# Neural Networks

## Introduction

Neural Networks are powerful tools that can be used to classify data into groups, predict continuous value outputs, generate images, and much more. By training the network on examples where it can see the answers, it can learn to transform input data in the correct way to give the right answer for examples it hasn't seen before. To calculate an output from a given input, information is fed forward through the network, with each layer calculated from values in the previous one.

Modeled upon brain structure, the network is made of nodes (values or "activations" between 0 and 1). These are structured into three or more layers, an input layer, a chosen number of hidden layers, and an output layer. In addition to the activations in each layer, there's also a bias which equals one, the purpose of which is to give the network extra "nobs and dials" to change, such that it can provide more accurate results.

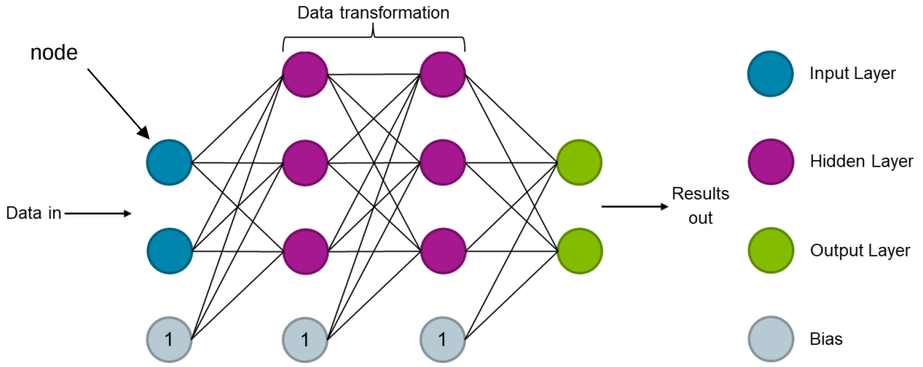


Fig. 1

## Inputs and Outputs

The inputs and outputs of a neural network are numerical, and represented in vector form. An example of a classification may make this easier to understand.

A classic neural network classification problem is that of handwritten letters. Here, the input is a column vector of brightness values for each pixel (for a 20x20 image the input layer would have 400 nodes), and the output is a column vector of length 10, where each node represents a digit. The value for each node in the output vector represents the network's confidence that the image is of that digit, so the digit corresponding to the node with the highest value is the networks prediction.

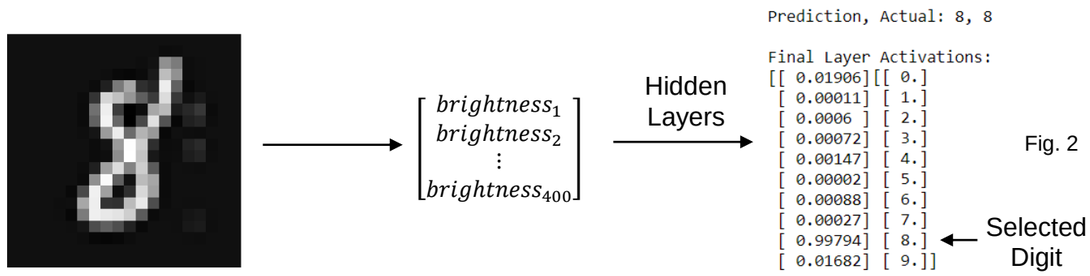


Fig. 2

## Intuition

Now that we understand the structure of inputs and outputs, we can focus on what goes on between them in the hidden layers.

Neural networks consist of activations whose values are determined by all the activations in the previous layer. What gives neural networks so much fidelity is their ability to change the relationships between nodes. This is done simply by applying a coefficient, which can be positive or negative. Once the individual coefficients have been applied to every node in the previous layer, the values are summed to give the value of the node in the next layer.

To visualize this, we'll make a simple neural network that only uses pixel 168 and pixel 209 brightness values as inputs, and can only classify digits as 1 or 2:

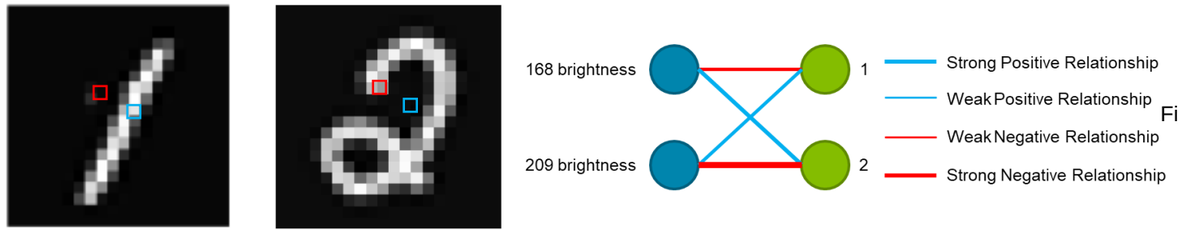


Fig. 3

When being trained, the neural network will alter the relationship between nodes to give the correct output from the given pixel values for images in the training data. This will results in weak/strong positive/negative relationships between nodes.

For a trained network with an input image of a 1, 168 will be a small number and 209 a large number, 209 will therefore have more influence over the final output than 168. 209 will be multiplied by a strong positive relationship to create a high activation for the 1 output node. Conversely, it will be multiplied by a strong negative relationship and create a low activation for the 2 output node, so the image will be classified as a 1.

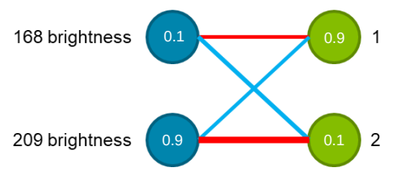


Fig. 4

With an input of a 2, 168 will be large and 209 small, so 168 will hold more influence over output activations. 168 will be multiplied by a negative relationship for the 1 output node, leading to a small activation, however it will be multiplied by a positive relationship with the 2 output node, leading to a high activation. The image will therefore be classified as a 2.

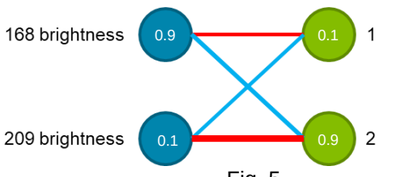


Fig. 5

Changing the relationship between nodes to fit the training data is applied across large neural networks, giving thousands of parameters to vary. This allows the network to detect subtle differences in input data and give accurate predictions.

To ensure the network's parameters are representative of data it could encounter, it's trained on 1000's of examples. This ensures the network isn't thrown off by something it hasn't seen before, and doesn't show an affinity for one classification.

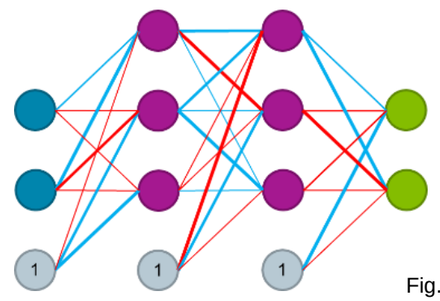


Fig. 6

# Neural Networks: The Maths

## The Relationship between Nodes

The output is calculated by feeding information through the network from input, using the sum of nodes' values from the previous layer to calculate node values in the next. Focusing on the calculation of one node's activation will give further insight into the relationship between them:

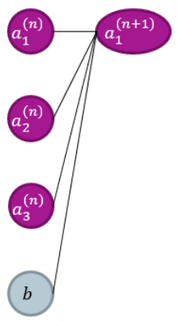


Fig. 7

The activation of a node in the layer $n+1$ is equal to the sum of the activations ($a$) in the layer $n$ and the bias ($b$), each multiplied by their own weight, $\theta$. The sum of this equation is then put through the sigmoid function $\sigma$, which is explained in the Logistic Regression poster.

$$a_1^{(n+1)} = \sigma(\theta_1^{(n)} a_1^{(n)} + \theta_2^{(n)} a_2^{(n)} + \theta_3^{(n)} a_3^{(n)} + \theta_4^{(n)} b^{(n)}) \qquad \text{Eq. 1}$$

There's an individual weight for the link between each node in two adjacent layers, which can be represented in a matrix. The activations in the layer $n$ are spread across columns, and in the layer $n+1$ across rows. For two layers with three activations each, the matrix would be of the form:

$$\begin{matrix} a_1^{(n)} & a_2^{(n)} & a_3^{(n)} & b^{(n)} \end{matrix}$$

$$\begin{matrix} a_1^{(n+1)} \\ a_2^{(n+1)} \\ a_3^{(n+1)} \end{matrix} \begin{bmatrix} \theta_{11}^{(n)} & \theta_{12}^{(n)} & \theta_{13}^{(n)} & \theta_{14}^{(n)} \\ \theta_{21}^{(n)} & \theta_{22}^{(n)} & \theta_{23}^{(n)} & \theta_{24}^{(n)} \\ \theta_{31}^{(n)} & \theta_{32}^{(n)} & \theta_{33}^{(n)} & \theta_{34}^{(n)} \end{bmatrix} \qquad \text{Fig. 8}$$

Using eq. 1 and Fig. 8, the calculation of all nodes in layer $n+1$ can be performed efficiently using matrices:

$$\begin{bmatrix} a_1^{(n+1)} \\ a_2^{(n+1)} \\ \vdots \\ a_i^{(n+1)} \end{bmatrix} = \begin{bmatrix} \theta_{11}^{(n)} & \theta_{12}^{(n)} & \cdots & \theta_{1j}^{(n)} \\ \theta_{21}^{(n)} & \theta_{22}^{(n)} & \cdots & \theta_{2j}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{i1}^{(n)} & \theta_{i2}^{(n)} & \cdots & \theta_{ij}^{(n)} \end{bmatrix} \begin{bmatrix} a_1^{(n)} \\ a_2^{(n)} \\ \vdots \\ a_{j-1}^{(n)} \\ b^{(n)} \end{bmatrix} \qquad \text{Eq. 2}$$

To extract the output from a given input in a trained neural network, Eq. 2 is applied between all layers consecutively. Starting with the input layer, the first hidden layer is calculated, followed by the second, through to the output. This is known as a **forward propagation.**

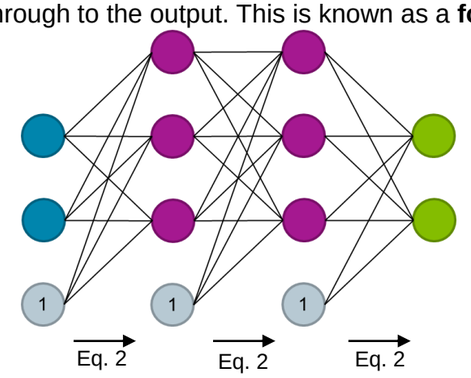


Fig. 9

## Backpropagation

Tuning the weights ($\theta$), which modify the relationship between nodes, is the key to training a neural network to predict the correct output. This is done via **backpropagation**. In contrast to feeding forward to predict an output, backpropagation communicates error backward through the network. The network will be trained on 1000's of examples, but to understand what's going on, we'll just focus on one:

Let's say we're training a network to define the flight phase of an aircraft (Ground, Climb, Cruise and Descent), our output layer will have four nodes, representing each phase. The first step is to run the network with a set of random weights and see how it performs with a given input, our one training example is "Climb", but the network has classified it as "Ground":

$$\begin{bmatrix} Altitude \\ Mach \\ \vdots \\ C_L \end{bmatrix} \xrightarrow{\text{Hidden Layers}} \begin{matrix} [[\ 0.00006097] & [[\text{b'CRUISE'}] \\ [\ 0.7499759\ ] & [\text{b'DESCENT'}] \\ [\ 0.55526551] & [\text{b'CLIMB'}] \\ [\ 0.98756056]] & [\text{b'GROUND'}]] \end{matrix} \qquad \text{Fig. 10}$$

Clearly, the network did not perform well. We want it to provide a value close to 1 for the node corresponding to climb, and values close to 0 for the others. To tell the network this we can use the label for a training example to define the error as the difference between the output ($a^{(output)}$) and the label ($y$).

$$\delta^{(output)} = a^{(output)} - y \qquad \text{Eq. 3}$$

$$\begin{matrix} [[\ 0.00006097] \\ [\ 0.7499759\ ] \\ [\ 0.55526551] \\ [\ 0.98756056]] \end{matrix} - \begin{matrix} [[0] \\ [0] \\ [1] \\ [0]] \end{matrix} = \begin{matrix} [[\ 0.00006097] \\ [\ 0.7499759\ ] \\ [-0.44473449] \\ [\ 0.98756056]] \end{matrix} \qquad \text{Fig. 11}$$

The output here represents how we want the network to change its output for this example. This error can be propagated backwards, for all layers except the output. This is calculated as follows:

$$\delta^{(n)} = (\theta^{(n)} \delta^{(n+1)}) \cdot a^{(n)}(I - a^{(n)}) \qquad I\text{-Identity Matrix} \qquad \text{Eq. 4}$$

This error can then be used to tweak the weights such that the network would output an answer closer to the desired one. The easiest way to think about this is that the error tells you how far from the desired activations the real ones were, and the weights and biases in the layer before can be changed to get closer to the desired values. This is reflected in the equation for calculating the changes to the weights (equation below applies to each one), where the change to the weights is directly dependent on the error of the layer they feed into:

$$\Delta_{ij}^{(n)} = a_j^{(n)} \cdot \delta_i^{(n+1)} \qquad \text{Eq. 5}$$

The above is applied across every training example (for $m$ training examples), and the $\Delta$ from each training example is added together into $\nabla$. Once this has been done for all training examples, each $\nabla$ is subtracted from their respective weight, and the process is repeated for another training iteration. For one training iteration:

$$\nabla_{ij}^{(n)} = \sum_{l=1}^{m} \Delta_{ij}^{(n)}{}_{l} \qquad \text{Eq. 6}$$

$$\theta_{ij}^{(n)} := \theta_{ij}^{(n)} - \nabla_{ij}^{(n)} \qquad \text{Eq. 7}$$

# Building a Neural Network

source code can be found at P:\ENGINEERING\FPO\Digitalisation\1 FPO Projects\IIX Data Analytics\02-General presentations\Knowledge folder - preparation\Data Science Learning Framework\Machine Learning Posters\4 - Neural Networks
(written letter identification network also available)

## Introduction

We'll use the example briefly mentioned above, classifying the flight phase of an aircraft, to build a neural network. Using in-service timeseries data from aircraft sensors, we'll use 30 flights (3161 rows) so the network trains quickly, however it's normally advisable to use more data than this for robustness.

## Data

Picking the right input data is important, 13 input parameters have been selected that should have some relationship to the flight phase. The data should then be separated into features (X) that the network will use to predict phases, and labels (y) that the network will use to train/ test its success.

| flight_segment | | altitude | mach | TAS | vertical_speed | mean_cruise_alt | static_pressure | WHLSPD_1 | AOA | flap_pos | COG | main_g_out |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| CRUISE | 0 | 37024.33333 | 0.781542 | 445.291667 | -17.000000 | 37505.77471 | 212.320312 | 0.0 | 2.373047 | 0.0 | 27.451508 | 0 |
| CRUISE | 1 | 37037.33333 | 0.794167 | 453.104167 | -6.333333 | 37505.77471 | 212.046875 | 0.0 | 2.197266 | 0.0 | 27.171492 | 0 |
| DESCENT | 2 | 38993.66667 | 0.773125 | 436.541667 | -14.666667 | 37505.77471 | 193.257812 | 0.0 | 2.812500 | 0.0 | 28.031540 | 0 |
| CRUISE | 3 | 39031.33333 | 0.784958 | 443.000000 | 18.666667 | 37505.77471 | 192.796875 | 0.0 | 2.460938 | 0.0 | 28.271553 | 0 |
| CRUISE | 4 | 39051.33333 | 0.785208 | 442.729167 | -74.000000 | 37505.77471 | 192.585938 | 0.0 | 2.416992 | 0.0 | 28.391560 | 0 |

Table 1

The labels should then be configured to the output of the neurons, to do this each training example label should be converted to a vector of 0's of length 4 (number of classes), with a 1 in the position of the phase classification of that training example.

Classification of each position in vector $\begin{bmatrix} \text{GROUND} \\ \text{CLIMB} \\ \text{CRUISE} \\ \text{DESCENT} \end{bmatrix}$ $\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$ "Climb" label $\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$ "Descent" label

Fig. 12

The data should then be split randomly into training and testing data using a roughly 70/30 split. The network's weights will be tweaked using the training data, then the testing data can be used to validate its accuracy, as it won't have seen them before.

## Structure

To create our network, we must create two sets of matrices: the activations and the weights. Let's say we want to create a network with 2 hidden layers of 50 nodes. We would create 4 vectors for the activations:

- The input layer: this must be a vector whose length is the number of features, plus 1 for the bias, so 14 nodes in our case
- The hidden layers: these must both of vectors of length 50, plus 1 for the bias.
- The output layer: this must be a vector whose length is the number of available classifications, in our case, 4.

These vectors should be initialized with a value of 1, so that the bias has an effect. All non-bias activation values will be overwritten during feedforward propagation from the input layer.

The size of each weights matrix is dictated by the size of the activation vectors. If a weight matrix is acting between layer $n$ and $n+1$, it must have as many rows as nodes in the layer $n+1$ (excluding bias), and as many columns as nodes in the layer $n$ (including bias). So for our network, the sizes of the weight matrices will be as follows:

- Input layer - 1st hidden layer: 50x14
- 1st hidden layer - 2nd hidden layer: 50x51
- 2nd hidden layer - output layer: 4x51

$$\begin{array}{c} \\ a_1^{(n+1)} \\ a_2^{(n+1)} \\ a_3^{(n+1)} \end{array} \begin{array}{c} \begin{array}{cccc} a_1^{(n)} & a_2^{(n)} & a_3^{(n)} & b^{(n)} \end{array} \\ \begin{bmatrix} \theta_{11}^{(n)} & \theta_{12}^{(n)} & \theta_{13}^{(n)} & \theta_{14}^{(n)} \\ \theta_{21}^{(n)} & \theta_{22}^{(n)} & \theta_{23}^{(n)} & \theta_{24}^{(n)} \\ \theta_{31}^{(n)} & \theta_{32}^{(n)} & \theta_{33}^{(n)} & \theta_{34}^{(n)} \end{bmatrix} \end{array}$$

When these matrices are generated, they should be initialized with random values for each weight.

## Training the Network

To train the network, each training example (row) once for each training iteration. This can be done using equations defined in the previous page.

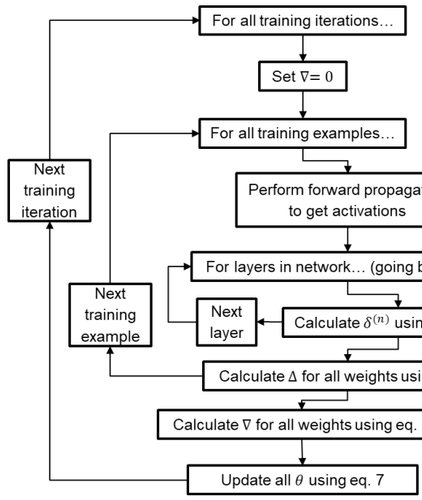


Fig. 13

## Results

To quantify how our network performed, we can implement forward propagation on all rows in the test data. We can then quantify the % accuracy with:

$$Accuracy\ (\%) = \frac{Correctly\ assigned\ test\ rows}{total\ test\ rows} \cdot 100 \quad \text{Eq. 8}$$

Our Network was able to achieve 92% accuracy for this example. It's also worth plotting the test data to visualise the classification.

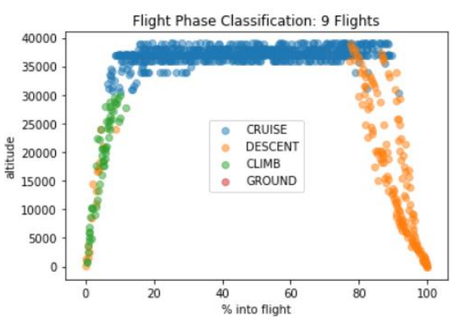


Fig. 14