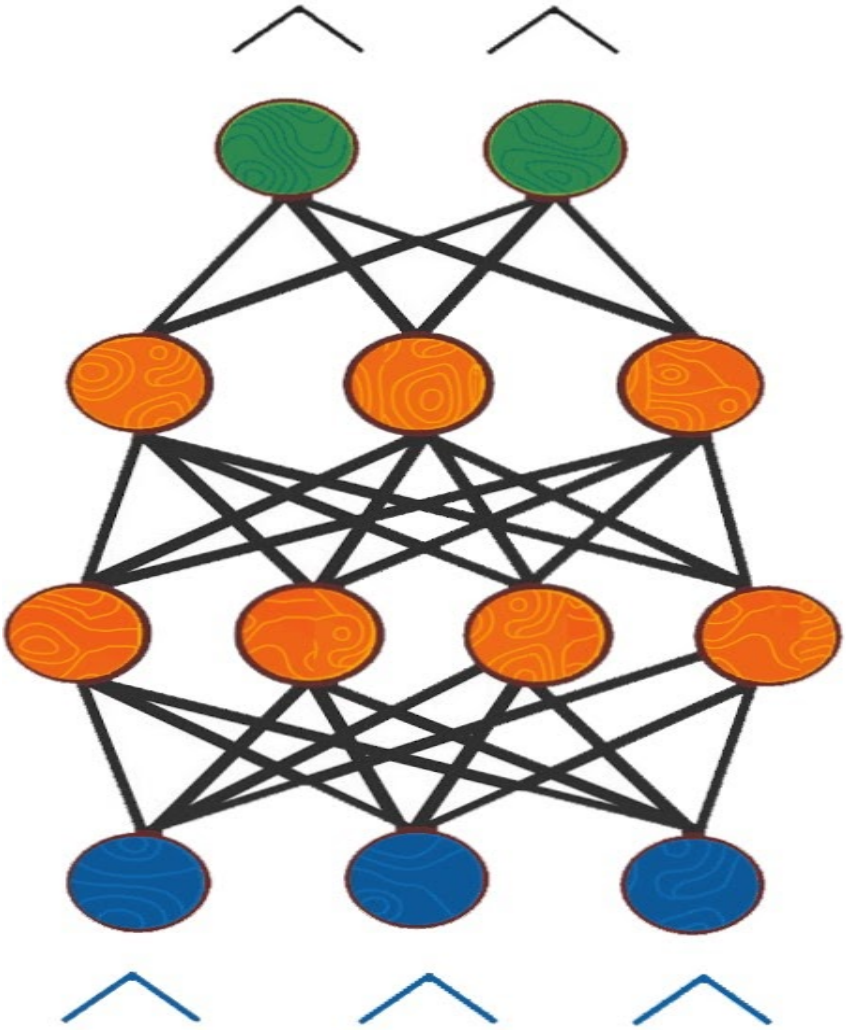
Applying Machine Learning Algorithms to AI in different Complex Environments and Simulations



Jake Watson

@j_owatson

Analysis

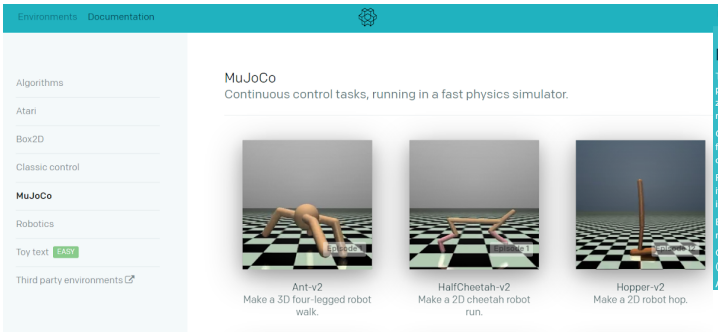
Background

As Artificial Intelligence continues to dominate the tech industry and lead the world in innovations and amazement, many still don't know or understand how the technology works; so for the foundations of my program I have decided to make a learning application that teaches about Artificial Intelligence and how they are used and trained to perform tasks. The program will do this through the use of video games and simulations. I plan on teaching the user through the use of them experimenting with AI in given environments. The idea of using environments more specifically games to measure AI performance isn't new and has been used by major companies like Google (Deepmind), OpenAI (Gym), and Microsoft (MineRL). It might not seem obvious now however this program could be combined or expanded with an Arduino system to create AI-powered robots, which could learn about their environments through simulation or physically interacting with them. The end goal is to create a program that allows a user to create custom AI's that can learn and play in different environments, the AI's should be highly customizable with the ability to upload them to a server so that they can compete and be downloaded multiple times by different users to experiment with them.

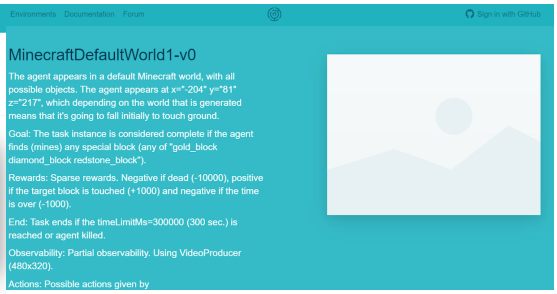
[Figure 0]

Inspiration

AI was first dreamed up by Alan Turing, while not a typical AI Turing was curious if a machine could pass off as a human on a test, not long after this one of the first primitive AIs was written to play checkers. Setting the first AI to play a computer game. Years later video games are still used to measure the performance of AI systems, one of the most famous uses of this is OpenAI's gym which has dozens of games and environments for people to test and train their AI on. Figure 1 and 2 shows some environments that OpenAI offers (Figure 2 is a closed competition of a Minecraft AI using the openai gym software)

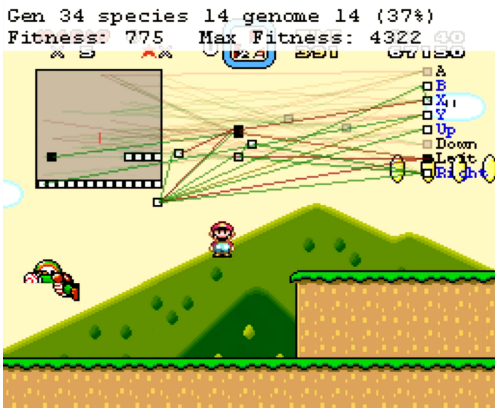


[Figure 1]



[figure 2]

OpenAI of course isn't the only company to use games to rate AI performance, Google's deepmind project has also used games to rate the performance of their AI. One of their most notable AIs was Agent 57 which was able to play all 57 atari games, most of which were above the human level of ability. All of these example AIs mainly used deep Q-Learning as a way to train the agents a type of unsupervised learning algorithm which is how I'll be training my agents to perform in environments, for my final piece of inspiration I'll be using Sethbling's MarI/O AI which was able to play Mario 64 through simulated evolution which is an idea which fascinates me. The idea of being able to evolve a machine to perform actions in a given environment and see it succeed in what it's doing seems like a piece of science fiction.



[Figure 3]

Hopefully, by the end of this project, I wish to achieve an AI which has learned to interact with

its environment, after all the type of AI I plan on using (a CNN) is simply put a series of floating-point numbers called weights which are typically between -1 and 1 that are multiplied through dot product the output is then passed through an activation function which produces probability/confidence of the AI; typically training one of these AIs is done through backpropagation where the activation function is derived and an optimal value for the weights is calculated. The concept of unsupervised machine learning however is much different since the backpropagation isn't really used. Unsupervised learning is mainly used when no labels are given for a dataset. I will be using unsupervised learning to let AIs play computer games and learn in their new environment.

Research

Deep Q Learning

The algorithm I'll be using is Deep Q learning and backpropagation algorithms. Deep Q Learning utilises the Bellman equation to decide the optimum outputs for the network, the equation for Deep Q Learning is described as

$$Q_{new}(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \cdot maxQ(s_{t+1}, A) - Q(s_t, a_t)]$$

Where, $Q_{new}(s_t, a_t)$ is the new output of the neural network at position t in the training set, where a represents the output vector of the neural network, and s is the observation of the neural network (inputs to the neural network).
By breaking down the Bellman equation into 4 major parts we get,

$$Old\ Q\ state = Q(s_t, a_t)$$
$$temporal\ Dif = \alpha[r_t + \gamma \cdot maxQ(s_{t+1}, A) - Q(s_t, a_t)]$$
$$discounted\ future\ Q\ state = \gamma \cdot maxQ(s_{t+1}, A)$$
$$the\ new\ Q\ states = Q_{new}(s_t, a_t)$$
$$temporal\ Dif\ target = r_t + \gamma \cdot maxQ(s_{t+1}, A)$$

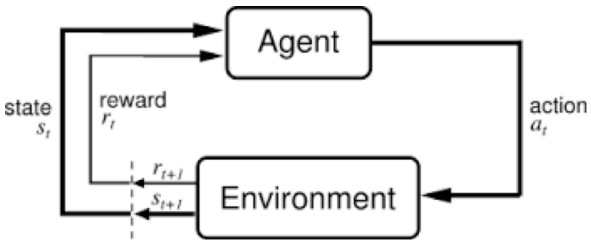
The $Old\ Q\ state$ is a function that will return the value a_t which is the output we get from observing the environment in the current time step. Each element in a_{old} is updated iteratively, each value updated will be called a_t; a for the action value of a_{old}, and the subscript t for the time step.

$temporal\ Dif$ is how much well update the value of a_t to produce the new desired output for $Q_{new}(s_t, a_t)$. We use the $temporal\ Dif$ to update every value in a to produce the outputs for $Q_{new}(s_t, a_t)$.

The $discounted\ future\ Q\ state$ is the predicted maximum value for a vector A output for the observation in s_{t+1}. A discount is then applied to this due to the uncertainty of future possibilities. We use this to decide whether or not the action we just took was a good decision or not.

And finally, our $temporal\ Dif\ target$ which is what is used to generate training outputs for the neural network. This function will add the reward to the maximum discounted future action this new vector that is created will be used to train the Neural Network, however, due to Deep Q Learning taking a long time the max will be dropped in favour for $r_t + \gamma \cdot Q(s_{t+1}, A)$ This new version of Deep Q Learning is known as SARSA and works in fundamentally the same way as before except we discount all future states and add the reward to the action we chose to use and then trained on the new target.

The beauty of using Deep Q Learning is that it can be represented as a very simple loop. (figure 4 shows an example of this loop) The graph to the left shows that as the AI agent interacts with the environment in the form of an action a_t, the environment returns a new state and a reward value s_t and r_t respectively.



[Figure 4]

Backpropagation

The other algorithm we'll be utilizing is backpropagation, which is used to train and update the AIs' weights and biases.

Consider a vector that has 2 randomly generated decimal numbers $\omega = [0.5,\ 0.3]$, another vector which is a unit vector of the same size $\hat{x} = [1,\ 0]$, and finally a random integer $b = 6$. This is our simplest model of a Neural Network, a perceptron, consisting of an input and an output with no hidden layers; the perceptron is the best way to understand the basics of a Neural Network. Before we do the maths of how a Neural Network thinks and trains, there is one last item left to cover; the activation function. The activation function is what we'll use to compress the outputs down to a manageable size since through training our vector ω can sometimes produce some massive values for outputs; activation deals with this by normalising our outputs. The function that will be used in this example is the $Hyper\ Tangent$ activation function described as,

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}},$$

whose derivative is defined as,

$$\frac{\partial\ tanh(x)}{\partial x} = 1 - \frac{(e^{x} - e^{-x})^{2}}{(e^{x} + e^{-x})^{2}}$$

The first step in our perceptron is to dot product our $\hat{x}$ by our ω vector to produce our unbiased output, we then add b to this output and pass the output which we'll dub y through our activation function.

$$c = \sum_{i=1}^{2} \omega_{i} \times \widehat{x}_{i}$$

$$w \cdot \widehat{x} \Rightarrow (0.5 \times 1) + (0.3 \times 0) = 0.5$$

Now we add our bias and pass through the activation function,

$$y = tanh(c + b)$$

$$tanh(0.5 + 6) = 0.999$$

Now that we have an untrained output we can prepare to calculate our $error$ for the perceptron, we start by subtracting the predicted value y from the real Y value which we'll say is 0 in this scenario.

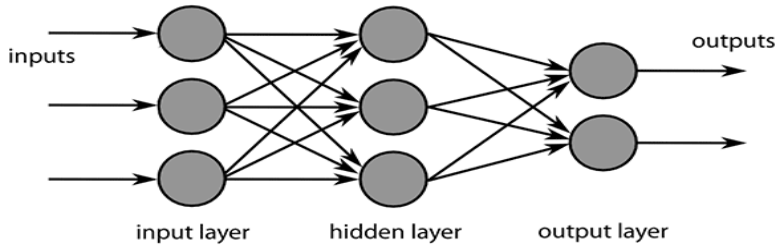
$$\varepsilon = \mathrm{Y} - y$$

$$0 - 0.999 = \ -0.999$$

This value, -0.999, is then used to calculate how much we should change each value in the vector ω; a similar is method is also used to change our biassed value, both equations are described below,

$$\omega_{i,new} = \omega_i + \ \varepsilon \cdot \alpha \cdot \widehat{x_i} \cdot tanh'(c + b)$$
$$b_{new} = b + \varepsilon \cdot tanh'(c + b)$$

We do these equations over and over again with different values for Y and vector $\widehat{x}$ until we get a perceptron that can accurately make predictions, which we trained it on. In reality, however, due to a perceptron's simplistic behaviour, it cannot make predictions for an XOR gate or decide on whether a picture is of a cat or dog. To do this we need a deep neural network. The maths for the most part is the same except we adapt it for matrices instead of vectors, the maths is used in a way to update potentially thousands of parameters. Each element in the matrix will represent a different weighted value of the network. Deep Neural Network models can do very complex things compared to perceptrons; this includes XOR gate, image recognition, and playing video games which is what we'll be doing. To the left is a diagram of a simple Deep Neural Network, the purpose of this network may be to predict if something is safe or not or to go left or right.



[Figure 5]

Compared to a perceptron, a Deep Neural Network (which from now on I'll be referring to as a Neural Network or Network for simplicity) is far more complicated, training a Neural Network of this type requires a lot more maths to train; so let's get into it.

The first thing we do is generate a random weight matrix whose number of columns is equal to the number of inputs, and rows are equal to the layers' output size. We do this over and over again till we reach the desired layer count, for example, say we wanted a Neural Network representation for the network above we would generate a "3x3 matrix"

for the first layer/set of connections and a “2x3 matrix” for the second layer. We could represent these layers as

$$let\ R \in \mathbb{R},\ R \in [-1, 1],$$
$$W_{ij}^{IH} = \begin{bmatrix} R & R & R \\ R & R & R \\ R & R & R \end{bmatrix}$$

$$W_{ij}^{HO} = \begin{bmatrix} R & R & R \\ R & R & R \end{bmatrix}$$

As you can see we’ve produced two random matrices whose superscript describe their location in the Neural Network, the superscript IH for example means $input\ to\ hidden$, and the superscript HO means $hidden\ to\ output$. The second step-in backpropagation now requires us to have a training set, which we’ll use an imaginary dataset to fit the network. The input for this Neural Network is a “1-row 3 columns vector” which we’ll call X and define it as,

$$X = [x_1,\ x_2,\ x_3],\ x \in \{1, 0\},$$

As you can see we’ve generated our input vector, now time for our output vector for this we’ll do the same however it’ll be a “1-row 2 columns vector”,

$$Y = [y_1,\ y_2],\ y \in \{1, 0\},$$

Now that we’ve got inputs and outputs we can begin to train our Neural Network, the first step is to multiply our matrix W^{IH} by our input vector X to get an output vector,

$$H = W^{IH} \bullet X,$$

Where H is a “1-row 3 columns vector” which can now be passed as inputs to W^{HO}

$$Y_p = W^{HO} \bullet H,$$

This multiplication will result in a vector which we’ll be the Neural Networks prediction on the input data. Since this is the Networks first attempt we can expect random outputs from the Network, to get the accuracy or this random guess for we subtract the true value from the guessed value,

$$\varepsilon = Y_{True} - Y_p,$$
$$\bar{\varepsilon} = \frac{1}{n}\sum_{i=0}^{n} E_i$$

As you can see from the equation above, the error is a very simple calculation. However, if we wanted our average error, say for plotting an error-time graph or error-epoch graph, then our average error can become very useful for measuring the performance of our Network as time goes by. You may also notice that our error equation is nearly identical to our perceptron error function, and that's because it is. You may be confused now since our Network is composed of multiple matrices instead of a single vector, and that's ok since this is a difficult problem to understand and fully wrap your head around so let's go into the realm of differentiation, chain rule, matrix & vector multiplication and finally transposition comes into play.

Maths breakdown

That may seem like quite a lot, however, I'll do a brief summary of all the meanings of these so you're not as confused. The first topic is differentiation and is probably the most simple topic on our list next to transposition. Simply put, differentiation is the method of finding the rate of change of a graph or equation. Differentiation is often put as $\frac{df(x)}{dx}$ where $df(x)$ is the function to differentiate and dx is the input to the function, the d in the formula represents change so don't worry too much about that as it's often represented in other ways.

The next topic to tackle is the chain rule, chain rule is the method of calculating the derivative of two or more functions that have been multiplied together. The chain rule is an integral part of backpropagation for the Neural Networks, without it Neural Networks would probably never exist. The equation of the chain rule is as follows

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx},$$

where y and u are functions, an example of this would be

$$(2x+1)^2, \text{ where } y = x^2, \text{ and } u = 2x+1 \text{ so}$$
$$\frac{dy}{dx} = (2x+1)^2 \Rightarrow 2(2x+1)\cdot 2 \Rightarrow 4(2x+1)$$

Using these methods we can use the chain rule to backpropagate through the network to calculate the error of each part of the neural network. Note that sometimes $\frac{dy}{dx}$ may be represented as $\frac{\partial}{\partial x}$

With chain rule covered we can now move onto the matrix and vector multiplication, this is quite a broad expression since there are many ways to do multiplication on a vector or matrix, however, the ones we'll be covering are matrix dot product and vector dot product. This may seem confusing and it is a bit so I'll only be using visual examples so you can get a moderate understanding of what we'll be doing.

So let's start with the matrix dot product. The matrix dot product is probably the most common function we'll use since it's what's required to get output from the network and is also used in training; to do matrix dot product, one rule must be followed. The width of matrix A must be equal to the height of matrix B. Matrix dot product can be represented as

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} a*e+b*g, & a*f+b*h \\ c*e+d*g, & c*f+d*h \end{bmatrix}$$

As you can see the new matrix has the height of matrix A, and the width of matrix B. Matrix dot products will appear a lot in this project so make sure you have a good understanding, the same goes with vector dot products as that appears a lot too and is what we'll be covering now.

The vector dot product is very similar to a matrix dot product as a vector can be put very simply as a 1-dimensional matrix. Vector dot is very useful since a column vector dot produced with a row vector can produce a matrix as an output; however, we won't be doing that as that can be classed as a matrix dot product since the vectors can be classed as 1-dimensional matrices. Instead what we'll be doing is multiplying two vectors of the same size together and summing their output to get a single value as an output, identical to how our perceptron worked.

$$\begin{bmatrix} a \\ b \end{bmatrix} \bullet \begin{bmatrix} c \\ d \end{bmatrix} = a*c+b*d$$

The vector dot product can be expressed as the following equation

$$A \bullet B = \sum_{i=1}^{n} A_i B_i$$

Where A and B are vectors whose dot product is equal to the sum of each element in the vector multiplied by the other vectors' corresponding value.
As you can see vector dot products are very simple, just like transposition, which is what we'll be briefly covering now.

Transposition may sound complex however it's a very simple process of flipping a matrix around, matrix transposition is often denoted by the superscript T.
Transpositioning is required to calculate our hidden errors in Neural Networks.
Transposition can also be applied to vectors to turn them either into column or row vectors. A matrix that has been transposed will look like its values have been flipped like the matrix below.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^T = \begin{bmatrix} a & c \\ b & d \end{bmatrix}$$

As you can see the matrix values have been flipped around; this is a major part of the backpropagation algorithm for neural networks, as without it training would be very difficult to do perform since we wouldn't have a way to measure the hidden error of the neural network which we'll cover soon.

Back to backpropagation

So now that we've covered the maths we can finally head into the backpropagation algorithm before we covered the maths above we calculated the error of the network after making a prediction on the dataset inputs $\widehat{X}$ we can make a fairly accurate assumption that our network was unable to predict the correct output and most likely had a large error.

If we take a step back to our perceptron we can express the output as an equation for a straight line $y = mx + b$, the problem for our neural network is that it has multiple layers and cant be expressed as a straight line graph, so how exactly do we update each layer? The answer, we use the chain rule to update each layer's weight in an iterative fashion. This process is a bit different from our perceptron model since a perceptron consists of

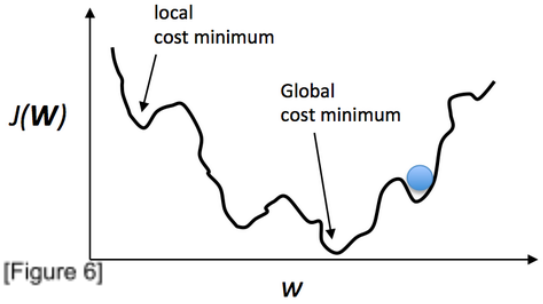
1 layer, compared to a large number of layers that our neural network can have. So the first step in updating our weights is to calculate the Δw for the output layer, this is done by taking the output vector and passing it through the derivative of our activation function and then multiplying it by our error vector.

$$\Delta w_{h \to o} = (tanh'(Y_i) \bullet \varepsilon) \bullet X_i^T$$

The equation above demonstrated how we can calculate our delta weights; in this equation X_i^T represents the input to the current layer that we would like to update and the symbol $\odot$ represents element-wise multiplication of two vectors; this simply put means that each element in vector one is multiplied with the corresponding element in vector two. This means for our delta weight equation that the derived output for that layer is multiplied with error for that layer which is then dot produced with input for the layer. Now this works if we only have one layer, like our perceptron, however, our Neural Network has multiple layers so how do we calculate the errors of each layer? The answer is we use the dot product to calculate the error of the layer, the error of a given layer is defined by the following formula,

$$\varepsilon_i = W^T \bullet \varepsilon_{i+1}$$

Where ε_i is the error of the layer you would like to calculate, W^T is the weights of the next layer in the series transposed, and finally, ε_{i+1} is the error for W^T. Once the delta weights are calculated they are then added to the current weights, however, the delta weights are typically multiplied by a scalar known as alpha or the learning rate.



[Figure 6]

We multiply weights by this scalar to prevent our neural network from getting stuck and not learning. This can be plotted on a graph where our cost $\frac{1}{2}(Y - y)^2$ is on the Y-axis and the weight is on the X-axis, figure 4 shows an example of what this graph may look like, as you can see it's a graph that resembles a quadratic that has been passed through a noise filter. The graph has many local minimums which can often lead to the Neural Network becoming stuck, when the network has first generated the error will most likely be high but as we train the network we'll begin to reduce the error this could result in the network being stuck at a local

minimum. To prevent this we use a learning rate to perform gradual steps, this can help prevent us from falling and becoming in local minimums.

The complete formula for updating weights is,

$$\varepsilon_i = W^T \bullet \varepsilon_{i+1}$$
$$W_{updated} = W + \alpha \bullet ((\ tanh'(Y_i) \bullet \varepsilon) \bullet X_i^T)$$

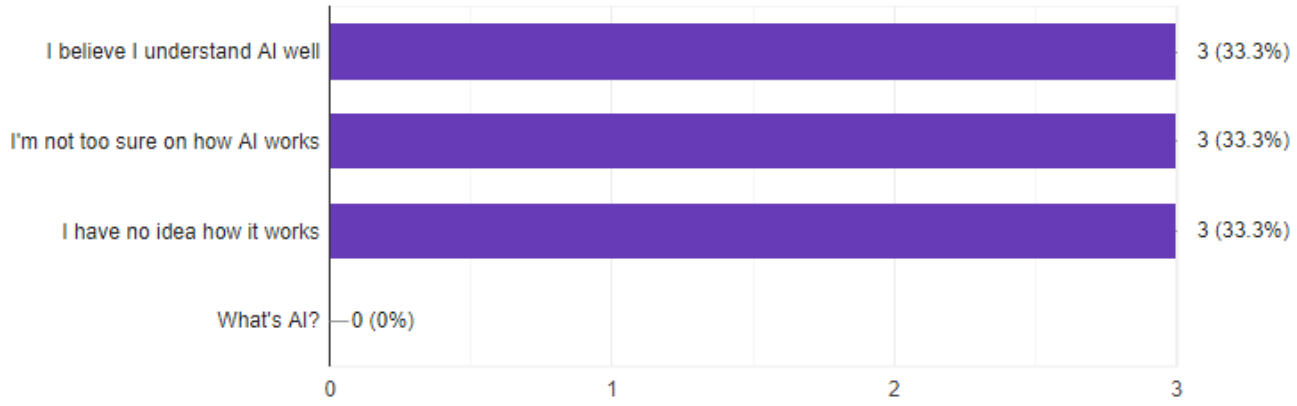
Now that the main maths behind Neural Networks has been covered we can move onto the next part of the research, the people survey.

Survey

The survey covers what people think about AI and their opinion of it, the survey covers some interesting topics including whether AI is a threat, how intelligent it is, and whether or not AI could play complex computer games.

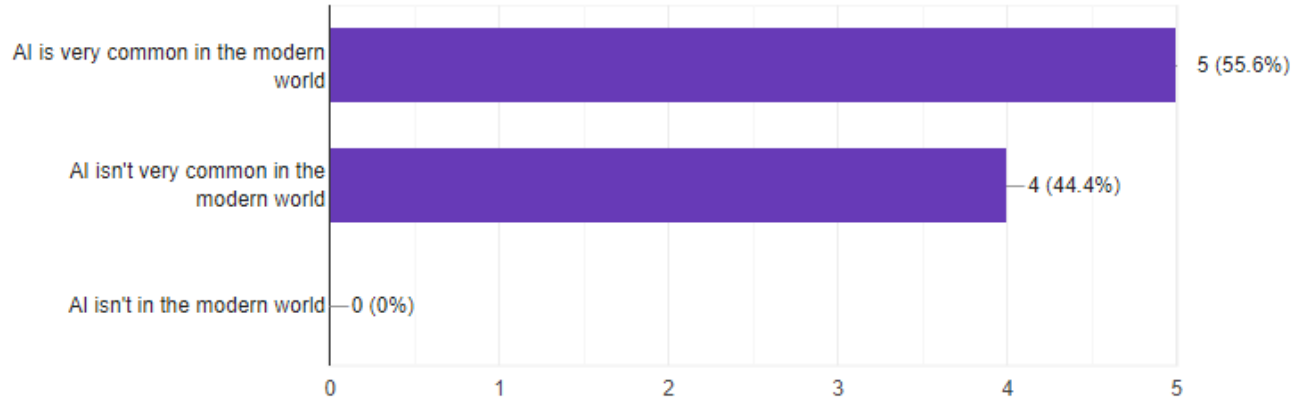
How well do you understand AI?

9 responses



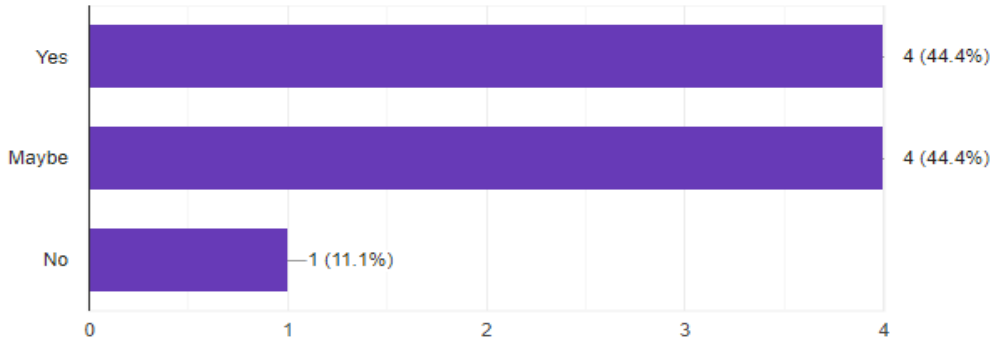
How common would you say AI is in the modern world?

9 responses



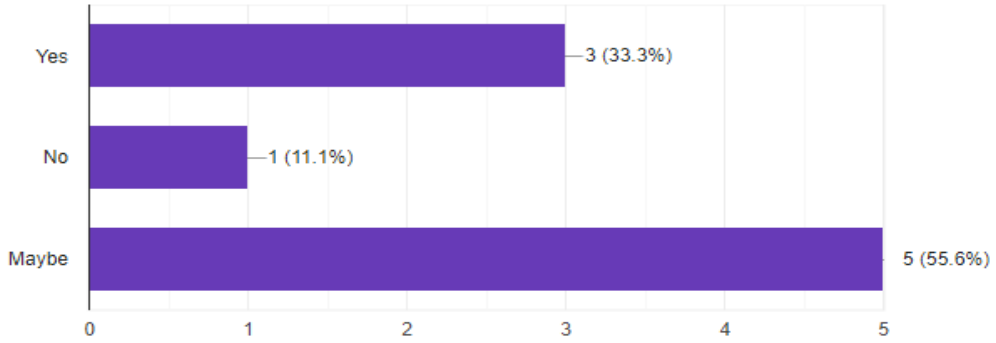
Do you believe that AI will be better for humanity?

9 responses



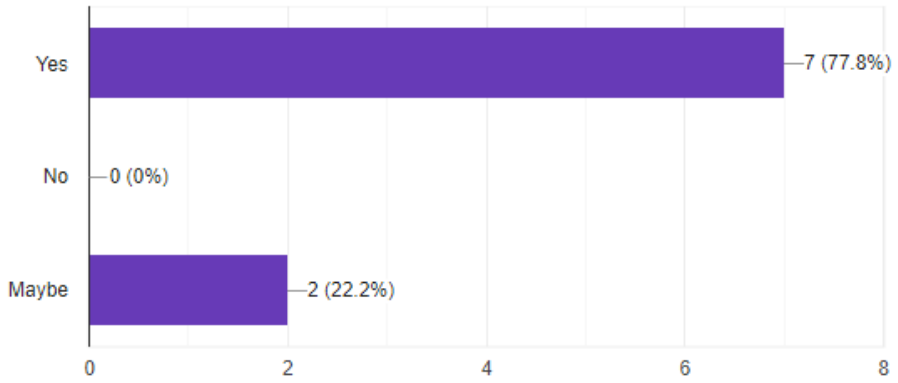
Would you say that AI is a threat to the world?

9 responses



Do you think it would be interesting to see an AI play a computer game and would you watch it?

9 responses



As you can see a large majority of the people surveyed stated that they would like to see an AI learn and play a video game or simulated environment, a large portion of people also seem confused about how AI works and how common it is in the modern world. The data here suggest that people would be interested in an application like the one I'm developing. It's not too far-fetched to state that the application could be used to inspire more people into STEM as well as further propel the app into the market it has been designed for.

Objectives

1. The program must have an effective AI engine
   1.1 - The engine must be able to carry out the matrix and vector operations
   1.2 - The engine must be able to solve AND & OR gate problems
   1.3 - The engine must be able to solve XOR problems with high accuracy
   1.4 - The engine must be easily expandable
   1.5 - The engine must be fast and use threading

2. The main application must have an easy use GUI
   2.1 - The application must be login protected
   2.2 - The application must be intuitive
   2.3 - The application must allow the user to customise their AIs
   2.4 - The application must have a save and load function for any AI model
   2.5 - The application must run any games/simulations on separate threads
   2.5 - The application must explain what the AI is doing

3. The application must be fast and have acceptable speeds & be optimised
   3.1 - The application must be able to train the AI quickly
   3.2 - The application must use threads for training
   3.3 - The application must run the games and sims at acceptable frame rates
   3.4 - The application must not consume an excessive amount of system resources to achieve the previous goals

4. The application must display live AI information & statistics
   4.1 - The application must display the average error while training
   4.2 - The application should display the output vector of the network
   4.3 - The application should save error overtime if requested

5. The application login system
   5.1 - The application should have a back end server
   5.2 - The main application should quickly connect to the database
   5.3 - The main application should encrypt all user data with a hashing algorithm
   5.4 - The main application should allow users to change the databases IP if they need to change it
   5.5 - The networking in the application should be fast and encrypt all traffic

6. The application should allow users to load and save AI models
   6.1 - The application should allow users to load and save models
   6.2 - The application should use a “*.csv” file extension for loading and saving AI models
   6.3 - The storing format should be similar to CSV
   6.4 - The algorithm to store and load files should be fast and efficient

7. The user should be encouraged to experiment with different configurations
   7.1 - Tooltip text should be used on inputs to encourage creativity
   7.2 - Tooltip text will be used to inform people on what each input does

The purpose of the more visible objectives is to inspire creativity and education in the users; it also allows the user to use a more readable and interactive interface rather than using a command line, while the hidden objectives allow the application to work at a faster more effective level. The objectives will also be completed more or less in that order, the reason for this is to get the larger segments of code out of the way first to increase productivity, then to tinker on the remaining objectives to try and get them working last. This should allow plenty of time to work on debugging and optimization of the code, fortunately, the use of a low-level language like java allows for relatively fast running code that shouldn’t require too much optimization.

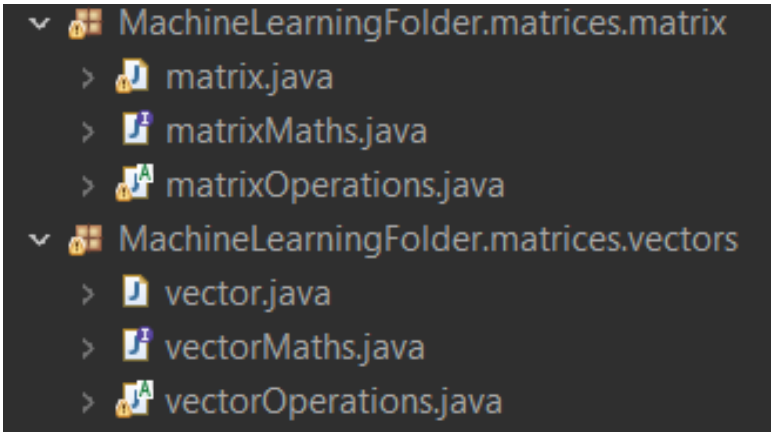
Design Section

The project I have undertaken contains many complex algorithms, from creating a library to handle matrix operations, to physics engines to calculate and render an environment in real-time. It can easily be said that my project is rich in algorithms, the main algorithms utilised are;

- Matrix operations/maths
- Deep Q Learning
- Neural Networks
- Reinforcement Learning
- Simple physics engine
- Time loops
- Linear/Non-linear activation functions
- Login system
- Server programming
- Hashing algorithms for encryption

As you can see there are quite a few important algorithms that are used to make my project work, the algorithms used vary in complexity but all are required to make my project function.

As you can see in the image below we have quite a few classes that are required to handle matrix operations and maths. These classes are mainly used by the class named GenerateNetwork to perform calculations that make the AI operate. Each class has an estimated average of about 100 lines making it impractical to screenshot. The whole thing, to compensate for this I will only screenshot the main pieces of code that hold the most value.



If you're more interested in this I will create a GitHub repository with the source code, compiled project, class diagram (this will be large) showing how all classes are related, as well as this document.

Neural Network Design

```
public generateNetwork(int inputs, int layers, int density, int outputs, double learningRate, double gamma, Object activation) {
    //all network parameters
    this.inputs = inputs;
    this.layers = layers;
    this.density = density;
    this.outputs = outputs;
    this.alpha = learningRate;
    this.initAlpha = alpha;
    this.gamma = gamma;
    this.func = activation;
    //initialising tensors for training
    this.weightTensor = new matrix[layers+1];
    this.p = new momentum[layers+1];
    this.deltaTensor = new matrix[layers+1];
    this.errorTensor = new vector[layers+2];
    this.outputTensor = new vector[layers+2];

    for(int layer = 0; layer<this.weightTensor.length; layer++) {
        if(layer == 0) {
            this.weightTensor[layer] = new matrix(this.inputs, this.density);
            this.weightTensor[layer].gaussianRandom(10);
            this.p[layer] = new momentum(0.9d, this.weightTensor[layer]);
            continue;
        }else if(layer==this.weightTensor.length-1) {
            this.weightTensor[layer] = new matrix(this.density, this.outputs);
            this.weightTensor[layer].gaussianRandom(10);
            this.p[layer] = new momentum(0.9d, this.weightTensor[layer]);
            continue;
        }
        this.weightTensor[layer] = new matrix(this.density, this.density);
        this.weightTensor[layer].gaussianRandom(10);
        this.p[layer] = new momentum(0.9d, this.weightTensor[layer]);
    }
}
```

Initialisation of parameters in constructor, constructor is overridden to allow for the loading of previous AI models

Initiating weight tensors as well as learning optimiser to speed up training in the more complex environment

The above constructor for the generateNetwork class takes in multiple parameters, it is overridden however the majority of the time it's referenced it has 7 parameters when referenced. It has a time complexity of O(n); this is a result of the singular for-loop. The first set of variables to be initialised is the basic core of the network, this consists of the number of inputs, outputs, etc. The second variables to be initialised are all arrays

and are called “tensors” by the comment, these are the weights as well as what is used to record the error, output, and delta weights during training. The array “p” is used to calculate momentum for the AI when training, this can be used to greatly reduce training time and resources by considering the gradient algorithm as rolling a ball down a hill (refer to figure 6 if confused) typical gradient descent may get the “ball” stuck at local minimums. However, by giving momentum to the algorithm the “ball” can skip some of these minimums, allowing it to converge quicker. The equation for this is below.

$$\upsilon_{dw} = \beta \bullet v_{dw} + (1 - \beta) \bullet \Delta w$$

$$W_{updated} = W + \alpha \bullet \nu_{dw}$$

The top equation calculates our Δw with momentum, β is typically close to 0.9 but can be made a value to whatever will benefit the user. We may also do this for the bias and that can be described as,

$$\upsilon_{db} = \beta \bullet v_{db} + (1 - \beta) \bullet \Delta b$$

$$b_{updated} = W + \alpha \bullet b_{dw}$$

```
public vector feedForward(vector input) throws Exception{
    vector output = input;
    for(int layer = 0; layer < this.weightTensor.length; layer++) {
            output = ((functions)this.func).activated(this.weightTensor[layer].matrixDot(this.weightTensor[layer], output));
    }
    return output;
}
```

This is our method to feed the network with an input and return an output.

The method above is used to feed data into the network; it is the coded version of our neural network. This simple method is one of the most important lines in the entire program, when called the method will require a parameter; the parameter is a vector called input. The output is set equal to the input; however, it is quickly overridden once the for loop runs, the for loop is used to calculate the output of each layer in the neural network. You may also notice “((functions)this.funct)” this casting is used so the user can use any activation function they want; it works by making the class functions an interface instead of class. This allows the casting to take place without the risk of an error. The method also uses the feed forward equation as previously discussed above,

$$Y_n = W_n \bullet Y_{n-1}$$

This equation is a simplified version of the one discussed earlier but broken down simply means “The output at layer n is equal to the weights at layer n, dot produced with the input vector or output of vector at layer n-1” the output is also passed through an activation function.

Disclaimer for the next part of code to be discussed, the whole piece of code couldn't actually all be fitted in one screenshot. So i've zoomed out however one line stretched out just a bit too much, for reference it is over 310 characters long however that line is responsible for applying the optimiser to the updated weights and the adding them on to the new weights. I do apologise for the loss of quality to, for some reason Google Docs the software i'm using to write this removes almost a fair portion of quality. Sorry for this inconvenience I have tried to make the quality better but unfortunately I haven't been able to make it 100% cleare. But if your interested in the code go to the github and follow this file direct
"src/MachineLearningFolder/generateNetwork.java"

```
public void train(vector[] Tinputs, vector[] Toutputs, int epochs, int memoryInc) throws Exception{
    for(int epoch = 0; epoch < epochs; epoch++) {
        for(int setIndex = 0; setIndex < memoryInc; setIndex++) {
            //feeding untrained network data to train on (output tensor index:0 is initial input)
            this.outputTensor[0] = Tinputs[setIndex];
            for(int layer = 1; layer < this.weightTensor.length+1; layer++) {
                this.outputTensor[layer] = ((functions)this.func).activated(this.weightTensor[layer-1].matrixDot(this.weightTensor[layer-1], this.outputTensor[layer-1]));
            }
            for(int errorIndex = this.errorTensor.length-1; errorIndex>=0; errorIndex--) {
                if(errorIndex == this.outputTensor.length-1) {
                    this.errorTensor[errorIndex] = this.outputTensor[errorIndex].vectorSubtraction(Toutputs[setIndex], this.outputTensor[errorIndex]);
                    this.errorTensor[errorIndex] = this.errorTensor[errorIndex].elementWiseMultiplication(this.errorTensor[errorIndex], ((functions)(this.func)).derived(this.outputTensor[errorIndex]));
                }else {
                    this.errorTensor[errorIndex] = this.weightTensor[errorIndex].matrixDot(this.weightTensor[errorIndex].transpose(), this.errorTensor[errorIndex+1]);
                    this.errorTensor[errorIndex] = this.errorTensor[errorIndex].elementWiseMultiplication(this.errorTensor[errorIndex], ((functions)(this.func)).derived(this.outputTensor[errorIndex]));
                }
            }
            for(int deltaIndex = this.deltaTensor.length-1; deltaIndex >= 0; deltaIndex--) {
                this.deltaTensor[deltaIndex] = this.outputTensor[deltaIndex].vectorDotToMatrix(this.errorTensor[deltaIndex+1], this.outputTensor[deltaIndex]);
            }
            for(int weightPropigationIndex = 0; weightPropigationIndex < this.weightTensor.length; weightPropigationIndex++) {
                this.weightTensor[weightPropigationIndex] = this.weightTensor[weightPropigationIndex].matrixAddition(this.weightTensor[weightPropigationIndex], this.deltaTensor[weightPropigationIndex].s
            }
        }
    }
}
```

This method here (the one discussed above) is the most important method in the whole program, without the neural network would not be able to learn the method has an estimated time complexity of O(n^6)(ignoring any constants and just going off the highest estimate) a high time complexity; however that's a rough figure based on just the for-loops in the program (there are for-loops that cannot be seen as they are referenced in called methods). The breakdown of this function will be an explanation of each for loop and its purpose.

The first for loop is the same as our feedForward method, its job is to record the output of every layer in the network when the data is fed into it. This is essential as the output of each layer is required to calculate the error of the network. The next for loop takes the output tensor that was generated in the first for loop and is used to calculate the error of each layer in the network (refer to the backpropagation section to see how this is done). The next for loop is then used to calculate the delta weights; this part determines how much we should change the weights, the final for loop then applies the momentum optimiser to the delta weights and then finally applies the changes to the weights. The for loops then finally end, there are two versions of this class one is used if the training

data does not match the size of the array it is stored in. This allows for a simple method to allow the AI to train on data of varying size.

```
public void setupBellman(vector rewardVector, vector[] observation, vector[] action, vector actionRec, int memoryLength) {
    this.memoryLength = memoryLength;
    this.rewardVector = rewardVector;
    this.observation = observation;
    this.action = action;
    this.newActionSet = new vector[this.memoryLength];
    for(int i = 0; i < this.memoryLength; i++) {
        this.newActionSet[i] = new vector(this.action[0].getSize());
    }
    this.actionRec = actionRec;
}

public vector[] calculateBellman() throws Exception {

    for(int timeStep = 0; timeStep<this.memoryLength; timeStep++) {

        vector SARSAvec = new vector(2);
        SARSAvec = (this.action[timeStep+1].scale(this.action[timeStep+1], this.gamma));
        SARSAvec.setElement((int)this.actionRec.getElement(timeStep),(double)(SARSAvec.getElement((int)this.actionRec.getElement(timeStep))
                + this.rewardVector.getElement(timeStep)));
        this.newActionSet[timeStep].replaceVector(SARSAvec);
        SARSAvec.print(false);
    }
    Thread.sleep(25);   //prevents a buggy looking mess
    return this.newActionSet;
}
```

The above methods are crucial for creating training data for the AI to use, these functions work by first setting up the bellman equation parameters for examping setting up the observations, actions taken, rewards, and recording time steps… The method also takes in five parameters which are the reward vector, observation vector, action vector, actions recorded, and the number of time steps which is denoted as memory Length. The second method then takes this data and begins to calculate the temporal difference of the network at a given time step in the training data. The temporal difference is calculated as $Temporal\ Dif\ Target = r_t + \gamma \cdot maxQ(S_{t+1}, A)$.

That concludes the neural network design for more information on neural networks go to the github repository at where the source code will be located for the NEA https://github.com/Jake1402/Jake-Watson-College-NEA the github will also contain more files such as a class paradigm a copy of this document as well as further reading materials and a list of sources and citations used during the development period of this document.

Physics Simulations

```
private void update(int deltaTime) {

    double temp = (this.ForceCart + this.poleMassLength*Math.pow(this.pendulumVel, 2)*Math.sin(this.theta))/this.totalMass;

    double thetaacc = ((this.gravity * Math.sin(this.theta)) - (Math.cos(this.theta) * temp))/
            (this.pendulumLength* ( (4.0/3.0) - ((this.massPendulum * Math.pow(Math.cos(this.theta), 2)) / this.totalMass)));

    double accCart = temp - (this.poleMassLength*this.thetaacc*Math.cos(this.theta))/this.totalMass;

    this.xPos += (cartVel/((desiredDeltaLoop - deltaTime)/1000000));
    this.cartVel += accCart;
    this.cartVel*=this.dampening;

    this.pendulumVel += thetaacc;
    this.theta += this.pendulumVel/((desiredDeltaLoop - deltaTime)/1000000);
    this.pendulumVel*=this.dampening;

    int renderPos = (int) (this.xPos + (cartVel/((desiredDeltaLoop - deltaTime)/1000000))*150);

    cart = new Rectangle((int)(renderPos), (int)((HEIGHT/2)-cartH)-6, cartW, cartH);
}
```

The above images