the principle that “neurons that fire together, wire together” (Kennedy, 2013, para. 5). This means that a neuron that fires repeatedly will have a stronger connection to the following neuron. It also implies that a neuron that fires less often will have little to no impact on the following neuron.

This principle is what trains the network. By strengthening the used neurological pathways, BNNs can deduce patterns, as repeated stimulus yields the same set of neurological pathways (Kennedy, 2013, para. 5). Unused, and therefore irrelevant pathways are pruned, leading the network to favor predictable stimulus.

Weights

Traditional FFNNs follow this concept through a system of *weights* (Graupe, 2007, p. 18). These weights are analogous to the synaptic strength in biological neurons. Referring back to (3), it is evident that the inputs, x, are multiplied by the weights, w, to produce the neuron’s output. This multiplication replicates the synaptic strength of biological neurons.

Modern FFNNs adjust these weights through a process called *back-propagation* (Lillicrap et al., 2020, p. 5). The output neurons are evaluated in this process, and their weights are adjusted according to their error from the expected (correct) output. This process is propagated down the network until it reaches the input layer, adjusting the weights in increments proportional to their error. This process trains the network similarly to a BNN. The only difference is that biological networks are always in a state of training, whereas artificial ones are trained using a limited dataset.

Operant Conditioning

Apart from this, humans also learn based on stimuli from other sources. For instance, a young student can learn to identify numbers because the teacher rewards them with a snack. In another case, an infant learns to avoid a stove because of the pain of touching it. This process is called *operant conditioning*.

Operant conditioning is based on rewarding desirable behavior. A textbook example is *Thorndike's cat*. Thorndike's cat is placed in a box with a lever and a food dispenser. When the cat presses the lever, food falls down the dispenser, and the cat is fed. In this example, the desirable behavior is pushing the lever and the reward is the food awarded. This eventually leads to the cat learning to press the lever to get food, which the trainer would want (Schultz, 2015, p. 854).

Deriving the Model

Applying all this to constructing a *biologically approximated neural network* (BANN), the training process is a direct transfer of this concept. The model would continue to make predictions. If the prediction is correct, a positive stimulus is given, reinforcing the pattern. If the prediction is incorrect, a negative stimulus is given, forcing the model to re-evaluate its current behavior.

Base Model

Recall equations (1) and (2), they are below in (4):

$$N(x) \;=\; \left\{x \le A_p{:}0, x > A_p{:}1\right\} \qquad (4)$$

$$S_{ij} = \left[S_{i1}, S_{i2}, \ldots, S_{ij}\right]$$

$$N\left(\sum_{j=1} \quad S_{(i-1)j}\right) \; i \ne 1$$

A_p represents the threshold for an action potential; S_{ij} is a matrix of a neuron's synaptic inputs, with S_{ij} corresponding to the ith neuron; and, $N(x)$, is the neuron model. Similarly to biological neurons, it is evident that the neuron fires only when the given input can surpass the threshold, as shown in the piecewise function, $N(x)$.

Combining (4) with the principles of synaptic strength in biological networks and weights in artificial ones, a similar system can be derived. A matrix of weights, similar to that of the synaptic inputs, can be created:

$$W_{ij} = \left[W_{i1}, W_{i2}, \ldots, W_{ij}\right] \qquad (5)$$

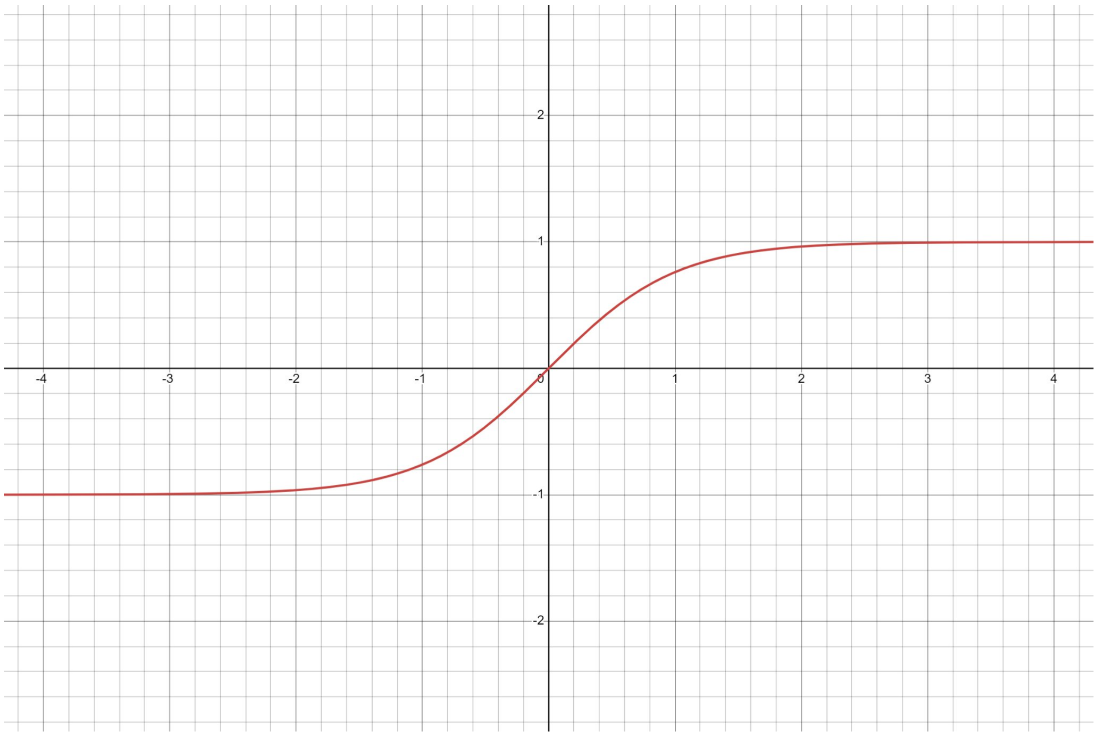
This weight matrix works the same as the synapse matrix, with W_i corresponding to the i th neuron, N_i, and W_{ij} corresponding to the jth synapse of the ith neuron. These weights can be multiplied by the synapse matrix, similar to the perceptron's equation in (3). Furthermore, these matrices will be multiplied per element to apply an individual weight to each neuron. The weighted inputs can be represented with the matrix, I_{ij}.

$$I_{ij} = W_{ij} \odot S_{ij} \;\; (6)$$

This, however, isn't enough, as an abnormally large input will skew the calculation an extreme amount. Therefore, each synaptic input must be normalized. This can be done using the activation function, borrowed from FFNNs. Since synapses can be either excitatory or inhibitory, it makes sense for the model's synapse to be able to output any value from $-1$to $+1$ — negative values representing inhibitory synapses and positive values representing excitatory ones. The function chosen must also be nonlinear, to prevent the model from being a simple linear combinator. As such, a hyperbolic tangent function is appropriate, as it has a range of $(-1,1)$. This is illustrated in Figure 3.

Figure 3

Hyperbolic Tangent Function, $tanh(x)$



Note. Created by the author.

The activation function will be applied to the sum of every synaptic input after the weights are applied, reducing it to within operating range. The following expressions summarize the operations.

$$I_{ij} = W_{ij} \odot S_{ij} \quad (7)$$

$$s = \sum_{i=1} \left(\sum_{j=1} I_{ij} \right)$$

$$N_i\,(tanh\,s)$$

The last thing is determining the threshold for firing. As discussed previously, a neuron rests at around -70 mV. The action potential occurs at around -50 mV. The issue with replicating these values exactly is that it assumes that excitatory values are negative, and inhibitory ones are positive. While this works for biology, it is unintuitive, so the BANN represents them differently. Henceforth, the chosen value for A_p, the threshold, is 0.7. The neurons will also have an ambient value of 0.5 to represent the resting voltage inside a biological neuron. This value will be represented by the value, A_r.

Putting all of this together, the final neuron model can be described in a simple equation.

$$N(x) \;=\; \left\{x + A_r \;\le\; A_p : 0,\, x + A_r \;>\; A_p : 1\right\} \qquad (8)$$

Corrective Stimulus

With the model mostly developed, it’s time to introduce the learning aspects. As of yet, the model proposed in (7) and (8) works well once the model is calibrated. However, if it is to replicate a BNN and how living organisms learn, it’s best to apply learning principles to it. As such, operant conditioning is the most straightforward and comprehensive approach that will maximize results while maintaining simplicity.

Recall that “neurons that fire together, wire together” (Kennedy, 2013, para. 5). Using this concept, a simple, yet effective training method is derived. Essentially, the more often a neuron fires, the stronger its corresponding weight. The converse also applies: the less often a neuron fires, the weaker its weight.

Mathematically, it’s a ratio of the amount a neuron fires, F_a, to how many times it could have fired, F_p. When the neuron fires at least half as much as it could have, the value of f, the firing ratio, is positive. When it fires less than half of its potential, f is negative. This is demonstrated in (10).

$$f = \frac{F_a}{F_p} - 0.5 \qquad (9)$$

f is then used for both the reward and punishment functions. The reward function is defined as follows:

$$r(w,m) = w + \{sign(w) \times m\,[f + random(0,1)]\} \qquad (10)$$

Equation (11) takes two parameters: the weight, and the magnitude, or strength, of the reward. Effectively, it strengthens frequently firing inhibitory and excitatory weights, while weakening lesser firing ones. The same concept is used for the punishment function:

$$p\,(w,m) = w - \{sign\,(w) \times m\,[f + random(0,1)]\} \qquad (11)$$

This works the same way as the reward function but strengthens less-often-firing weights while weakening frequently-fired ones.

Note that $random(x,y)$ returns a random number using a normal distribution, where $\mu = x$ and $\sigma = y$. It's necessary to add some randomness to simulate the natural variances and prevent the network from repeating the same errors. It forces the network to behave differently each time.

Testing the Model

While the proposed BANN model should work in theory, it must be tested to validate the effectiveness of modeling an ANN based on biology. Both models will attempt to accomplish the same learning task and will be evaluated based on learning speed, resource utilization, and prediction accuracy.

The Experiment

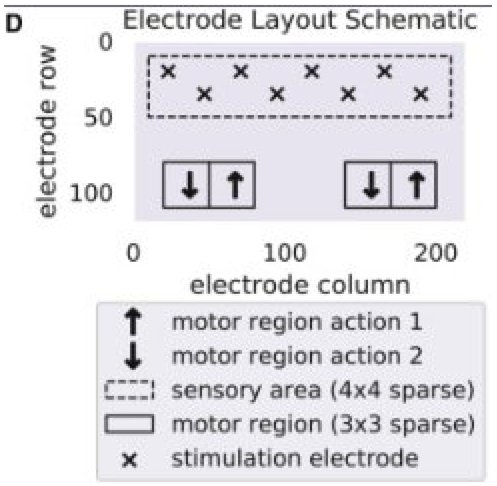
In a 2022 study, Brett J. Kagan and his associates trained a set of living, biological neurons to play the computer game, *Pong*. This system, named *DishBrain*, was a culture of biological neurons connected to a computer where the game ran (Kagan et al., 2022, p. 3958). This culture is wired up to a computer running a program that allows it to interact with the neurons in the culture via electricity probes. This is all done in real-time, with all the stimuli under the control of the researchers.

This is the perfect test for both an FFNN and the proposed BANN. It is simple enough to be replicated but with enough nuance to demonstrate the capabilities of both models. Using a custom script in the Python programming language, both models can be simulated, with the FFNN using PyTorch, a popular artificial intelligence toolkit for Python.

The Setup. DishBrain was set up according to Figure 4.

Figure 4

DishBrain Setup



Note. Adapted from “In vitro neurons learn and exhibit sentience when embodied in a simulated game-world.” by B. J. Kagan, A. C. Kitchen, N. T. Tran, F. Habibollahi, M. Khajehnejad, B. J. Parker, A. Bhat, B. Rollo, A. Razi, and K. J. Friston, 2022, *Neuron*, *110*(23), https://doi.org/10.1016/j.neuron.2022.09.001. Copyright 2022 by B. J. Kagan, A. C. Kitchen, N. T. Tran, F. Habibollahi, M. Khajehnejad, B. J. Parker, A. Bhat, B. Rollo, A. Razi, and K. J. Friston.

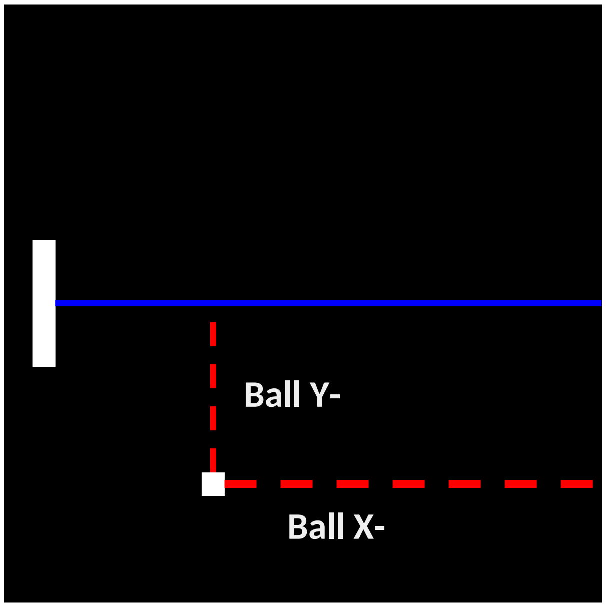
For DishBrain, a predetermined area was chosen as the output. These outputs correlated to the movement of the paddle (Kagan et al., 2022, p. 3958-3959). If there was firing in motor region 1, the paddle would move up, and if in motor region 2, the paddle would move down. This would only happen, however, for the region with the most activity, meaning the most neurons firing.

Receiving Stimulus. As for the inputs, eight electrodes were placed in the sensory area. However, because neurons operate using binary, either firing or not, the location of the ball and the action the model takes must be encoded as such.

DishBrain's encoding procedure is reasonably simple. Figure 5 shows a basic example of a moment in a Pong game.

Figure 5

Example Pong Screen



Note. Created by the author.

The ball's x-coordinate is encoded in terms of frequency, firing more often when the ball approaches the left-most wall (Kagan et al., 2022, p. 3976). The ball's y-position relative to the paddle is encoded in terms of percentages above or below the paddle. This corresponds to one of eight electrodes in the culture that serve as the inputs. For instance, when the ball is extremely above the paddle, the first electrode might get a signal, while when it's below, the last electrode would.

Using Kagen's DishBrain setup as precedent, both the BANN and ANN will undergo the same experiment. Both systems have eight inputs and four outputs, with stimulus being provided the same way as with DishBrain. As for the hidden layers, repeated trial and error during testing found that 100 hidden neurons, organized in four groups of twenty-five neurons, were enough to display learning while minimizing resource utilization.

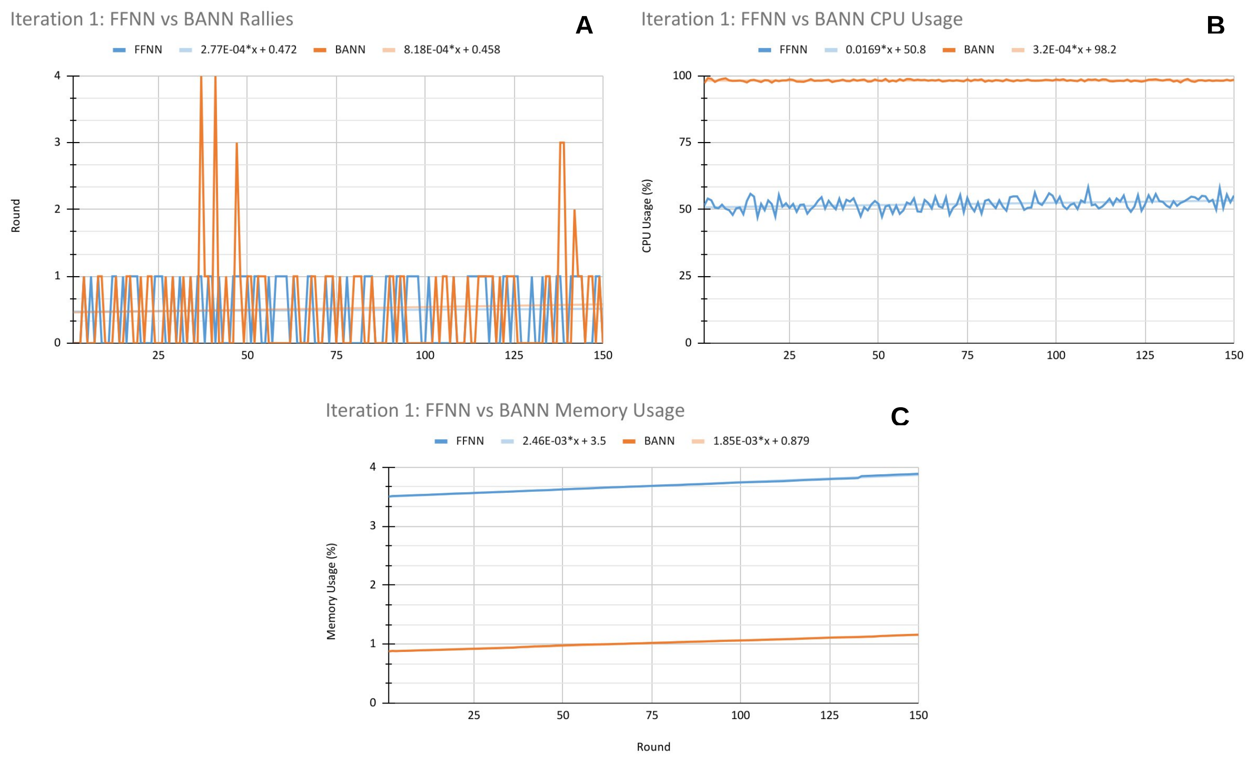
As for the inputs, the only modification will be in how the ball's x-coordinate will be encoded. While frequency-based encoding could work, the models are better suited with value-based encoding, since they receive constant input. If b_x is the ball's x-coordinate, and w_x is the game window's width, then $v_x = w_x \div b_x$, where $b_x \neq 0$. In cases where $b_x = 0$, then v_x just equals the window's width. See Appendix A for the implementation details.

The Results

The BANN and FFNN were taught to follow the ball since it led to the best results compared to letting them learn strategies independently. Both networks completed 500 rounds per trial, for 3 trials. Three metrics were collected for each trial: the average rallies per round; average memory usage per round, both in percentages and megabytes (MB); and average CPU usage per round. Trial 1's metrics are summarized in Figure 6.

Figure 6

Summarized Metrics of Trial



Note. Created by the author using experimental data from the study.

Breaking down the data for Trial 1, it is evident that there are stark differences in the collected data. The first thing that stands out is the BANN's spikes, getting a maximum score of 4. The FFNN, however, only reaches a score of 1.

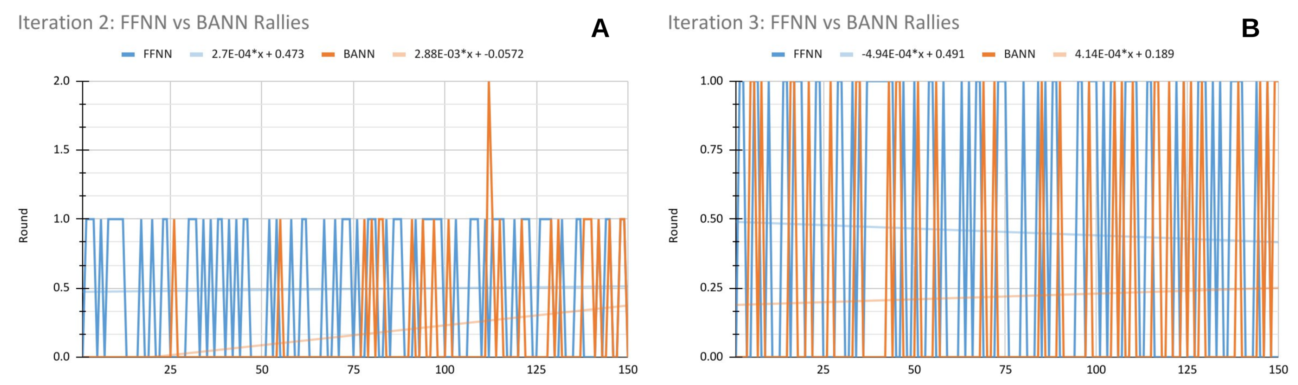
Figure 6B shows that the FFNN used less CPU than the BANN. The cause may be the unoptimized code used in the BANN script. The code is rudimentary and has room for improvement compared to the PyTorch library maintained and optimized by professional developers.

Looking at the memory utilization in 6C, it's clear that the BANN used significantly less memory than the FFNN. This may be because the BANN stores very little information, only keeping the weight values per neuron in memory. Because the FFNN uses PyTorch, the program most likely uses extra metadata, since PyTorch is an extensive library.

Trials 2 and 3 show the same trends, apart from the rallies per round. They are shown below in Figure 7. The corresponding trendlines are plotted as well.

Figure 7

Rallies Per Round for Trials 2 and 3



Note. Created by the author using data collectetd from the study.

As seen in Trial 1, the BANN's trendline slope, 8.18×10^{-4}, is almost three times as much as that of the FFNN, 2.77×10^{-4}. This indicates that the BANN improved in accuracy more quickly than the FFNN. This assumption is even more exaggerated in Trials 2 and 3. In each trial, the BANN's trendline slope was more than the FFNN.

Experimental Analysis

Comparing the results of the BANN and FFNN, it is evident that the proposed and developed BANN is as adept at learning as the FFNN. However, there are some limitations to the experiment above. First, the experiment is not representative of general applications of artificial intelligence. It is only a rudimentary example to test the model proposed in the paper. As such, further testing must be made before a conclusive claim about its efficacy can be made. Second, as previously mentioned, the custom BANN code is unoptimized and likely inefficient, especially when compared with a large-scale code base like PyTorch. This may have skewed the results concerning its CPU and memory usage.

Regardless, the trials show that the BANN can learn to play Pong better than an FFNN while using less memory. However, the FFNN is more performant when it comes to CPU usage. Realistically, this translates to the BANN using less energy than the FFNN while being slightly better in terms of learning accuracy.

Conclusion

This paper aims to answer the question "To what extent does a mathematical model of a *biological neural network* (BNN) outperform a conventional *feed-forward neural network* (FFNN) in terms of resource utilization and prediction accuracy?" After an analysis of biological neurons, a mathematical model, called the *biologically approximated neural network* (BANN), was constructed and compared against an FFNN training them to play the Pong computer game.

The results showed that the BANN was more accurate and used less memory than the FFNN, but also used more CPU. <u>This leads to the conclusion that a mathematical model of a BNN outperforms an FFNN in terms of prediction accuracy, but compromises in terms of resource utilization.</u>

Artificial intelligence has been in a rapid state of development for years. From Tesla's self-driving cars, to Alexa, to ChatGPT, AI has become prevalent in modern society. However, one of the largest drawbacks has been the amount of processing power and training data needed to train and run these models. From personal experience, training and running a traditional neural network model is not something that can easily be done using common electronics.

With more rigorous testing, the BANN may prove to be an alternative way to solve this problem without sacrificing prediction accuracy. If so, this would not only further increase the rate of development for AI models, but also reduce the energy footprint for large-scale models, such as ChatGPT, and make AI more accessible to those without abundance in processing power.

References

Dowling, J. E. (2018). *Understanding the brain: From cells to behavior to cognition* (Revised edition. ed.). W.W. Norton and Company.

Graupe, D. (2007). *Principles of artificial neural networks* (2nd ed.). World Scientific.

Hoehn, K., & Silber, K. W. (2016). Neuron. In M. S. Hill (Ed.), *Biology* (2nd ed., Vol. 3, pp. 135-140). Macmillan Reference USA. Gale in Context: Science. https://link.gale.com/apps/doc/CX3629800295/SCIC?u=spot51331&sid=bookmark-SCIC&xid=f82cc33e

Kagan, B. J., Kitchen, A. C., Tran, N. T., Habibollahi, F., Khajehnejad, M., Parker, B. J., Bhat, A., Rollo, B., Razi, A., & Friston, K. J. (2022). In vitro neurons learn and exhibit sentience when embodied in a simulated game-world. *Neuron*, *110*(23), 3952-3969.e8. https://doi.org/10.1016/j.neuron.2022.09.001

Kennedy, M. B. (2013). Synaptic signaling in learning and memory. *Cold Spring Harbor Perspectives in Biology*, *8*(2), a016824. https://doi.org/10.1101/cshperspect.a016824

Lillicrap, T. P., Santoro, A., Marris, L., Akerman, C. J., & Hinton, G. (2020). Backpropagation and the brain. *Nature Reviews Neuroscience*, *21*(6), 335-346. https://doi.org/10.1038/s41583-020-0277-3

Moazeni, S. (2023, July 27). *Q&A: UW researcher discusses just how much energy ChatGPT uses* (Interview by S. McQuate & UW News) [Transcript]. UW News. Retrieved October 27, 2024, from https://www.washington.edu/news/2023/07/27/how-much-energy-does-chatgpt-use/

Schultz, W. (2015). Neuronal reward and decision signals: From theories to data. *Physiological Reviews*, *95*(3), 853-951. https://doi.org/10.1152/physrev.00023.2014

Swanson, L. W. (2012). *Brain Architecture: Understanding the Basic Plan* (2nd ed.). Oxford University Press.

Types of neurons. (n.d.). The University of Queensland. Retrieved April 22, 2024, from https://qbi.uq.edu.au/brain/brain-anatomy/types-neurons

Vincent, J. (2024, February 16). *How much electricity does AI consume?* The Verge. Retrieved October 27, 2024, from https://www.theverge.com/24066646/ai-electricity-energy-watts-generative-consumption

Appendix A

Relevant Source Code

BANN Source Code

```
import numpy as np
import pickle
import numpy.typing as npt

FloatArray = npt.NDArray[np.float32]

class Weight(object):
    value: float | np.float32
    amount_fired: int = 0
    passes: int = 0

    def __init__(self, value: float | np.float32) -> None:
        self.value = value

class Neuron:
    resting_potential = 0.55
    action_potential = 0.7
    weights: list[Weight]
    inputs: int

    def __init__(self, inputs: int) -> None:
        self.weights = [
            Weight(np.random.normal(0, 10)) for _ in range(0, inputs)
        ]

        self.inputs = inputs

    def __activation_fn(self, input: np.floating) -> np.floating:
        return np.tanh(input)

    def fire(self, inputs: FloatArray) -> float:
        output = 0.0

        def f(i):
            index = int(i)

            if inputs[index] == 1.0:
                self.weights[index].amount_fired += 1

            self.weights[index].passes += 1

            return self.weights[index].value

        weights = np.array([f(i) for i in range(len(self.weights))])

        weighted_inputs = inputs * weights
        weighted_sum = weighted_inputs.sum()
        activated_sum = self.__activation_fn(weighted_sum)
        rp = self.resting_potential
        ap = self.action_potential

        if activated_sum + rp <= ap:
            output = 0.0
        else:
            output = np.copysign(1, activated_sum)

        return output

    def reward(self, magnitude: float = 1.0) -> None:
        for weight in self.weights:
            # negative if it fired less than the total amount of passes
            firing_rate = (weight.amount_fired / weight.passes) - 0.5
            sign = np.copysign(1, weight.value)

            # modify the weight depending on how much the neuron fires
            weight.value += sign * magnitude * \
                (firing_rate + np.random.normal(0, 1))

    def punish(self, magnitude: float = 1.0) -> None:
        for weight in self.weights:
            # negative if it fired less than the total amount of passes
            firing_rate = (weight.amount_fired / weight.passes) - 0.5
            sign = np.copysign(1, weight.value)

            # modify the weight depending on how much the neuron fires
            weight.value -= sign * magnitude * \
                (firing_rate + np.random.normal(0, 1))


class NeuralNetwork:
    dim: tuple
    __layers: list[list[neuron.Neuron]] = []

    def __init__(self, dim: tuple) -> None:
        self.dim = dim
        print(f'initializing {self.dim}')

        if len(dim) == 0:
            raise ValueError('Network must have at least one layer')

        for i, neuron_count in enumerate(dim):
            if neuron_count <= 0:
                raise ValueError('All layers must have at least one neuron')

            inputs = 1 if i == 0 else dim[i - 1]
            print(
                f'initializing layer {i} ({inputs} inputs -> {neuron_count}
neurons)')
            self.__layers.append([neuron.Neuron(inputs)
                                  for _ in range(neuron_count)])
            print('initialized')

        print('done')

    def predict(self, inputs: list[float]) -> list[float]:
        previous_outputs = np.array(inputs, dtype=np.float32)
        current_outputs = np.array([], dtype=np.float32)

        for i, layer in enumerate(self.__layers):
            outputs = []

            for j, neuron in enumerate(layer):
                output = 0.0
                if i == 0:
                    output = neuron.fire(np.array([previous_outputs[j]]))
                else:
                    output = neuron.fire(previous_outputs)

                outputs.append(output)

            current_outputs = np.array(outputs, dtype=np.float32)
            previous_outputs = current_outputs

        return current_outputs.flatten().tolist()

    def reward(self, magnitude: float = 1.0) -> None:
        for layer in self.__layers:
            for neuron in layer:
                neuron.reward(magnitude)

    def punish(self, magnitude: float = 1.0) -> None:
        for layer in self.__layers:
            for neuron in layer:
                neuron.punish(magnitude)

    def save(self, loc: str) -> None:
        layer_weights = [
            [Weights(neuron.weights) for neuron in layer]
            for layer in self.__layers
        ]

        serialized = SerializedNetwork(self.dim, layer_weights)
        with open(loc, 'wb') as file:
            pickle.dump(serialized, file)

    @staticmethod
    def load(loc: str):
        network: NeuralNetwork

        with open(loc, 'rb') as file:
            unpickled: SerializedNetwork =  pickle.load(file)
            network = NeuralNetwork(unpickled.dimensions)

            for i, layer in enumerate(network.__layers):
                for j, neuron in enumerate(layer):
                    neuron.weights = unpickled.layer_weights[i][j].weights
        return network

class Weights(object):
    weights: list[neuron.Weight]

    def __init__(self, weights: list[neuron.Weight]) -> None:
        self.weights = weights

class SerializedNetwork(object):
    dimensions: tuple
    layer_weights: list[list[Weights]]

    def __init__(self, dimensions: tuple, layer_weights: list[list[Weights]])
-> None:
        self.dimensions = dimensions
        self.layer_weights = layer_weights
```

FFNN Source Code

```
import torch
from torch import nn

class NeuralNetwork(nn.Module):
    dim: tuple
    layers: list[nn.Module]
    linear_relu_stack: nn.Sequential

    def __init__(self, dim: tuple):
        super().__init__()
        self.dim = dim
        layers = []

        for i, inputs in enumerate(dim):
            layers.append(nn.LazyLinear(inputs))
        if i < len(dim):
            layers.append(nn.Tanh())

        self.layers = layers
        self.linear_relu_stack = nn.Sequential(*layers)

    def forward(self, inputs: list[float]):
        return self.linear_relu_stack(torch.tensor(inputs))
```

Input Configuration Source Code

```
from ai.pong import PongGame

def get_inputs(game: PongGame) -> list[float]:
    # First 4 = above the paddle
    # Second 4 = below the paddle
    inputs = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]

    paddle = game.paddle
    ball = game.ball
    winsize = game.renderer.win_size
    block_size = winsize[1] / 8
    intervals = []
    ball_distance = (winsize[0] / ball.rect.centerx) if ball.rect.centerx !=
0 else winsize[0]

    index = 0

    for i in range(-3, 3):
        current_offset = block_size * i
        last_offset = block_size * (i + 1)
        paddle_loc = paddle.rect.centery

        intervals.append(
            (paddle_loc + current_offset, paddle_loc + last_offset))

    intervals.insert(0, (0, intervals[0][0]))
    intervals.append((intervals[len(intervals) - 1][1], winsize[1]))

    colors = [
        (139, 0, 0),  # darkest red
        (192, 64, 0),  # medium red
        (255, 99, 71),  # light red
        (255, 192, 203),  # lightest red

        (173, 216, 230),  # lightest blue
        (0, 191, 255),  # light blue
        (0, 0, 192),  # medium blue
        (0, 0, 139)  # darkest blue
    ]

    for i, interval in enumerate(intervals):
        if interval[0] <= ball.rect.centery <= interval[1]:
            inputs[i] = ball_distance
            index = i

    ball.image.fill(colors[index])

    return inputs
```