How Neural Networks Work

MLP Model

The standard example model used to explain neural networks is the number-image classification problem. A greyscale image is represented as a 3D grid of pixels. Each pixel is assigned a value between 0 and 1. A value of 1 would represent a completely white pixel and a value of 0 would represent a completely black pixel. For the YC_bC_r color model, the Y represents luminance (value 0-255), C_b represents the blue chrominance, and C_r represents the red chrominance. For a black and white image, C_b and C_r are both set to zero, so the image could be represented by a matrix of luminance values assigned to each pixel. The EMNIST dataset has 70,000 images of hand drawn numbers, fitted to a 28 x 28-pixel grid. Each image is labelled with its correct identification.

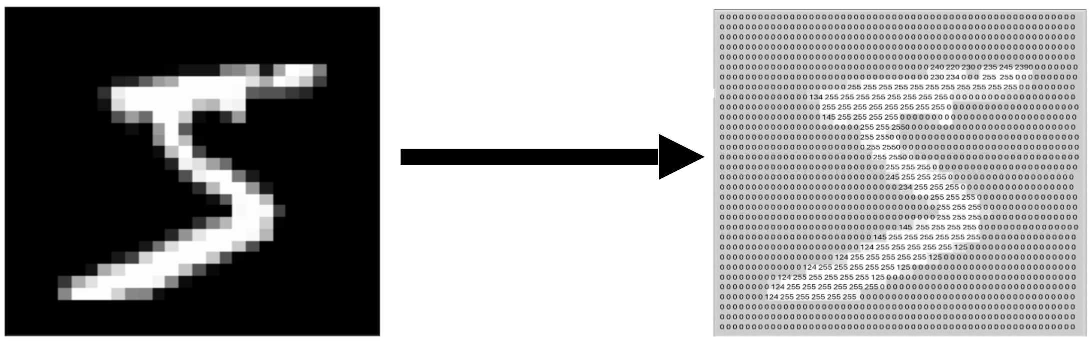


Figure 1 MNIST 5 from database. The hand drawn image is transformed into a matrix with each pixel given a luminance value.

Because the chrominance values are ignored, each image can thus be represented as a 28x28 (784)-dimensional vector.

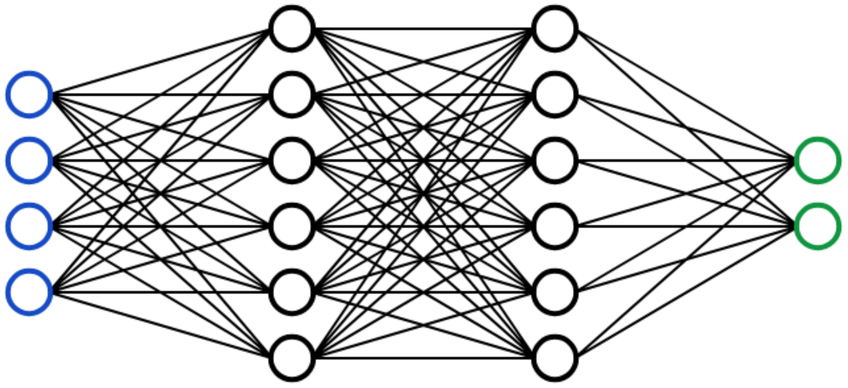


Figure 2 A simple MLP neural network. Each blue circle on the left would represent a pixel luminance value (0-255) . Since there are 28x28 pixels, there would be 784 blue circles. Since we are classifying 10 digits, the farthest right column would have 10 green circles.

Neural networks have hidden layers. Hidden layers are outputs of functions performed on input data that are then fed into further hidden layers or the final output. In figure two, the first hidden layer has 6 circles. This means that if I had a four-pixel image (four blue circles), each pixel value would be fed into 6 functions. A single circle in the first 6-unit hidden layer, which we call a *node* would be a sum of each pixel's function output mapped to that circle. If the function that takes each pixel's luminance value n as an input is the same for every input value, we can write each output value in the first hidden layer as $y = \sum_{n=0}^{4} f(n)$. The function $f(n)$ that maps each pixel's value to an output is a linear function: $f(n) = wn + b$. Remember n is the luminance value. Each luminance value's contribution to the output $y_1 = \sum_{n=0}^{(4)} f(n)$ is determined by the size of $f(n)$. In this case, the size of $f(n)$ for each output depends on the unique value of w multiplied by the luminance value n. The y-intercept of each function, b, referred to as the *bias* in neural networks, is a constant value that is subtracted (or a negative number added) from the output of each unique function from one layer to the next. Thus, the complete mapping of a pixel number value to a hidden layer circle is $f(n) = wn + b$.

Each output, $y_i = \sum_{n=0}^{\#\, nodes\ in\ previous\ layer} f(n)$, is computed and then the process is repeated by treating each output y_1 the same as the pixel luminance value n to calculate the next layer of outputs, y_2: $y_2 = \sum_{n=0}^{6} f(y_1)$ and $f(y_1) = wy_1 + b$. The process is again repeated with y_2 as the inputs to compute the final output values of the entire neural network.

Because the size of the output value of each node reflects its magnitude of contribution throughout the network, we call the value assigned to hidden layer nodes "activations" because a large value would activate subsequent nodes. Say we want hidden layer nodes to be either activated or deactivated. We want to constrain the value of our output functions $f(y_n)$ to be either 0 or 1. To do so, we apply a sigmoid activation function on each node's output value $f(n_1)$:

$$\sigma_f = \frac{1}{1 + e^{-f(y_n)}}$$

Looking at the function, as $f(y_n)$ approaches positive infinity, $e^{-f(y_n)}$ approaches 0, so σ_f approaches 1. Alternatively, as $f(y_n)$ approaches negative infinity, $e^{-f(y_n)}$ approaches infinity, so σ_f approaches 0. These activations are introduced into neural networks so that hidden layer nodes are not directly linearly correlated. There are many other activation functions used in neural networks to introduce nonlinearity, such as the tanh function,

$$y_{n+1} = \frac{2}{1 + e^{-2\left(f(y_n)\right)}} - 1$$

which has similar asymptotic behavior as the sigmoid function.

Now we know how to propagate through a neural network to compute final outputs. However, starting with random weights and biases, our final outputs are completely random. Using labelled data, we then "train" the neural network by tuning the weights and bias values to correctly predict unknown outputs. To establish a regiment for improvement, there must first be a quantitative evaluation metric that captures how good our weights and biases are. The most simply quantitative metric is an MSE function that measures the difference between the neural network's output and the correct output.

$$MSE = \left|\left|y_f - y_{actual}\right|\right|^2.$$

Since in our case we do not just have one output node, but two output nodes, the MSE loss will be the sum of each i in the final layer:

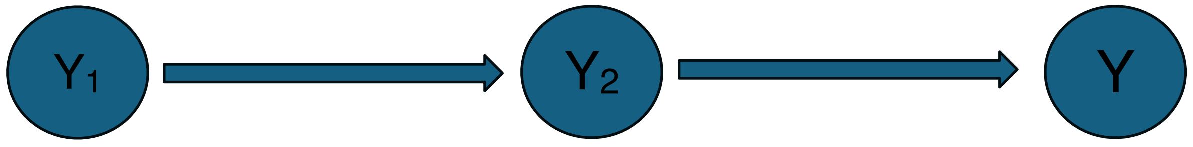
$$C(y_f^i,\ y_{actual}^i) = \sum_{n=0}^{i} ||\left(y_f^i - y_{actual}^i\right)||^2.$$

The goal of the neural network is to accurately predict each final output node y_{actual}^i, so $y_f^i - y_{actual}^i \approx 0$. If you recall from multivariable calculus, the gradient of a function of multiple input variables (i.e. $f(x, y, z)$)is computed by taking the partial derivative of each input variable with respect to the output (i.e $\nabla f = \left\langle \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right\rangle$). The computed gradient tells you the direction of steepest ascent, and the magnitude of the gradient tells you the slope of the ascent. Analogously our MSE cost function is a multivariable function of i number of inputs y_f^i and y_{actual}^i . Because the y_{actual}^i values are not part of our neural network and thus are treated as constants, we compute the gradient with respect to our computed y_f^i values.

$$\nabla C = \left\langle \frac{\partial C}{\partial y_f^0}, \frac{\partial C}{\partial y_f^1}, \frac{\partial C}{\partial y_f^2} \ldots, \frac{\partial C}{\partial y_f^i} \right\rangle$$

We want to *minimize* the cost function. So, we want to compute the negative gradient of the cost function to find the direction of steepest *decent*. Formally, this method is referred to as *gradient descent*. Because each output value y_f^i has been computed through multiple layers of functions with parameters (weights and biases) we want to optimize, calculating each component in the gradient vector($-\ \nabla C$) requires implementation of the chain-rule procedure.

Let's take a look at a very simple multi-layer perceptron. One in which we have only one input, one hidden layer, and one output.



The following equations describe the simplest version of a neural network:

$$y_1 = n \qquad 1.1$$

$$y_2 = y_1 w_1 - b_1 \qquad 1.2$$

$$y_3 = \sigma\left(y_2 w_2 - b_2\right) \qquad 1.3$$

$$C = \left|\left|y_3 - y_{actual}\right|\right|^2 \qquad 1.4$$

Say we are trying to predict whether a chemist will finish her PhD. The input y_1 is the probability of the chemist getting married within her first two years. We know that if the probability is above 0.7, she will likely not finish her PhD. However, if the probability is below 0.7, she will likely finish her PhD. In this simple neural network, we have four unknown parameters: w_1, b_1, w_2, b_2. We want to optimize each parameter to correctly predict the probable outcome of the student finishing her PhD based on the probability that she will get married.

We assign random weights and biases initially to the neural network: $w_1 = 0.2,\ b_1 = 0.8,\ w_2 = 0.4,\ b_2, =\ 0.9$. We are going to use the sigmoid activation function $\sigma\ = \dfrac{1}{1+e^{-(y_i w_i - b_i)}}$ on our output values because we want the final output to predict one of two options: likely (1) or unlikely (0). We start with our first training value 0.6. Propagating through the layers we get the following values:

$$y_1 = 0.6,\ y_2 = -0.68,\ y_3 = 0.65$$

Our randomly initialized weights and biases did not predict the correct output value of 0 ($y_3 = 0.65$), and our initial cost is: $C = 0.425$. So, we must adjust our weights and biases. To do so, we must calculate the negative gradient of the cost function C (y_{actual} values are treated as constants).

$$-\nabla C = \ -\left\langle \frac{\partial C}{\partial y_3} \right\rangle$$

Because the whole purpose of designing a neural network is to compute y_3 by having optimal weights and biases, we must find how to minimize C with respect to the weights and biases themselves. We want our gradient to be a vector relating our cost function as a function of unknown input variables of weights and biases:

$$C\big(w_1, b_1, w_2, b_2\big) = \left|\left|(\sigma\Big(\big(nw_1 - b_1\big)w_2 - b_2\Big)) - y_{actual}\right|\right|^2 \qquad 2.11$$

Using the sigmoid function $\dfrac{1}{1+e^{-x}}$ as the activation function σ

$$C\big(w_1, b_1, w_2, b_2\big) = \left|\left|\frac{1}{1-e^{-((nw_1 - b_1)w_2 - b_2)}} - y_{actual}\right|\right|^2 \qquad 2.21$$

The above cost function is identical to 1.4, but y_3 is substituted with the subsequent functions used in its computation (1.1-1.3). Now, we can calculate the negative gradient of the cost function with respect to each variable:

$$-\nabla C = -\left\langle \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial b_2} \right\rangle$$

You can do this by directly implementing the chain rule with equation 2.2. However, it is more straightforward (and practical with typical neural network sizes) to break down by separately computing derivatives for equations 1.3 and 1.4, and then applying the chain rule. This algorithm is formally known as *backpropagation*. For example, to compute the change in the cost function with respect to w_2:

$$\frac{dC}{dw_2} = \frac{dC}{dy_3} * \frac{dy_3}{dw_2}$$

Using equations 1.2 -1.4, we can see that for w_2,

$$\frac{\partial C}{\partial w_2} = 2|\left|y_3 - y_{actual}\right||*\left(\frac{e^{-(y_2 w_2 - b_2)}}{\left(1 + e^{-(y_2 w_2 - b_2)}\right)^2}\right)*y_2 \qquad 3.1$$

Similarly, for b_2,

$$\frac{\partial C}{\partial b_2} = 2|\left|y_3 - y_{actual}\right||*\left(\frac{e^{-(y_2 w_2 - b_2)}}{\left(1 + e^{-(y_2 w_2 - b_2)}\right)^2}\right)* - 1 \qquad 3.2$$

To calculate the change in the cost function with respect to the changes in w_1 :

$$\frac{\partial C}{\partial w_1} = \frac{\partial C}{\partial y_3} * \frac{\partial y_3}{\partial y_2} * \frac{\partial y_2}{\partial w_1}$$

Using equations 1.1 -1.4, we can see that for w_1,

$$\frac{\partial C}{\partial w_1} = 2\left|\left|y_3 - y_{actual}\right|\right|*\left(\frac{e^{-(y_2 w_2 - b_2)}}{\left(1 + e^{-(y_2 w_2 - b_2)}\right)^2}\right)*w_2*y_1 \qquad 3.3$$

Similarly, for b_1,

$$\frac{\partial C}{\partial b_1} = 2|\left|y_3 - y_{actual}\right||*\left(\frac{e^{-(y_2 w_2 - b_2)}}{\left(1 + e^{-(y_2 w_2 - b_2)}\right)^2}\right)*w_2* - 1 \qquad 3.4$$

To compute the overall gradient vector for the cost function, you compute each partial derivative $(\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b_1}, \frac{\partial C}{\partial w_2}, \frac{\partial C}{\partial b_2})$ for each training sample in your training set and then average the partial derivatives computed for each sample. For the sake of reducing complexity, I will demonstrate computing the gradient with just 1 training sample ($n = 0.6,\ y_{actual} = 0$).

Substituting our initial parameter values: $w_1 = 0.2,\ b_1 = 0.8,\ w_2 = 0.4,\ b_2, =\ 0.9$ into each of the four respective partial derivative functions that make up the vector ∇C, we get:

$$-\nabla C = -\langle 0.037,\ -0.061,\ -0.10,\ -0.15\ \rangle$$

Now, we establish a learning rate $n = 1$ that will determine the magnitude of each weight and bias adjustment in the direction of its corresponding gradient value. Using the variable p to indicate a general parameter (weight or a bias),

$$p' = p - n \nabla C \tag{4.1}$$

We get a new set of weights and biases: $w_1{}' = 0.237,\ b_1{}' = 0.738,\ w_2{}' = 0.437,\ b_2{}' = 0.745$. Now, starting with the same input of $y = 0.6$, the new cost is calculated to be $C' = 0.38$, which is less than our initial cost $C = 0.425$.

Imagine we have a set of 500 training samples that we feed through our network. To optimize the parameters using gradient descent and the backpropagation algorithm, this requires computing the gradient for each sample, averaging the gradients, and then readjusting the parameters using eq. 4.1. That would require computing equations 3.1-3.4 500 times, which is *a lot* of computation. Fortunately, for our small neural network, even our personal computers would be able to perform this sequence of computations easily. However, most neural networks used for real deep learning applications have thousands, millions, and even billions (e.g. ChatGPT) of parameters that need to be optimized. Even supercomputers would run into issues optimizing these complex networks.

Thus, there are many methods used by ML engineers to reduce the computational cost of parameter optimization using gradient descent. One method used ubiquitously is to separate the training set into batches, then optimize the parameters by performing the gradient descent procedure on every sample in each batch. Every round that gradient descent is performed on a batch is called an *epoch*. This entire procedure is referred to as *stochastic gradient descent (SGD)*.