The Usability of Neural Network Logical Operators in Security Login Systems

(Soon to be) Dr David Mills[*]
Department of AI
Utlandia
`davidmills@GG.ut`

Abstract

In the current age of AI and upcoming AGI, it is important to follow the current. Therefore, I, the administrator and up-and-coming Head of AI, have implemented a login supported by Neural Network logical operators. Instead of using the simple AND/OR rules for the login, the network is trained to mirror this behavior. Here is a short paper explaining the basics of the functionality, so that you, the company's bosses and employees, can be impressed by my knowledge.

1 Neural Networks

Inspired by the human brain, artificial neural networks are implemented similarly, consisting of a network of nodes with connections between nodes, called weights. These networks are essentially function approximations, being able to learn and map a given input to a given output by carefully adjusting the weights between the nodes of the network.

1.1 Structure

A typical neural network consists of an **input layer**, followed by **hidden layers**, ending with an **output layer**. The input layer is where you, obviously, input the data that you want to result in a given output. This layer is typically created so that each node in the layer, referred to as input nodes, is mapped to a single input feature. An input feature is simply a single value of the input. The input filters are then paired up with a desired output feature/target, so if you want to predict age from height and weight, the input features are the height and weight, while the target feature is the age. Formally, one often refers to the input features and target values as input X, and target y.

The hidden layers are weight connections created to save the information needed to fully map the input to the output, and this can vary in size. The different layers and connections are illustrated in Figure 1.

To update the weights of the network (train), one relies on two basic algorithms. The first is the forward pass, followed by the backpropagation.

1.2 Forward Pass

As the name suggests, when forward passing in a neural network, you input the input features and pass them through the different layers of the network until you reach the end and are left with the output. The math behind the forward pass is quite simple. An example of a forward pass of a single-layered neural network is shown below. The X is the input features, w^T are the weights transformed, and

[*]Goood Games.

Preprint. Under review.

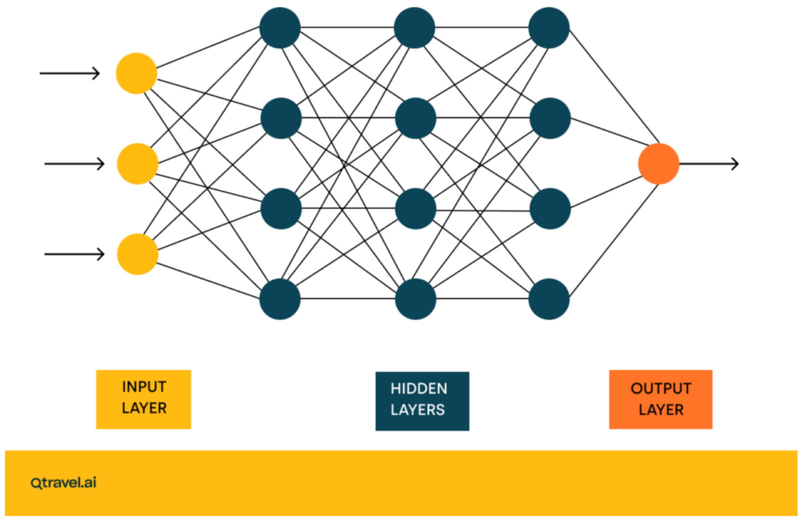


Figure 1: Visual graphic of a neural network. Source `https://www.qtravel.ai/blog/what-are-neural-networks-and-what-are-their-applications/`

lastly, the b is the bias. (You don't have to worry about the math of it all, but if interested, I suggest you look up matrix multiplication.)

$$\hat{y} = X \cdot w^T + b \tag{1}$$

1.3 Backpropagation

Backpropagation is the art of adjusting the weights so that the results of the forward pass become closer to the target output. This is done by computing the derivatives, more specifically the derivative (gradients) of the **loss** (see later sections) with respect to the weights. This tells us how much we should move the weights in order to minimize the loss, which again is a metric of how well the output of the network is. The author of this paper will not bore the reader with extensive mathematics, as this is beyond the scope of the common software engineer and administrator, and for fear of losing my job. Do be aware that this is not simple high school derivatives (hehe joking, it's actually the very same math techniques). To make this efficient with multiple nodes, some matrices are introduced, but again, this is outside your scope.

1.4 Activation Functions

To ensure that the neural networks can learn non-linearity (a big math word that means they can learn more approximations), the introduction of activation functions, or layers, has proven quite handy. These layers are typically placed between the different layers in the network, and at the very end, to give the output certain properties. The following text will list and explain the top 4 activation functions.

1.4.1 Sigmoid

The sigmoid activation function (Figure 2a) is nifty in terms of limiting the output values, y, of the function to a range of $y \in [0, 1]$. This is especially useful at the end of a neural network trained for binary classification, deciding if the input is a cat (0) or a dog (1).

1.4.2 Tanh

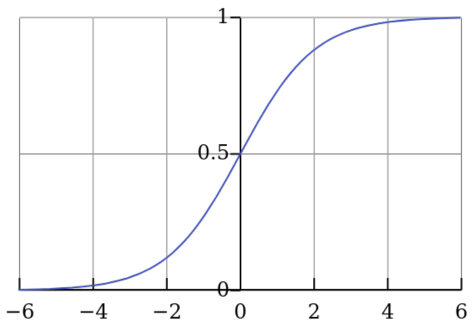
Similarly to the sigmoid function, tanh limits the output y to a range of $y \in [-1, 1]$. This can be seen in Figure 2b.

1.4.3 ReLU

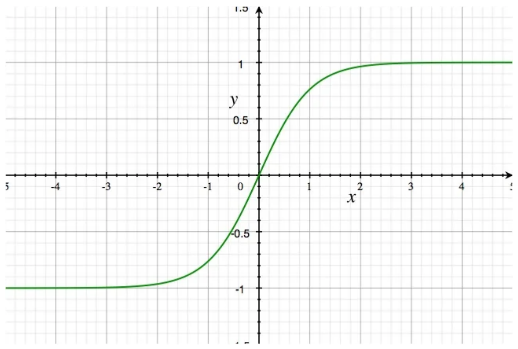
The simplest and (believe it or not) proven to be the most efficient and useful of them all, the ReLU activation function clamps negative input to zero, and leaves the positive values alone. It's beyond the author how in the world this is so good at making non-linearity and great neural networks, maybe even outside of God's knowledge as well. The function can be seen in Figure 2c.

1.4.4 Softmax

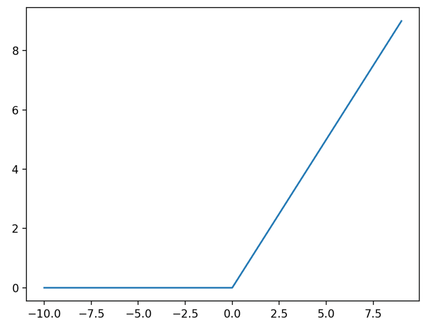
The very last activation function the author will name-drop is the Softmax function. This function maps the input to a probability distribution based on their original values relative to each other. This is especially good when having multiple classifications, where the network can output a probability distribution of the most likely class, say 10% cat, 20% dog, and 70% turtle. An illustration of the activation function can be seen in Figure 2d.



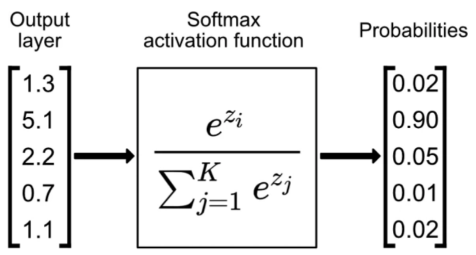
(a) Sigmoid Activation Function ($y \in [0, 1]$). *(Source: `https://en.wikipedia.org/wiki/Sigmoid_function`)*



(b) Tanh Activation Function ($y \in [-1, 1]$). *(Source: `https://www.ml-science.com/tanh-activation-function`)*



(c) ReLU Activation Function. *(Source: `https://sebastianraschka.com/faq/docs/relu-derivative.html`)*


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

(d) Softmax Activation Function. *(Source: `https://www.singlestore.com/blog/a-guide-to-softmax-activation-function/`)*

Figure 2: Comparison of common Activation Functions used in neural networks.

1.5 Training

Now that you have a feeling of how one can pass values through the network using the forward pass, and how to backpropagate and evaluate the adjustments needed to make a better approximation, one can finally continue with more practical information regarding the training of the network.

1.5.1 Loss

The loss, or loss function, is a famous expression used in the field of machine learning. The loss function is a quantification of how well the output of the network is compared to the target value(s), and is used for different purposes. If you are doing a simple regression task (predict age based on height and weight), the use of **MSE** (Mean Squared Error), or **RMSE** (Root Mean Square Error) is used. If one were to make a binary classification network, one often opts for **BCE** (Binary Cross Entropy), and for multiple classifications one uses the regular **CE** (Cross Entropy).

A perfectly trained model will be able to output a value such that the loss becomes 0. This is almost never possible, especially for more complex tasks, but the main objective is still (most commonly) to make the loss as close to 0 as possible.

1.5.2 Learning Rate

When finding the amount of adjustments to make to a neural network's weights during training, the **learning rate** α or η) is a critical hyperparameter that dictates the **step size** taken in the direction of the loss function's negative gradient.

It controls how quickly the model adapts to the data. If the step is too large, the training can become unstable; if the step is too small, convergence will be excessively slow.

The weight update rule in gradient descent incorporates the learning rate as follows:

$$W_{\text{new}} = W_{\text{old}} - \alpha \cdot \frac{\partial L}{\partial W}$$

where $\frac{\partial L}{\partial W}$ is the gradient of the loss function L with respect to the weight W.

1.5.3 Train - Val - Test split

A model is deployed to classify dogs against cats, but is suddenly faced with an orange cat, which it has never seen before. It fails miserably. This is a typical occurrence, and while hard to train a model not data never been seen, one can get a fairer view of how well it is performing by validating and testing the model on data it has never seen before. This is where the train-val-test split comes in. The typical approach is to split all available data into two splits, one split with 70-90% (depending on how much total data one has available) of the data into a training split and the rest into a test split. The test data is used after training, where the loss is evaluated, and gives a more nuanced image of how well the model is actually performing on data it has never trained on. One can also split the training data into a 90-10 split, where the 10% is used for validation between each training epoch. This gives a good estimate of generalization during training.

1.5.4 Batches and Optimizers

Only as a side note for the most curious readers. Training is often done in batches rather than single samples because batches give more stable gradient estimates, allowing efficient more efficient training. Common optimizers include **SGD**, **Adam**, **RMSprop**, and **Adagrad**. (If ever unsure what optimizer to use, opt for Adam, as this is the golden standard, even though most AI engineers have no idea why this is the case.)

2 Logic Operators

Logic operators are probably known to most, but for the ones who might work in other areas (such as HR), logic operators are combinations of two logical components, True and False, and a set of what combinations of these should output. The classic examples are shown in Table 1, and Table 2, displaying the AND, and (lol) OR operators.

3 Neural Network as Logical Operator

The following text explains how the me, the author, trained the different networks to behave as logical operators. The AND and OR network consists of two input weights, outputting to a single layer. A

Table 1: AND Gate Truth Table

Input 1	Input 2	Output
True	True	True
True	False	False
False	True	False
False	False	False

Table 2: OR Gate Truth Table

Input 1	Input 2	Output
True	True	True
True	False	True
False	True	True
False	False	False

sigmoid activation function is put at the end, to ensure that the logical value is between 0 and 1 (False or True), and can then be clamped to the nearest value. For the XOR operator, a network with 2 layers is used, as XOR is not a linear operator, and needs to lines to divide the True and False output. This will be visible in the figures. The XOR network also uses sigmoid as the activation function in the output layer, and does not use a activation function between the input and output layer.

The networks are trained over 2000 epochs, with the BCE loss function.

3.1 AND

The and operator, simple in nature. If the input values are something else than 1 and 1 (True, True), then return False, else return True. Below are some figures displaying the training, and other stats that ensure the network is perfect and up to speed, ready to be: git add, git commit, git push, git gtfo.

Figure 3 shows the network training, where the x and y axes are the input values to the network, and the blue dots represent the target output of 0 (False), and the red the target output of 1 (True). The red or blue in the background represents the model output for the given x and y input. Figure 4 shows the training information of the network, with the regression boundaries, the BCE loss progress, and even how the weights of the adjusted during training.

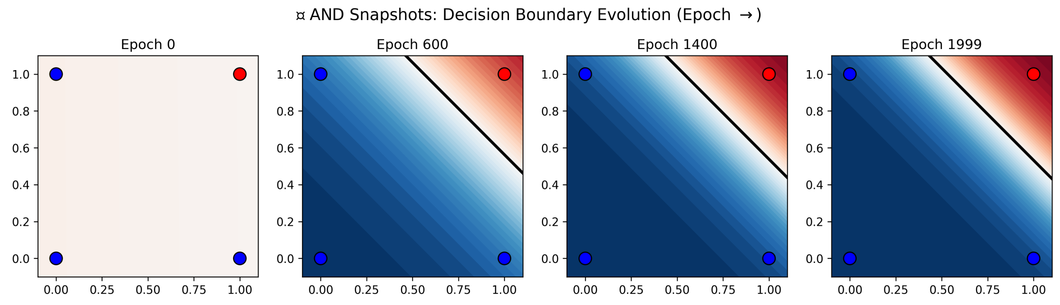
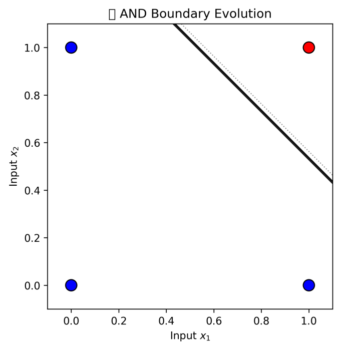
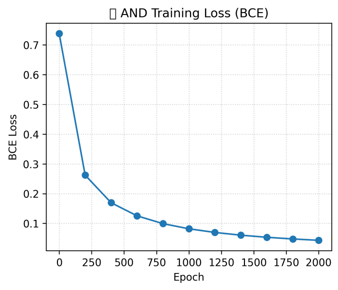


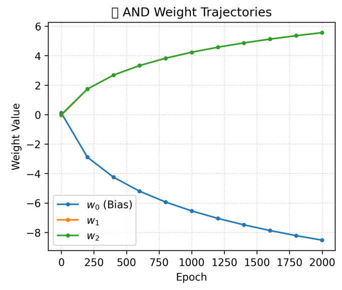
Figure 3: Regression line progressing during training.



(a) Regression Boundary for AND



(b) AND BCE loss



(c) Weights for AND network

Figure 4: Analysis plots for the AND logic gate network training.

3.2 OR

Next comes the OR operator with similarities to the AND network. Figure 5 shows the network training, where the x and y axes are the input values to the network, and the blue dots represent the target output of 0 (False), and the red the target output of 1 (True). The red or blue in the background represents the model output for the given x and y input. Figure 6 shows the training information of the network, with the regression boundaries, the BCE loss progress, and even how the weights of the adjusted during training.

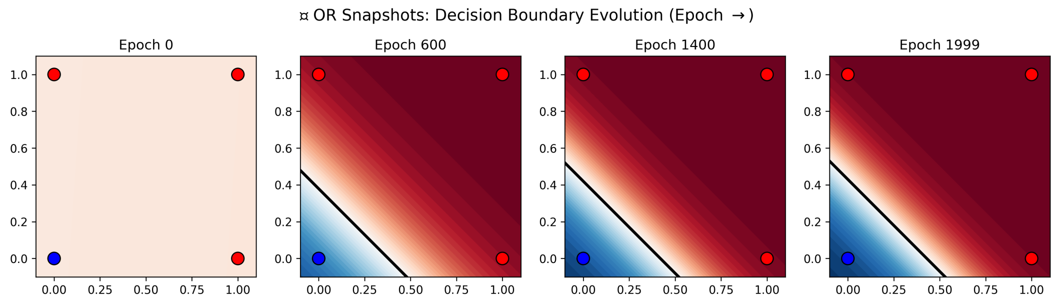
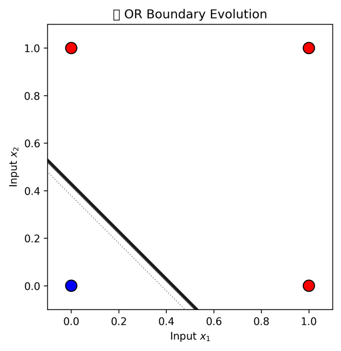
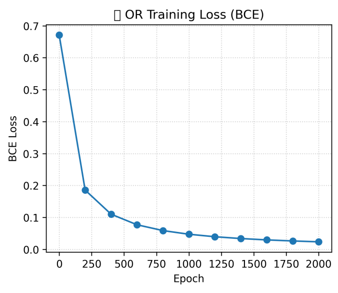


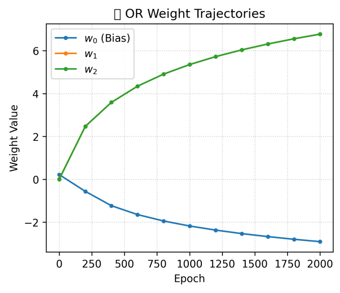
Figure 5: Regression line progressing during training.



(a) Regression Boundary for OR



(b) OR BCE loss



(c) Weights for OR network

Figure 6: Analysis plots for the OR logic gate network training.

3.3 XOR

Lastly comes the XOR, which is a bit more special as one can't split the True and False target outputs with a single line. Therefore a second layer is needed in the network, resulting in a second regression line, which makes it possible to output the correct value for XOR.

Figure 7 shows the network training, where the x and y axes are the input values to the network, and the blue dots represent the target output of 0 (False), and the red the target output of 1 (True). The red or blue in the background represents the model output for the given x and y input. Figure 8 shows the training information of the network, with the regression boundaries, the BCE loss progress, and even how the weights of the adjusted during training. Note that this model needed additional epochs during training.

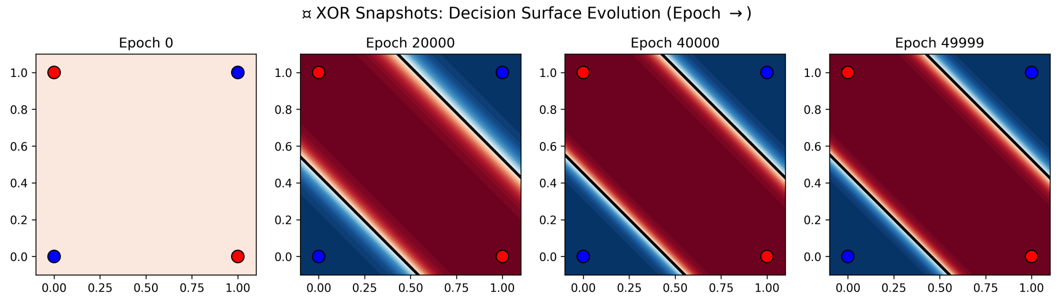
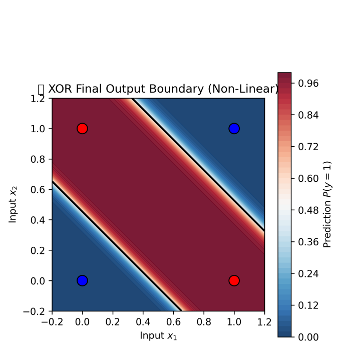
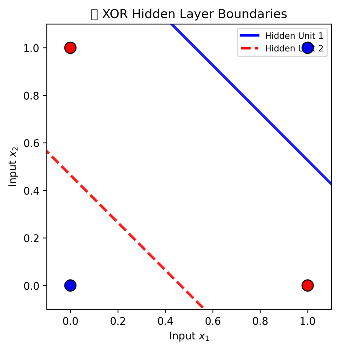
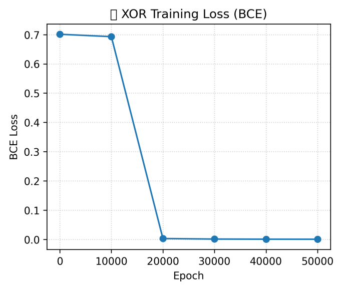


Figure 7: Regression lines progressing during training.







(a) Final Output Surface/Boundary (b) Hidden Unit Boundaries (c) XOR BCE loss

Figure 8: Core analysis plots for the XOR MLP network training, highlighting the final non-linear boundary and the linear boundaries created by the hidden units.

4 Practical Training of the Network

The following text gives information about the endpoints that were used to train the AND operator. These **should** be deactivated in production if DevOps is competent.

4.1 /Train/step

The network was trained through the `/train/step` endpoint, which takes a single training step by posting data with the following JSON structure:

```
{
    "x1": [x1],
    "x2": [x2],
    "y": [target]
}
```

The endpoint returns the loss of the training step, using Binary CE loss. I noted that when the loss was low enough (around 0.008), there was no point in continuing the training.

4.2 /Reset

I also implemented a `/reset` endpoint in cases where the model weights were not savable. A simple POST to this endpoint resets it to the original, so I could experiment with different input and target values.

5 Conclusion

Now that you are finally up to speed with elite AI knowledge, you can comprehend the reason for removing the old, bad use of logical operators in code, instead of implementing my very own logical operator networks. The plots never lie, the networks are perfect, and an important step towards the AI-fication of the Goood Games workplace.