#h1 24 - Building your first artificial neural network

#h2 Start small, aim high

#h3 24.1 - Introduction

#pg The beauty of artificial neurons is that we can safely abstract many of the biological nuances of actual neurons away without losing the computational promise of neuronal processing. We can leave out the neurotransmitters, the action potentials, and many other biological processes. We can therefore define a node (the artificial counterpart of a neuron) as an abstract computation device that receives data from other nodes, integrates this data, and sends out a result to yet another set of nodes. So, what is this data we are passing around? And how do we integrate the information and compute an output for other nodes to receive? In other words, what is the computational framework of artificial neural networks? The answer to the first is straightforward: we are using numerical data representing something of interest (an image, a set of demographics of a patient). The answer to the second question defines the entire field of artificial neural networks. But we can start simple. Those who have spent time going through the earlier chapters and already have experience with gradient descent will see the parallel with how we train neural networks. If not, this chapter will help you prepare to understand gradient descent in the context of neural networks, as the learning rule we will describe is essentially its predecessor.

#h3 24.2 – Definitions

#pg The groundwork for how we think about ANNs was laid by McCullough and Pitts. They proposed a standard neural network model that is still in use today. Neural networks consist of nodes (artificial neurons) that receive inputs from other nodes. We then sum these inputs to yield the total activation of the node. In the McCullough and Pitts model (MP), this activation is subsequentially thresholded to a binary output before being sent as inputs to further nodes. In the MP model, inputs to the node are binary, and because of the thresholding within the node, nodes only send binary input to other output nodes. By keeping the transmission between nodes binary, McCullough and Pitts mimicked biological neurons that communicate by sending single pulses of activity called action potentials (1) interspersed with no output (0). The threshold the MP model uses is a simple Boolean operation, testing whether the activation is equal or larger to a parameter theta. Binary outputs ignore that neurons communicate with the rate they produce action pulses, but we will get back to that.

#im ../assets/figures/024/024-01.png 50 256 Figure 24.1 - Individual inputs in the MP model are denoted as xi and are binary. Collectively, we denote these as vector x. Theta denotes the thresholding value. Activation g is the sum of all inputs, and function f represents the thresholding operation on g yielding the output.

#br

#im ../assets/equations/024/png/024-01.png 50 32

#im ../assets/equations/024/png/024-02.png 50 64

#im ../assets/equations/024/png/024-03.png 50 48 Equations 24.1-3 - The three equations governing the MP model. The first states the inputs to the MP unit are binary (24.1). The second computes the total activation g of the MP unit as the sum of all its inputs xi (24.2). The third represents the thresholding operation f on the activation g, using threshold theta. Activation equal to or exceeding threshold theta yields an output y of 1, and any other activation yields 0.

#h3 24.3 – Building an AND logic gate

#pg With its binary inputs and outputs, the MP might look very basic compared to the massive deep belief neural networks we have today. But it is a great starting point to understand them. So, let's build our first neural network, using two input nodes and only one output node to construct a logic AND gate. An AND gate compares two bits and yields a 1 only when two input bits are both 1 themselves.

#ns 70 center

#nh

#nc Input 1

#nc Input 2

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc 0

#nl

#nr

#nc 0

#nc 1

#nc 0

#nl

#nr

#nc 1

#nc 0

#nc 0

#nl

#nr

#nc 1

#nc 1

#nc 1

#nl

#ne

#ca Table 24.1 - Input/output relationship of the AND logic gate

#pg The parameter that is up to us to find is theta, the threshold value that will produce an output of 1, given that the inputs are 1 also. And this threshold isn't hard to find. The activation of the output node of our modest neural network is the sum of all its inputs, which only takes on the value of 2 for the one input pattern that should produce an output of 1. Therefore, if we use a threshold theta of 2, we obtain the correct result, see Table 24.2.

#ns 70 center

#nh

#nc Input 1

#nc Input 2

#nc Activation

#nc Threshold

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc 0 + 0 = 0

#nc >=2

#nc 0

#nl

#nr

#nc 0

#nc 1

#nc 0 + 1 = 1

#nc >=2

#nc 0

#nl

#nr

#nc 1

#nc 0

#nc 1 + 0 = 1

#nc >=2

#nc 0

#nl

#nr

#nc 1

#nc 1

#nc 1 + 1 = 2

#nc >=2

#nc 1

#nl

#ne

#ca Table 24.2 - Creating an MP unit that performs the AND operation using a threshold theta.

#br

#pg And there you have it! Your first neural network! You probably already worked out that an OR gate isn't hard to build, either. Setting theta to 1 instead of 0 will accomplish that.

#ns 70 center

#nh

#nc Input 1

#nc Input 2

#nc Activation

#nc Threshold

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc 0 + 0 = 0

#nc >=1

#nc 0

#nl

#nr

#nc 0

#nc 1

#nc 0 + 1 = 1

#nc >=1

#nc 1

#nl

#nr

#nc 1

#nc 0

#nc 1 + 0 = 1

#nc >=1

#nc 1

#nl

#nr

#nc 1

#nc 1

#nc 1 + 1 = 2

#nc >=1

#nc 1

#nl

#ne

#ca Table 24.3 - Creating an MP unit that performs the OR operation using a threshold theta.

#br

#pg Let's do one more: a NOT gate. We only have one input (a bit of 0 or 1), and we want the opposite bit to be the output. For this, we can use an inverted threshold, testing for values smaller than or equal to our theta. And with a theta of 0, our MP model yields the correct output, as shown in Table 24.4.

#ns 70 center

#nh

#nc Input

#nc Activation

#nc Threshold

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc <1

#nc 1

#nl

#nr

#nc 1

#nc 1

#nc <1

#nc 0

#nl

#ne

#ca Table 24.4 - Creating an MP unit that performs the NOT operation using a threshold theta. Please note that we have inverted the application of the threshold (from >= to <)

#h3 24.4 – The Perceptron

#pg Introduced by Frank Rosenblatt, the Perceptron modernized the MP model by introducing some new features that made it far more versatile. It also introduced an automatic method of training models so that we can find their parameters through computation. It put the word learning firmly into the term 'machine learning'. However, it is still a very basic classifier: its outputs are binary. The main upgrade from the MD model was the introduction of weights. We now apply a real-valued weight to each input before being summed to yield its activation. Second, Rosenblatt removed the theta parameter in favor of an unparameterized threshold function where the node would produce an output of 1 only if the activation (weighted sum of inputs) is larger than 0, 0 otherwise. Equations 24.4 to 24.6 show the subtle difference between the original MP implementation and the Perceptron.

#im ../assets/equations/024/png/024-04.png 50 32

#im ../assets/equations/024/png/024-05.png 50 64

#im ../assets/equations/024/png/024-06.png 50 48 Equations 24.4-6 - The three equations governing the Perceptron model. The first states that the inputs to the Perceptron are still binary, but each input node has a weight wi associated with it, which is real-valued (24.4). The second equation tells us how to compute the total activation g of the Perceptron unit as the sum of all its inputs xi multiplied with each weight wi, plus a bias b (24.5). The third represents the thresholding operation f on the activation g. Activation exceeding 0 yields an output y of 1, any activation that is 0 or less yields an output of 0.

#br

#pg Allowing individual weights on each input, rather than a single threshold, creates a much larger state space: the complexity of patterns that neural networks can handle. The Perceptron also introduced the notion of a bias node. Remember that we need an offset or intercept in both regression and classification to move the function up and down, as the weight parameters can only change the function's slope. We needed to add a constant b to our model: y = ax + b. From an algebraic perspective, this means that, in practice, we need to add a column of ones to our matrix X of inputs. Bias nodes in neural networks achieve the same. Their input will always be one, but we can still set the weight to any value (analogous to the b in y = ax + b). Including this bias node in the calculation of the activation allows for shifting the activation by a fixed amount, independent of any input, just like the constant b in our regression and classification models.

#h3 24.5 – AND, OR, and NOT gates revisited

#pg Let's rebuild our AND, OR, and NOT gate, this time using the perceptron model. We will still figure out the weights by hand as an exercise. In the next section, we will introduce the learning rule that will do that for us automatically. For our AND gate, some experimenting quickly yields a set of parameters w = [0.5,0.5] (the weights on input 1 and input 2 weights) and b = -0.5 (a bias term weight of -0.5) that will do the trick. Only one of the activations is greater than 0, which is the correct one.

#ns 70 center

#nh

#nc Input 1

#nc Input 2

#nc Activation

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc 0 x 0.5 + 0 x 0.5 + -0.5 = -0.5

#nc 0

#nl

#nr

#nc 0

#nc 1

#nc 0 x 0.5 + 1 x 0.5 + -0.5 = +0.0

#nc 0

#nl

#nr

#nc 1

#nc 0

#nc 1 x 0.5 + 0 x 0.5 + -0.5 = +0.0

#nc 0

#nl

#nr

#nc 1

#nc 1

#nc 1 x 0.5 + 1 x 0.5 + -0.5 = +0.5

#nc 1

#nl

#ne

#ca Table 24.5 - Creating a Perceptron unit that performs the AND operation. With weights w = [0.5,0.5] and bias b = minus 0.5, we obtain the required output. Note that we don't list the threshold value theta here. Unlike in the MP model, the thresholding value of a perceptron is always 0.

#br

#pg Simply increasing the strength of the bias term from -0.5 to 0.0 will change our AND to an OR logic gate:

#ns 70 center

#nh

#nc Input 1

#nc Input 2

#nc Activation

#nc Output

#nl

#nr

#nc 0

#nc 0

#nc 0 x 0.5 + 0 x 0.5 + -0.0 = 0.0

#nc 0

#nl

#nr

#nc 0

#nc 1

#nc 0 x 0.5 + 1 x 0.5 + -0.0 = 0.5

#nc 1

#nl

#nr

#nc 1

#nc 0

#nc 1 x 0.5 + 0 x 0.5 + -0.0 = 0.5

#nc 1

#nl

#nr

#nc 1

#nc 1

#nc 1 x 0.5 + 1 x 0.5 + -0.0 = 1.0

#nc 1

#nl

#ne

#ca Table 24.6 - Creating a Perceptron unit that performs the OR operation. With weights w = [0.5,0.5] and bias b = 0.0, we obtain the required output.

#br

#pg Finally, for our NOT gate, we use a weight of -1.0 and a bias of 0.5 to achieve the desired inversion of the bit. And unlike the MP model, the Perceptron doesn't require us to flip the thresholding rule.

#ns 70 center

#nh

#nc Input

#nc Activation

#nc Output

#nl

#nr

#nc 0

#nc 0.0 x -1.0 + 0.5 = 0.5

#nc 1

#nl

#nr

#nc 1

#nc 1.0 x -1.0 + 0.5 = -0.5

#nc 0

#nl

#ne

#ca Table 24.7 - Creating a Perceptron unit that performs the N OT operation. With weights w = [-1.0] and bias b = 0.5, we obtain the required output.

#h3 24.6 - Rosenblatt’s Perceptron learning rule

#pg We set the weights by hand for the examples above, which wasn't all that hard to do. But clearly, for the modern-day deep belief neural networks of millions and millions of weights to set, this is no longer a human task. And optimizing the weights is what makes neural networks so powerful: their ability to learn. It is a good exercise to walk through training a single perceptron. Its learning algorithm isn't that different from the algorithms we use today. Critically, it tries to find the optimal weights by using the difference between a ground truth outcome and what our algorithm currently is predicting. It wasn't until a decade later that the field of AI realized that this defines machine learning: the optimization between predicted and actual outcomes. Rosenblatt's work is truly visionary in that respect.

#h3 24.7 - Perceptron Training Algorithm - Step 0 - Initialization

#pg In the notebook, you can follow along with training a perceptron on the famous Iris dataset. Here, we will talk through the steps the algorithm takes. Again, many of the concepts, ideas, and methods described here are still the basis of modern neural networks. A thorough understanding of it will greatly help us understand the newest generation of nets we have today. The first step is initialization. After our data is in a suitable format, we initialize our network by setting our weights and biases to random values. As we will see later, we shouldn't set them to zero and not make them too large. Therefore, we typically pick these initial weights from a normal distribution with a mean of 0 and a small standard deviation. Next, we train the algorithm. In the original algorithm, we adjust the weights and biases for each data point in our data in turn. Let us walk through the update of a Perceptron unit's parameters w and b using a single input vector x. The Perceptron unit receives two inputs from vector x: x0 and x1. And each is associated with a weight (w0 and w1, respectively). Finally, the unit also has a bias, a fixed input b that is constant and not modulated by any input.

#im ../assets/equations/024/png/024-07.png 50 20

#im ../assets/equations/024/png/024-08.png 50 16 Equations 24.7-8 - The values w and b for our training example, as outlined in sections 24.6 onward.

#h3 24.8 - Perceptron Training Algorithm - Step 1 - Compute Activation and Output

#pg We then first compute the activation of the node for the current input x = [0.5,0]. Remember that activation of a node is the sum of the inputs to it times its associated weights w plus the bias b, see Equation 24.5. So, with two features, we compute the activation for a particular input x, weight w, and bias b, as shown in Equations 24.9-12. We then threshold activation g using a sgn (short for sign, don't ask me why) function. Anything larger than 0 will yield an output of 1, and anything else yields 0.

#im ../assets/equations/024/png/024-09.png 50 18

#im ../assets/equations/024/png/024-10.png 50 16

#im ../assets/equations/024/png/024-11.png 50 16

#im ../assets/equations/024/png/024-12.png 50 20 Equations 24.9-12 - Computing the activation g from inputs x, weight w, and bias b. Using the sgn function, we use this activation g to compute output ypred (y 'predicted').

#h3 24.9 - Perceptron Training Algorithm - Step 2 - Compute the error

#pg We then subtract this output from this data point's known or desired output. Since both values can only take on the value of 0 or 1, this yields a new value which can also only be 0 or 1.

#im ../assets/equations/024/png/024-13.png 50 20 Equation 24.13 - Computing the Error between our predicted output ypred and the actual output y through subtraction, in the case we predict 1, but the desired or actual outcome y is known to be 0.

#br

#pg It is helpful to understand this value in two ways. First, it is the Error between our prediction and the actual or desired outcome. Second, since it is either 0 or 1, we only know whether we made an error or not for this data point, but the magnitude of said Error. Still, we can and will use this binary error value to decide whether our weights need adjusting based on this particular data point.

#h3 24.10 - Perceptron Training Algorithm - Step 3 - Update the weights

#pg The next step multiplies our Error with the input x and a small learning rate r. Since all inputs and outputs in a Perceptron are binary, multiplying our Error with input x will always result in one of three values: -1, 0, and 1. Let's keep this in mind when we unpack the final update rule later. Even though it can only take on a limited set of values, this still represents the gradient, a concept we introduced back in Chapters 14 and 15 when we discussed gradient descent (GD). If you worked through these chapters, you would hopefully see the similarities between the use of gradient descent in linear and logistic regression on the one hand and how the Perceptron is updating its weights on the other. In addition, our regression and classification models incorporated a constant b to model any offset of the data relative to the y-axis. In a neural network, we use a mathematically equivalent method, which we refer to as a bias unit. A bias unit is essentially another input (e.g., x2) with its weight (e.g., w2), but the input x2 is constant and 1 for all possible inputs (e.g., x2 = 1). However, it does have an adjustable weight b, similar to the w's of the input nodes. Also, note that we update our weights by subtracting the weighted gradient. Subtracting rather than adding might seem counterintuitive at first. However, a positive gradient indicates an increase in the value of our function. Since that function here is the Error, we do not want to increase it. Instead, we want to move in the opposite direction of the gradient, following the downwards direction, producing a lower error value.

#im ../assets/equations/024/png/024-14.png 50 20

#im ../assets/equations/024/png/024-15.png 50 20

#im ../assets/equations/024/png/024-16.png 50 24 Equations 24.14-16 - Updating the weights, generalized formula. We update each input weight by subtracting the product of 1) the Error between the actual and predicted outcome, 2) the input, and 3) a small learning rate lr. The bias unit is updated without any reference to the input data, as we defined a bias as a unit receiving a constant value of 1 for all possible inputs.

#br

#im ../assets/equations/024/png/024-17.png 50 20

#im ../assets/equations/024/png/024-18.png 50 20

#im ../assets/equations/024/png/024-19.png 50 24 Equations 24.17-19 - Same as the above, but now using the values we used in our example.

#br

#pg Returning briefly to the similarity between gradient descent and Rosenblatt's learning algorithm: this is no coincidence. They are highly similar, mathematically speaking. The difference is in the details, mainly in terms of the binary levels of activation used in Rosenblatt's Perceptron. A brief survey of the literature suggests that Rosenblatt was unaware of the similarity between gradient descent (which predates the Perceptron quite a bit) as it was introduced initially by Cauchy in 1847 and again by Hadamard in 1904) and his algorithm, but not conclusively. However, that seems far less important than observing the direct mathematical link between many conventional machine learning algorithms and their newer neural network counterparts. Understanding this connection is like peaking under the rug of machine learning: the considerable similarities in optimizing their parameters to yield more accurate predictions. In other words: how they learn.

#h3 24.11 - Perceptron Training Algorithm - Step 4 - Convergence and halt

#pg Once we have iterated over each data point, we can check if we did any better predicting all data points, by computing some performance metric, like loss or accuracy. If this metric stops improving or our increments in performance get too small to be worth it, we can halt the learning. As mentioned in the beginning, it is the precursor of the modern approach to training various machine learning algorithms, from logistic regression to convolutional deep belief neural networks. The key elements of the error, gradient, and learning rate are all there. The only real difference is that the Perceptron uses a binary error function. In contrast, its modern version, gradient descent, uses a continuous-valued error.

#h3 24.12 - Activation Functions

#pg So far, we have seen the sign or threshold activation function. Over the years, researchers introduced several alternative activation functions we can apply to a node's activation to yield an output.

#im ../assets/figures/024/024-02.png 50 128 Figure 24.2 - Three different activation functions. We will explore the differences between these functions in the chapters to come.

#br

#pg These functions include sigmoids, exponentials, step functions, and others. In fact, one of the most significant breakthroughs in our modern ANN era wasn't really because of the amazing GPUs we now had at our disposal or the enormous new data sources to train networks on. Instead, it was the finding and proof that a particular activation function made longer extremely efficient: the rectified linear activation unit or ReLU for short. It is a hybrid between the linear and sgn activation function. Below a certain threshold, this function will always generate 0 output. Above this threshold, the output increases as a linear function of its inputs. This new activation function solved a problem that became increasingly more pronounced as networks became larger: the problem of the vanishing gradients. Remember that the gradient tells us how to update our weights to reduce the loss and increase the performance of our neural networks (as well as the conventional algorithms we discussed in Chapters 14 and 15). When the activation of a node becomes highly positive or negative, notice that the output of applying a sigmoid (see chapter 14) to the activation will saturate and asymptote to 0 for highly negative levels of activation and 1 for highly positive levels of activation. These essentially flat regions of the sigmoid function have gradients that approach 0, meaning there is no gradient of slope to speak off. In other words, the gradient vanishes. Since we use the gradient to adjust the weights, when that gradient is essentially 0, the weights stop being adjusted altogether, and the network stops learning. The ReLU activation function does not have this property, at least for highly positive activation levels, thereby mitigating the problem of these vanishing gradients.

#h3 24.13 - An innovation with a darker side

#pg For all it brought to the table, the Perceptron also marked the decline in interest in artificial neural networks. As discussed earlier, it had a big insurmountable problem: certain logical problems that could never be predicted correctly by the algorithm. More technically, the Perceptron cannot deal with data that are not linearly separable from its single input-output set of layers. This separability problem is easiest to imagine in 2D, but it applies to datasets of any dimensionality. For a 2D input space (like a logical gate), you can, in theory, use a perceptron only when you can draw a straight line between data points belonging to each class. However, we cannot do the same for some logical input-output relationships, like the XOR logic gate, as shown in Figure 24.3. There is simply no way a perceptron could ever give you 100% accuracy. This realization halted much of the research. Because although people realized that adding additional layers between input and output would solve this (at least partially, as we shall see), no learning algorithm could handle training these additional in-between layers, commonly referred to as hidden layers. Until backpropagation came along. Backpropagation was the much-needed algorithm that allowed us to do just that, and we will describe it in some detail in the next chapter.

#h3 24.14 - Biological vs. Artificial Neural Networks, revisited

#pg We started by noting that we can abstract away much of the biological detail we find in real neurons in our artificial neural networks. With biological detail, we refer to all the different neurophysiological processes that go on in biological neural networks, at many different levels. Everything happening in and between neurons, from the expression of genes that provide the blueprint for so-called ion pumps: proteins needed for conducting electrochemical activation to the release and signaling through neurotransmitters between neurons, are the product of a long evolutionary history. But ultimately, these processes all combine into the same basic computational framework: our brains work because our neurons receive and transmit information by combining inputs into outputs. The biological detail of how this is done seems irrelevant from the computational point of view. All we need are suitable models that describe how inputs are combined into outputs mathematically. But some research seems to indicate that we do not fully understand yet how, perhaps, some of these processes we abstracted away should be taken into account. For example, neurons are notoriously noisy. When activated, they will produce a stream of action potentials at some rate, depending on the strength of the stimulus. Louder sounds produce a higher rate of actional potentials in neurons in our auditory system. However, the interval between spikes is far from fixed and shows considerable variation. Across some reasonable period, compared to the stimulus duration, the average number of spikes is highly predictable. Therefore, in artificial neural networks, the activity of a node is seen as analogous to this average number of spikes of a biological neuron, and the variable spacing between any of those spikes is just noise inherent in biological systems. However, some researchers are not convinced that we should be eliminating this variability, as it might not be noise at all. They argue that the variability in the spike train could reflect a signal of some stimulus propriety or state that isn't captured by the average rate of the spikes. Others pointed out the possible importance or necessity of other biological details left out in the design of artificial networks, such as the different neurotransmitters neurons use to signal to one another. It is tempting to think that an artificial neural network will ultimately compute things in a way that is highly similar or even identical to how biological systems have evolved to compute. This might be a false equivalency or, at best, an unproven one. Biological systems inspired artificial neural networks, but it isn't a given that they run similar 'computing algorithms'. Proponents of this point of view often point to backpropagation as an example. While we use it in practically all our modern artificial neural networks, there is no biological counterpart to this algorithm, as far as we know. What essential biological detail did we leave out of our computational abstractions that required us to use backpropagation as an artificial alternative? Time and research will tell.

#h3 24.15 - Demo notebooks

#pg The demo notebooks work through the logic gates implemented by the original MP model and Rosenblatt's Perceptron. In addition, the Perceptron notebooks also implement the perceptron learning rule.

#h3 24.16 - References

#bs

#be

#be

#bp Gurney, K. (2018). An Introduction to Neural Networks. United States: CRC Press.

#bp Hebb, D. (2005). The Organization of Behavior: A Neuropsychological Theory. United Kingdom: Taylor & Francis.

#bp Mack, S. H., Kandel, E. R., Koester, J. D., Siegelbaum, S. A. (2021). Principles of Neural Science, Sixth Edition. United States: McGraw-Hill Education.

#bp Minsky, M. (1988). The society of mind. United Kingdom: Simon & Schuster.

#bp Rosenblatt, F. (1962). Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms. United States: Spartan Books.

#bp Russell, S., Norvig, P. (2016). Artificial Intelligence: A Modern Approach. CreateSpace Independent Publishing Platform.

#be