# How Machines Learn

Florian Henryk Creutzig

March 2022

## 1 Introduction

How machines learn: The math behind neural networks and backpropagation
This text is purely for exploring the concept of machine learning and the mathematics behind neural networks. It is not a practical guide to practically preform machine learning. This text exists because I wanted to learn backpropagation, but found the Wikipedia article to be quite poor at explaining the math behind the concept. Therefore, I have read up on it and hope to provide a more clear explanation than Wikipedia provides.

## 2 Content

Content:

## 3 What is machine learning and what are neural networks?

What is machine learning? Machine learning is a technique to generate a numerical way to estimate unknown data based on known data. A machine learning model is a mathematical system that takes numbers in, called features, and return number(s), called predictions. There are many different types of machine learning, but here I will specifically discuss a type of machine learning model called an Artificial Neural Network (ANN), often simply called Neural Networks. While there are many different modifications and a lot of variations of NNs, here we will only discuss Ordinary Neural Networks (ONN), the most simple and plain flavor of NNs.

What are Neural Networks and how do they work? A neural network is modeled after the brain, a Biological Neural Network (BNN). In BNNs, biological neurons are connected to each other and send electrical signals of varying strength through these. NNs function similarly, though they are more organized. Electrical signal become numbers (still called signals) and the nodes are organized into different layers. The input layer, typically visualized as being on the left, is where the features are input. From there, signals are sent to the right, through connection edges in the network. In a dense layer every node is connected to all nodes in the following layer. Here, we will exclusively be discussing NNs with only dense layers. As the signals are sent through the network, they eventually reach the rightmost layer, the output layer, where the resulting signals are the prediction. All layers that are not the input or output layer are called hidden layers and are what makes deep learning deep.

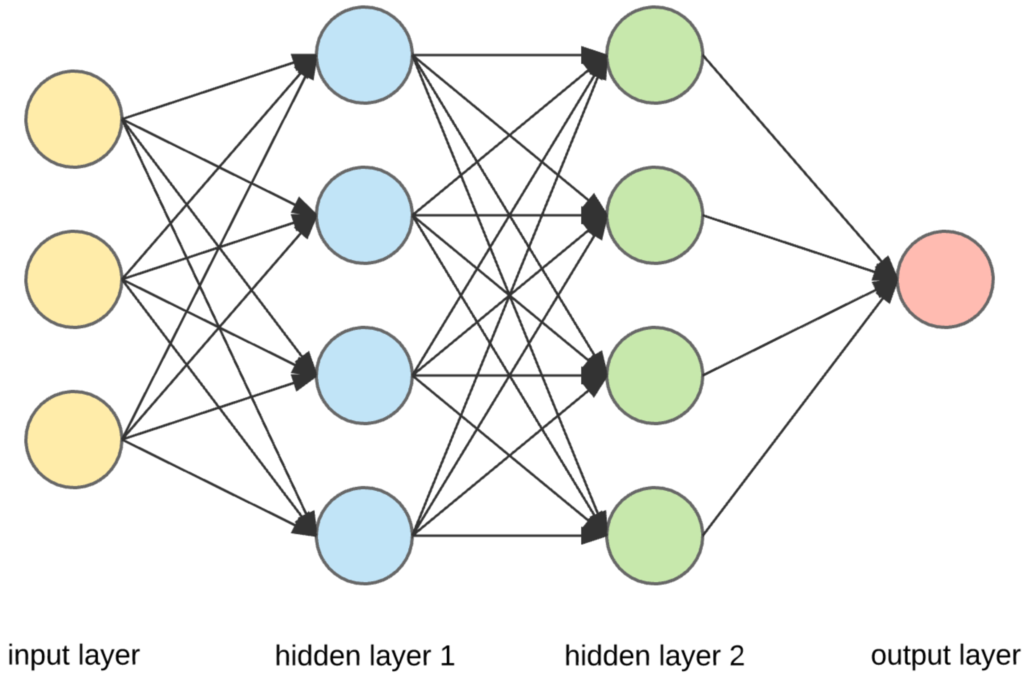


*Figure 1. A visual depiction of a simple neural network. Credit to: Towardsdatascience.com*

The interesting bit of the network is what happens when different signals are sent into a node. Each signal in each edge is assigned a strength value, based on a parameter (also often called a weight), i.e., the signal is multiplied by this parameter. After this, the sum of all incoming signals to a node is calculated and put through an activation function to get a signal which is passed on to the next layer. This function can differ based on what kind of neural network we are working with, though the activation functions for nodes in a layer are typically the same. The number we get after applying the activation function is called the activation of that node.

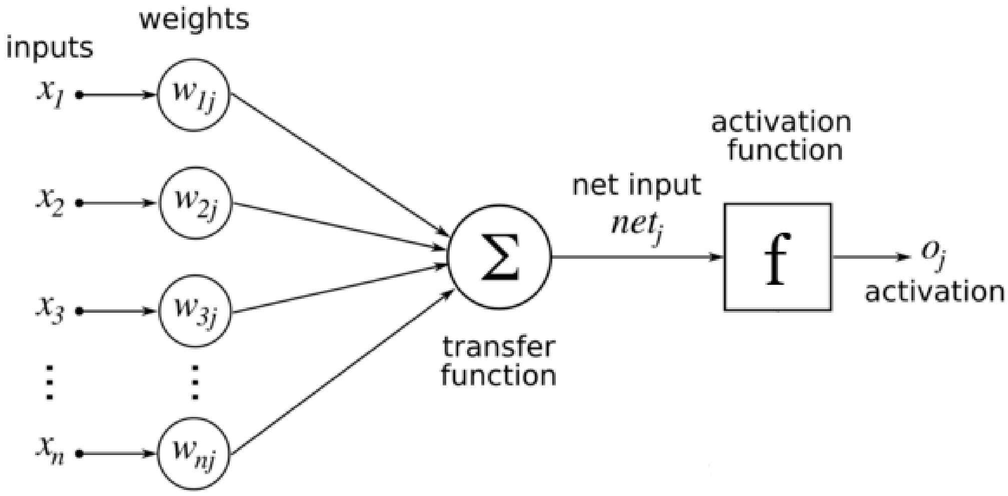


*Figure 2. A visual depiction of a single node in a neural network. Credit to Wikipedia*

This complex architecture let's the model "see" deep and complex connections in data and make complex predictions based on this. But how do we teach it to do that and predict something that is not total rubbish? As it stands right now, the model is just doing a lot of math in a very fancy manner without accomplishing much. This is where the parameters come in. By adjusting them based on previous observations, we can get a model that can accurately predict new outcomes. To do this, we need a lot of cases, where we know what the outcome should be and make the model try to replicate this using adjustment of the parameters. The first step of this is to get a numerical measure of how well the model can predict each datapoint. This can be done in a multitude of ways depending on the task at hand (like testing how many of the datapoints the model correctly labels, if the goal is labeling), but we will be looking at a cost function, measuring the distance between the models prediction and the correct values.

This cost function can be a simple absolute difference or some other function to measure distance, but for the purpose of this text, we do not care what function it is if it is differentiable in all pairs (x, y). It also does not matter what activation functions are being used, if they also are differentiable in all x.

# 4 Mathematical introduction and notation

## 4.1 Linear Algebra

While there may be many readers that are not familiar with fundamental linear algebra, this may not be essential for understanding back-propagation. However, some things should be noted. A vector of n dimensions is, for the purpose of this paper, a collection of n numbers in a fixed a order. A matrix of dimension m x n is similarly a collection of m vectors that themselves can be seen as n x 1 matrices. Multiplication two matrices is a slightly tricky business and it is important to remember that it is not commutative, i.e. $A \cdot B \neq B \cdot A$. While I do not want to explain how matrix multiplication works in this paper, you can read up on it on e.g.

`https://en.wikipedia.org/wiki/Matrix_multiplication`

One important thing about matrix multiplication I will cover here is how the dimensions work. A matrix with n rows and m columns, we call an n x m matrix. If we have $A \cdot B = C$, matrix multiplication is only defined if the inner dimensions match up, i.e. if A has dimensions n x m and B has dimensions r x k, then you can only multiply if m = r. The resulting matrix C will then have the outer dimensions: n x k.

Also, quickly, I would like to introduce the transpose. If you transpose a matrix, you swap the indexes. This can be thought of flipping the matrix along the diagonal and results in the dimensions being inverted. I.e. the transpose of a n x m matrix is a m x n matrix.

Furthermore, it is important to note that, for the most part, linear algebra is nice. This means that most everything that works with real numbers also works in higher dimensions. It also leads to results such as us having the ability to describe the multiplication and summation of weights in the neural network by a simple multiplication between a matrix and a vector: $net^k = W^k \cdot o^k$, where $net^k$ is the vector the net values in layer k, $W^k$ is the matrix of weights and $o^k$ is the vector of the activations in layer k.

Finally, I will briefly discuss tensors. A tensor of third degree can be thought of as a collection of matrices in a vector. We will need to use multiplication between matrices and tensors. If we multiply a n x m matrix with an l x m x k tensor, you can view it as multiplying the matrix with the transpose each of the k matrices in the tensor. The dimensions for each matrix are then n x m · m x l = n x l. The final tensor then becomes n x l. If you actually want to compute this (and matrix products), I recommend just using numpy.dot in pyhton.

## 4.2 Derivation and complete derivatives

There are a lot of derivatives that need computing in this paper. So, we should get comfortable with the computation of complicated derivatives.

First of all, a derivative of a function f with respect to a variable $x_i$ is denoted by $\frac{\partial f}{\partial x_i}$. This means that only $x_i$ is seen as a variable to differentiate with, while all other variables are seen as constants.

If we have a function that takes inn an n dimensional vector and returns an m dimensional vector, the total derivative is a function, represented by a matrix of m x n dimensions where each entry is the derivative of one output with respect to one input. It will be denoted by $\frac{\partial f}{\partial x}$ where x is a vector and f is a fitting function. If we have a matrix (m x k) as the input and a vector of dimension n, the total derivative is a tensor of n x m x k. The chain rule surprisingly still works with total derivatives, i.e. $\frac{\partial f(g(x))}{\partial x}(x) = \frac{\partial f}{\partial g(x)}(g(x)) \cdot \frac{\partial g}{\partial x}(x)$

To evaluate a function f at a given input x, we denote f(x), no matter what the dimensions of x or f(x) are. This may seem trivial, but it is important to remember for more complicated expressions like $\frac{\partial f}{\partial x}(x)$.

## 4.3 Notation

In this paper, we will further denote the following: L will be an integer denoting the total number of layers in a neural network. We will also denote the number of nodes in layer k as n(k). Further, we will denote the activation function at layer k as $f^k$, which is a function going from a n(k)-dimensional vector to a n(k) dimensional vector. For the most part, this will be the same function applied to every element of the vector. The parameters (weights) going from layer k to layer k+1 are represented by a matrix, $W^k$, with dimensions n(k+1) x n(k). Furthermore, the activations for layer k are called $c^k$, as are the net values $net^k$. I will note that if I ever use $o^k$ without specifying and input, I intend to evaluate by x. With all this and recalling that we can apply weights by multiplying with the relevant matrix, we can represent the output of our model as follows:

$$g(x) := f^L(W^{L-1} \cdot f^{L-1}(W^{L-2} \ldots f^2(W^1 \cdot x) \ldots))$$

We can further simplify by defining the entire transformation between layer k and k+1 as $o^{k+1} = \Phi^{k+1}(o^k) = f^{k+1}(W^k \cdot o^k)$, which makes the model:

$$g(x) := \Phi^L(\Phi^{L-1}(...\Phi^2(x)...))$$

And finally, the cost function is simply denoted as: $C(y_i, g(x_i))$, for some input $x_i$ and correct value $y_i$

All right, that was a lot setup of notation and math. Let's actually use it.

# 5 Backpropagation

Generally, backpropagation works by expanding the concept of gradient descent. Gradient descent is an algorithm which finds optimized values in multidimensional space using derivatives. The idea behind gradient descent is that if we can compute the derivative of the cost function with respect to each parameter individually, we could use all these partial derivatives to find the gradient. The gradient would be based on the direction (positive or negative) and magnitude of the derivative and would allow us to find a fitting change to the parameters that reduces the error function. If adjust the parameters accordingly and repeat this a few times, we could get close to the optimal parameters for our model. The question then becomes: how do we find this derivative? If we use the chain rule twice, we can find the following

$$\frac{\partial C}{\partial W^{k-1}}(y_i, g(x_i)) = \frac{\partial C}{\partial o^k}(y_i, g(x_i))\frac{\partial o^k}{\partial W^{k-1}}(W^{k-1}) = \frac{\partial C}{\partial o^k}(y_i, g(x_i))\frac{\partial o^k}{\partial net^k}(net^k)\frac{\partial net^k}{\partial W^{k-1}}(W^{k-1})$$

We will have to find all these three partial derivatives. Firstly, we have:

$$\frac{\partial net^k}{\partial W^{k-1}}(W^{k-1}) = \frac{\partial W^{k-1} \cdot o^{k-1}}{\partial W^{k-1}}(W^{k-1}) = o^{k-1}$$

(This is actually slightly wrong, but we will get to that)

Secondly, we get:

$$\frac{\partial o^k}{\partial net^k}(net^k) = \frac{\partial f^k(net^k)}{\partial net^k}(net^k)$$

Which is the total derivative of the activation function. Lastly, we have the hardest term to deviate:

$$\frac{\partial C}{\partial o^k}(y_i, g(x_i))$$

This is easy to evaluate if the current layer is the output, i.e. k = L; we simply take the derivative of our cost function with respect to the predicted y-value i.e. g(x). But this quickly gets tricky as we get further left in the network.

To find this derivative, first we use the chain rule to find:

$$\frac{\partial C}{\partial o^k}(y_i, g(x_i)) = \frac{\partial C}{\partial g(x)}(y_i, g(x_i)) \cdot \frac{\partial g}{\partial o^k}(x_i)$$

While the first term is easy to find, we will have more trouble with the second term. Let's break that down:

$$\begin{aligned} o^k(x_i) &= \Phi^k(...\Phi^3(\Phi^2(x_i))...) \\ g(x_i) &:= \Phi^{L-1}(...\Phi^{k+1}(o^k)...) \end{aligned}$$

We can use the chain rule on the latter term (in total L-k times) to get:

$$\frac{\partial g}{\partial o^k}(x_i) = (\Phi^L)'(\Phi^{L-1}(...\Phi^{k+1}(o^k)...)) \cdot (\Phi^{L-1})'(\Phi^{L-2}(...\Phi^{k+1}(o^k)...)) \cdot ... \cdot (\Phi^{k+1})'(o^k) \cdot \frac{\partial o^k}{\partial o^k}(x_i)$$

Which we can (thankfully) simplify to:

$$\frac{\partial g}{\partial o^k}(x_i) = (\Phi^L)'(o^{L-1}) \cdot (\Phi^{L-1})'(o^{L-2}) \cdot ... \cdot (\Phi^{k+1})'(o^k) \cdot 1$$

$$\frac{\partial g}{\partial o^k}(x_i) = \prod_{l=k}^{L-1} (\Phi^{l+1})'(o^l)$$

Here, it is important to recall that matrix multiplication is not commutative, and you will need to preserve the order of multiplication shown in the former equation.

Before we begin to evaluate these terms, it is also important to remember that the terms themselves don't depend on k. This means that we can compute all these terms once and that will be sufficient for use for all k, saving computing time.
As for the evaluation, remember that:

$$\Phi^{l+1}(o^l) = f^{l+1}(W^l \cdot o^l)$$

And therefore:

$$(\Phi^{l+1})'(o^l) = \frac{\partial \Phi^{l+1}}{\partial o^l}(o^l) = \frac{\partial f^{l+1}}{\partial net^{l+1}}(net^{l+1}) \cdot W^l$$

The first term is simply the derivative of the activation function $(f^l)'(net^l)$, remembering that $net^{l+1} = W^l \cdot o^l$. To summarize:

$$\frac{\partial C}{\partial o^k}(y_i, g(x_i)) = \frac{\partial C}{\partial g(x)}(y_i, g(x_i)) \cdot \prod_{l=k}^{L-1} (f^{l+1})'(W^l \cdot o^l) \cdot W^l$$

And finally, the full equation:

$$\frac{\partial C}{\partial W^{k-1}}(y_i, g(x_i)) = \frac{\partial C}{\partial g(x_i)}(y_i, g(x_i)) \cdot \left( \prod_{l=k}^{L-1} (f^{l+1})'(W^l \cdot o^l) \cdot W^l \right) \cdot (f^k)'(net^k) \cdot o^{k-1}$$

If you have been paying extremely close attention (I don't blame you if you have not), you will notice that while all the matrix multiplications work out: $\frac{\partial C}{\partial g(x_i)}(y_i, g(x_i))$ is 1 x n(L), $\prod_{l=k}^{L-1}(f^{l+1})'(W^l \cdot o^l) \cdot W^l$ works out to n(L) x n(k), $(f^k)'(net^k)$ becomes n(k) x n(k) and $o^{k-1}$ is n(k-1) x 1, after preforming all the multiplications one of the dimensions does not work out! The multiplication to the right works ends up being 1 x n(k) $\cdot$ n(k-1) x 1. Not only does this not work out, by what we have established earlier about total derivatives, $\frac{\partial C}{\partial W^{k-1}}(y, x)$ should be a tensor of dimensions 1 x n(k) x n(k-1).

The missing link is the term $\frac{\partial net^k}{\partial W^{k-1}}$. By our criteria for total differentiation, this should be a tensor of dimensions n(k) x n(k) x n(k-1), not the n(k) x 1 dimensional $o^k$. This tensor turns out to be simply an issue of dimension, with the total derivative yielding the n(k) x n(k) x n(k-1) we are looking for. This is a horrible way to store data, but here, the tensor operations come in. We multiply using this wired tensor, and the dimensions work out! We have n(k) x n(k) $\cdot$ n(k) x n(k) x n(k-1) = n(k) x n(k) x n(k-1) by tensor multiplication. Furthermore, n(L) x n(k) $\cdot$ n(k) x n(k) x n(k-1) = n(k) x n(L) x n(k-1) and finally, 1 x n(L) $\cdot$ n(k) x n(L) x n(k-1) = 1 x n(k) x n(k-1), which finally is a tensor of correct dimensions!

As you can see, if we use this algorithm in practice, we need to first compute all the activations $o^k$ in what is called the "forward pass" before we can go back and calculate these derivatives.

While there are multiple algorithms that use this gradient to perform machine learning, we will be coming back to the simplest one: gradient descent. Gradient descent takes all training data and repeats this process with all the training data to get an average derivative for every weight and use this to compute the change to the weights by the learning rate $\alpha$, i.e. how fast the network adapts to changes. This is machine learning!

$$W^k = W^k - \alpha \frac{\partial C}{\partial W^k}(y_i, g(x_i))$$

## 6 Neurovevolution

Now, after all that math, you might not see any beauty in machine learning. After all, we have just seen how adjusting the weights is a fairly boring process, involving a lot of chain rules, derivatives and number-crunching. There is none of what you might find in a typical demonstration of machine learning: many generations of machines failing endlessly at a task, but selecting and mutating to find a better solution. This is also an important concept in machine learning called artificial evolution. While backpropagation and its corresponding networks are a form of supervised learning, where you have a dataset of inputs and corresponding correct values, artificial evolution needs no data. It is a system where different algorithms are tested in environments, such as playing simple video games.

One of the more common forms of artificial evolution is neuroevolution (NE). In NE the brains of each individual is an artificial neural network. The adjusting of weights however, works in quite a different manner. The algorithm of NEs works in three separate steps, modeling after biological evolution. The fundamental steps of artificial evolution are as follows: selection, crossover and mutation. These steps are used to evaluate the models and create a new generation that tires to further improve on the best models.

### 6.1 Selection

In artificial evolution and more specifically NE, instead of improving a single neural network, we have a lot of different ones, called a population of neural nets consisting of individuals. In selection, a population is tested by how well it can perform the task at hand. Here, we see neurons playing Mario or some other game. After this, we can get the best individuals and use them for crossover.

### 6.2 Crossover

In crossover, we will let the genes of different individuals, that is, the parameters of the neural nets, be exchanged in random ways to create new offspring. Note that depending on the kind of neuroevolution this step may actually be counterproductive. This is because we can get parameters from two individuals crossing, that succeed by using two different approaching and forcing these to combine can lead to little more than a mess. However, if we use and approach such as NEAT, this step is modified somewhat and becomes essential.

### 6.3 Mutation

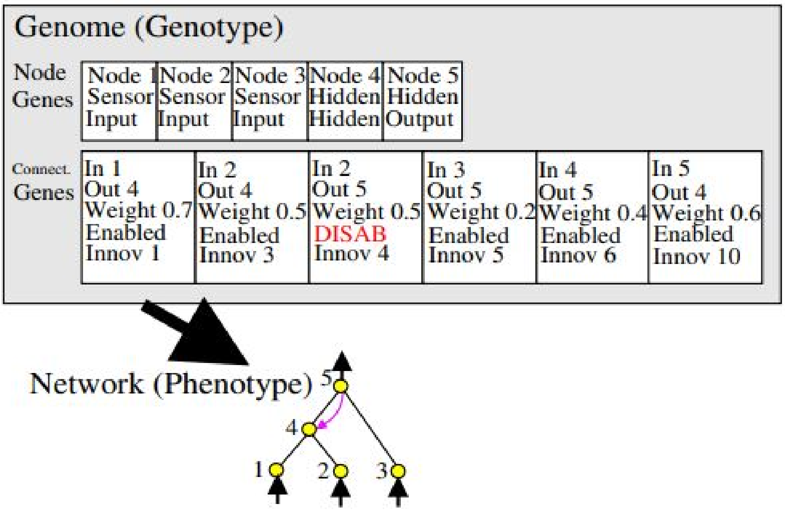
Finally, the new individuals that have been created, need to have a few mutations. This is to create a varied gene pool and to create the possibilities of pushing the individuals in the right direction.

## 7 Augmenting Topologies and NEAT

Enter NEAT. NEAT stands for neuroevolution of augmenting topologies. With NEAT, the neural network is generated as the program runs. Nodes are added, removed connections are dynamically changed. With these more fluid dynamics of NEATs neural networks, the network can no longer be connected into layers. Instead, it starts with just the input and output layers, but not set up as such. Then, connections are added between nodes and nodes are added in the middle of connections as the networks evolve.

### 7.1 Encoding the models

In NEAT, the model must have a structure that is easy to augment. Therefore, it is only the connections of different nodes in the system, with their corresponding weights that are saved. The way to evaluate the network is to preform time steps. First, the input nodes are activated with the given input. Then, the nodes connected to these are activated, and so on until the output nodes are activated, where all the outputs are accumulated. Note that some nodes may get activated multiple times, some in constant cycles. To prevent infinite cycles, we will limit the number of steps that the network can perform. When a node is activated again, it overrides the current activation.



| | | | | |
|---|---|---|---|---|
| Node 1<br>Sensor<br>Input | Node 2<br>Sensor<br>Input | Node 3<br>Sensor<br>Input | Node 4<br>Hidden<br>Hidden | Node 5<br>Hidden<br>Output |

| | | | | | |
|---|---|---|---|---|---|
| In 1<br>Out 4<br>Weight 0.7<br>Enabled<br>Innov 1 | In 2<br>Out 4<br>Weight 0.5<br>Enabled<br>Innov 3 | In 2<br>Out 5<br>Weight 0.5<br>DISAB<br>Innov 4 | In 3<br>Out 5<br>Weight 0.2<br>Enabled<br>Innov 5 | In 4<br>Out 5<br>Weight 0.4<br>Enabled<br>Innov 6 | In 5<br>Out 4<br>Weight 0.6<br>Enabled<br>Innov 10 |

*Figure 3. A visual depiction how a network in NEAT is represented. Credit to the original NEAT paper (Stanley and Miikkulainen) for the figure and for subsequent figures about NEAT*

### 7.2 Selection and Speciation

In the selection process, NEAT adds an extra step. Here, NEAT further takes inspiration from nature. In order to preserve multiple different approaches to solve a problem, models are divided into species based on how similar they are. Then, models will compete not globally, but withing their species. This ensures that we get the best solutions from each approach, even approaches that take a long time to converge to a good solution are preserved.

### 7.3 Crossover

Crossover in NEAT works quite curiously. Every connection is assigned an innovation number, when created. If the connection between these two nodes already exists, it retains the innovation number given earlier to that connection. What we end up with is an innovation number corresponding to each connection. During crossover, we lay the innovations out in a line, and we randomly choose the connection of one of the parents for that individual (given the connection exists and is active). If only one parent has the relevant connection, this gets added if the parent in question has a better score than the other parent, otherwise it is left unchanged. Alternatively, you could just use the connection from whoever has it, given only one parent has it (used in figure 4). The nodes of the new network are determined by what edges it inherits. All nodes that the edges touch, are in the new network.

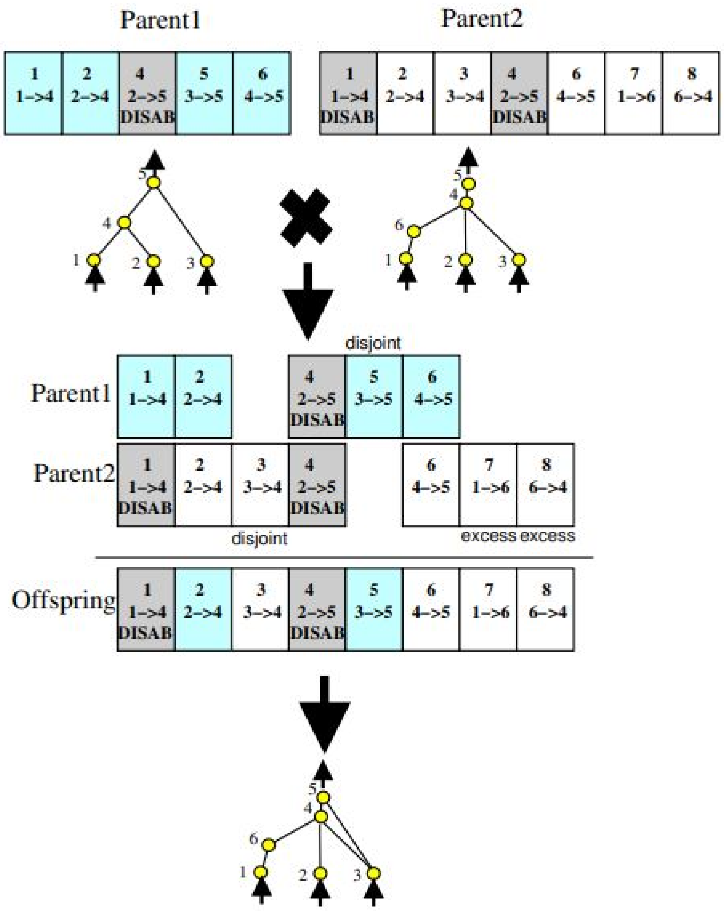


*Figure 4. An illustration of two NEAT networks crossing over to create one individual of the new generation.*

## 7.4 Mutation

The mutations in NEAT also come in a bunch of different varieties. Firstly, we have normal weight adjustment, like in regular neuroevolution. Here, we have to adjust the weights by multiplying with a random number between 0 and 2, or make completely new weights. This is a standard setup, but the exact way to implement weight shifting can vary and I encourage experimentation.

In addition to this, NEAT also adds a couple new mutations for changing the topology of the network. First of all, you can add a new node in the middle of an existing edge. This will create two new edges and the old edge should be disabled. Furthermore, another mutation is flipping if a particular edge is disabled, i.e. turning it on if it is off and vice verca. Finally, you can mutate to add a new connection between two nodes that don't currently have an edge.

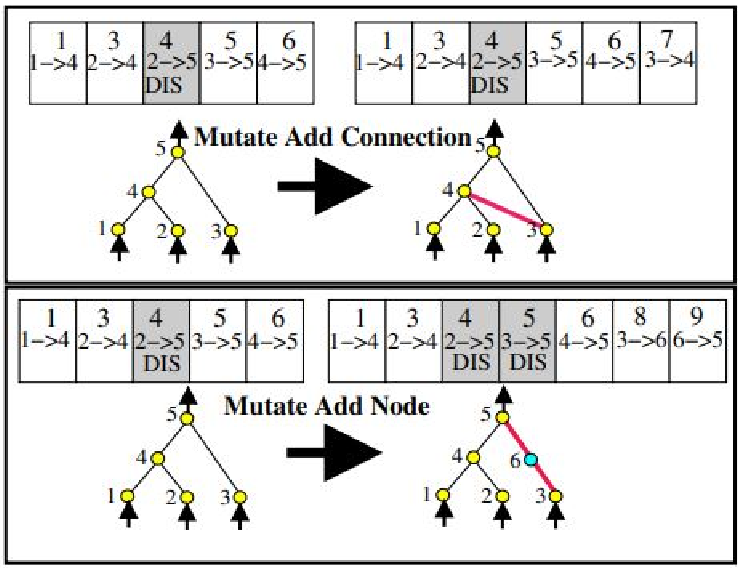


*Figure 5. An illustration of the "add connection" and "add node" mutations of NEAT*

# 8 Appendix A: Sources

Wikipedia contributors. (2022, March 11). Backpropagation. In Wikipedia, The Free Encyclopedia. Retrieved March 24, 2022, from https://en.wikipedia.org/wiki/Backpropagation

Stanley, Miikkulainen (n.d.) Efficient Evolution of Neural Network Topologies From university of Texas at Austin. Retived March 24, 2022, from http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf

https://towardsdatascience.com/a-primer-on-the-fundamental-concepts-of-neuroevolution-9068f532

https://en.wikipedia.org/wiki/Matrix_multiplication

https://www.youtube.com/watch?v=b3D8jPmcw-g

Image of a neural nett: https://towardsdatascience.com/applied-deep-learning-part-1 -artificial-neural-networks-d7834f67a4f6?gi=8851a35ea627