Statistical Mechanics, Neural Networks, and Machine Learning: *Using Powerful Brain Strategies to Improve Machine Learning*

DRAFT Chapter 5
The Transfer Function:
The Essence of a Neural Network Microstructure

Alianna J. Maren
Northwestern University School of Professional Studies
Master of Science in Data Science Program

Chapter Draft: 2020-01-02

5.1 Neural Network Microstructures and Transfer Functions

In order to design good (and effective) architectures, we need to understand neural networks at three different levels which were identified in the previous chapters:

1. ***Microstructure*** - what happens *inside* a given node or "neuron,"
2. ***Mesostructure*** (where *meso* means *middle* -the structure of a simple neural network, e.g., the number of layers, number of nodes per layer, etc., and
3. ***Macrostructure*** - architectures constructed from multiple interacting neural networks, or when one large neural network contains (relatively self-contained) sub-networks.

In this chapter, we focus on *microstructures'*. Specifically, we're going to look at the various transfer functions that can be used within a neural network. We will give most of our attention, using a simple worked example, to the signmoid transfer function, although we'll also identify other common ones, and we'll also assess which transfer functions best for different neural networks, and why.

Once we've completed this chapter, we will be able to identify:

1. What sort of transfer function(s) to use for a particular kind of neural network,
2. What the implications of that transfer function will be in network performance, and also
3. What kinds of potential problems can result from using our selected transfer function.

The role of transfer functions in neural networks is to take the set of inputs that go into a single node, within the neural network, and *transform* that set of input values into a single, useful output value.

AUTHOR'S NOTE for further work: *Put in a figure showing role of a transfer function in a NN.*

Early neural networks almost exclusively used either the sigmoid (logistic) or hyperbolic tangent (tanh) transfer functions. The following Figure 5.1 illustrates these two functions. The sigmoid function was initially the most popular. However, it too often led to networks that couldn't converge, because ***(fill in)***.

As a result, the hyperbolic tangent became much more popular, especially for the hidden nodes in a network [1].

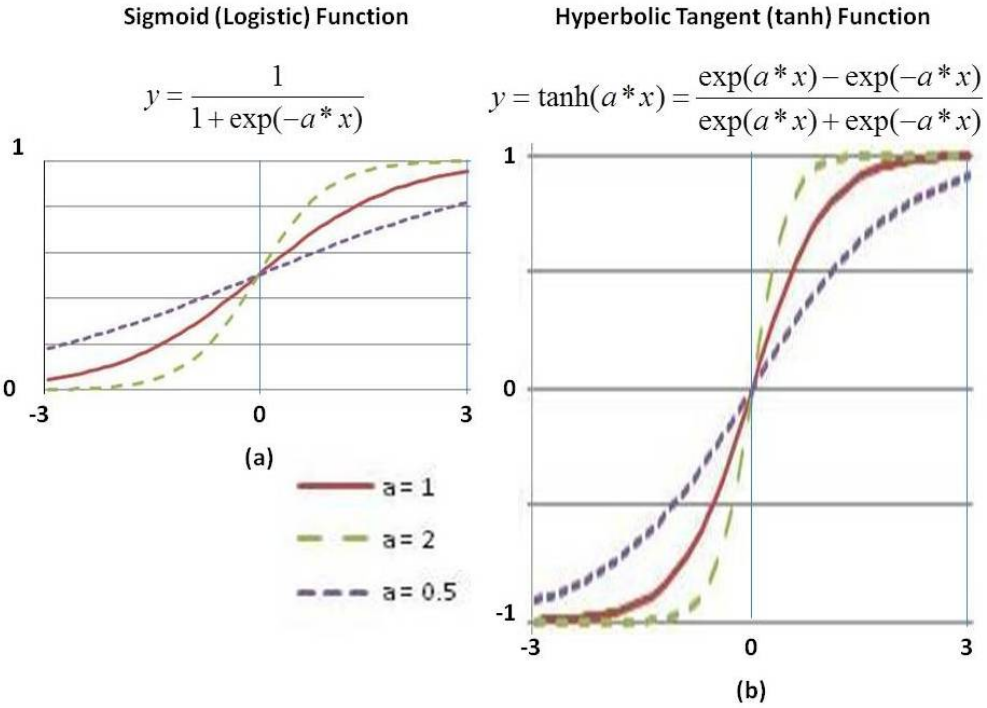

Figure 5.1: The sigmoid (logistic) transfer function.

More recently, a different kind of transfer function, the ReLU (for Rectified Linear Unit) has become popular, primarily for convolutional neural networks (CNNs) and restricted Boltzmann machines (RBMs) [2].

In this chapter, we will define the three most widely-used transfer functions, and show how to compute their derivatives. We need these derivatives, because they are a crucial part of computing the gradient descent, which is the essence of the *learning method* that we'll use for a wide range of neural networks and deep learning architectures.

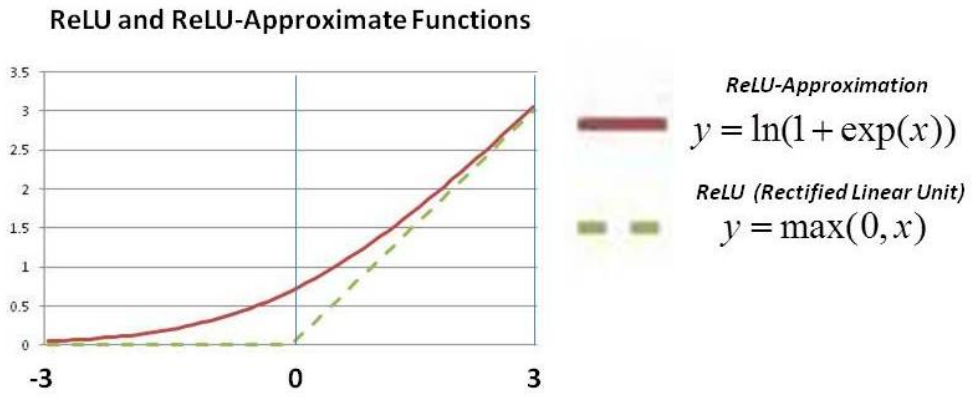

Figure 5.2: The sigmoid (logistic) transfer function.

5.2 Example: Transfer Functions in Action

The following Figure 5.3 shows a diagram for one of the simplest possible neural networks.

Example 5.1.

Figure shows an entire, albeit very simple, neural network that has already been trained to give a good solution for the classic X-OR problem. This problem is important in neural network circles - and was a benchmark in proving a valid *learning rule* for neural networks, because the solution for this is not in a linearly separable space. That is, we can't draw a set of straight lines and "separate" the solution into the appropriate number of spaces, one space per solution.

There are just two inputs to this network, and they can be any of the four possible combinations of 0 and 1. Because we have four possible inputs, we could potentially have four different output "classes." Instead, we want to have *only two* output classes:

- Inputs $0, 0$ give an output ("classification") of 0,
- Inputs of either $1, 0$ OR $0, 1$ (this is the "OR" part of the "X-OR" or "Exclusive-OR" problem) should give an output ("classification") of 1, and
- Inputs $1, 1$ give an output ("classification") of 0.

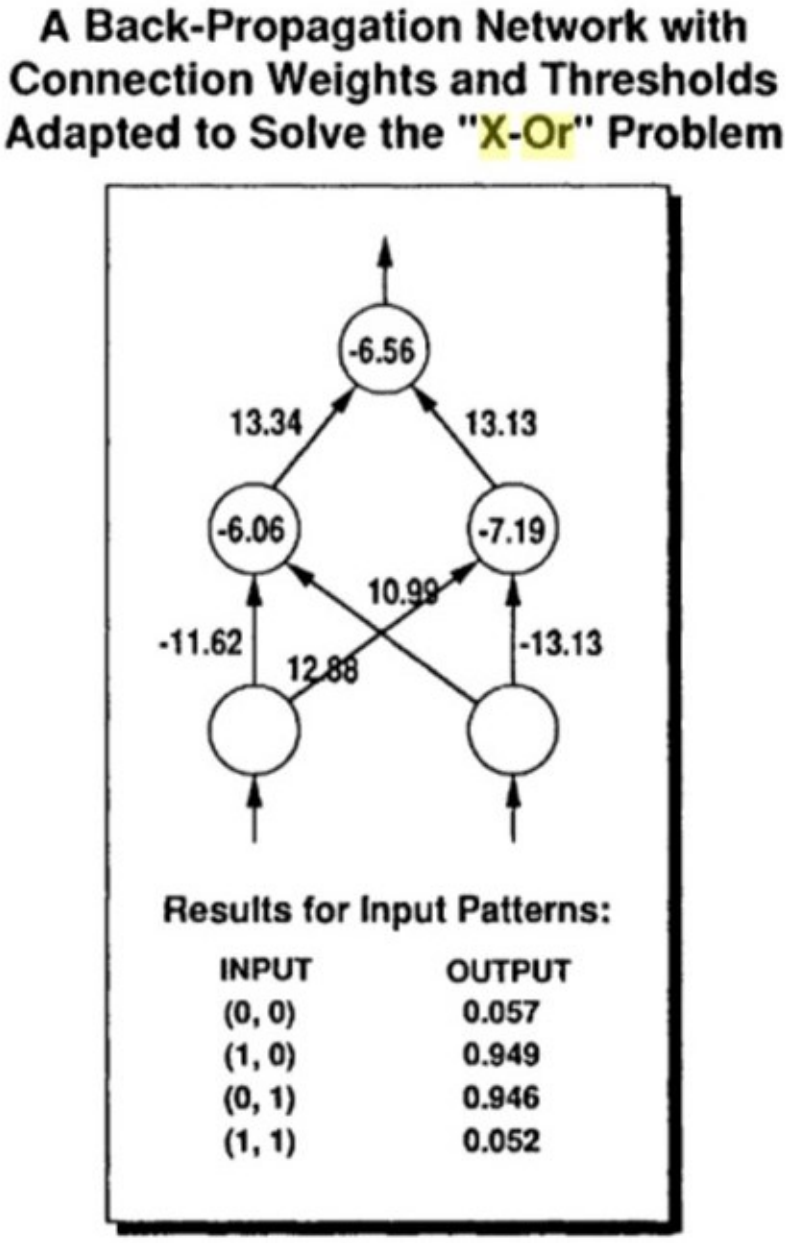

Figure 5.3: A worked example for a simple MLP neural network designed to solve the X-OR problem. This network has three layers, with two nodes in the input layer, two in the middle or "hidden" layer, and just one in the output layer. See the text for a detailed explanation. Figure taken from A.J. Maren et al. (1990), *Handbook of Neural Computing Applications*, Chapter 5, Figure 5.3, pg. 62. (New York: Academic).

It is very easy to create a network that will satisfy the first two conditions for solving the X-OR problem. However, it is *very tough* to get a set of connection weights that will satisfy *all three* of these conditions.

This doesn't mean that there are only a few "solutions" that will work. In fact, even for this easiest-of-all-possible neural networks, there are a wide range of different *solution sets*, where each *set* contains an infinite number of possible values, each of which will work. However, stumbling onto any of these sets is not easy; this is why we've needed *learning algorithms*.

AUTHOR'S NOTE for further work: *Complete the walk-through for this example.*

5.3 The Sigmoid (Logistic) Transfer Function

We'll begin with one of the most commonly-used transfer functions, called the *sigmoid function*, because it produces an S-shaped curve.

The Sigmoid Transfer Function:

$$y = \frac{1}{1 + exp(-\alpha x)} = (1 + exp(-\alpha x))^{-1}. \quad (5.1)$$

See this transfer function in Fig. 5.4.

AUTHOR'S NOTE for further work: Redo the above figure: (1) remove side legend elements; place on the bottom. (2) change the running indices on x-axis to be easier to read. (3) simplify the y-axis - make it easier to read.

Before we go further, let's make a few mathematical comments on this transfer function.

Interpreting the Sigmoid Transfer Function:

Input text HERE

Its purpose, in the world of neural networks, is to scale the inputs into a given node to be within a reasonable range for the outputs. That means that

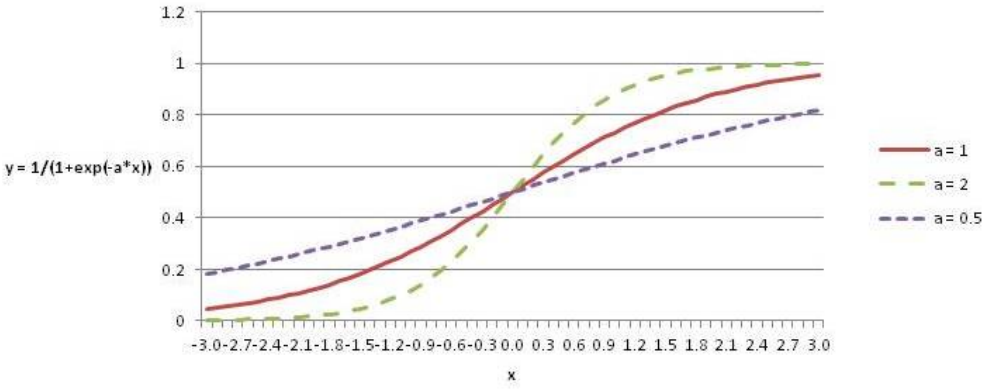

Figure 5.4: The sigmoid (logistic) transfer function.

even if we put very large negative or positive values into the node, we want the output to be much more limited; we're choosing a transfer function here that will limit the outputs to be between 0 and 1.

We want our transfer function to be smoothly continuous; no discontinuities (sudden jumps). Also, we want its overall behavior to be consistent; we don't want something that goes up and then goes down. In other words, we want a function that is *monotonic*. In our particular case, we want a function that is *monotonically increasing*; that is, it starts with small values and moves smoothly to progressively larger values. That is exactly what we get with this particular function.

We notice that this particular function is *asymptotic* at both extremes, which is something that we want. This fulfills our previously-stated requirement, that we want the outputs to be limited within a certain range. By asymptotic, we mean that the function approaches certain limits, but never quite reaches them. When x is a very large negative, the value for y is a very small positive number; close to (but never quite reaching) zero. (As $x \to -\infty, y \to +0$.) Further, when x is a very large positive, the value for y approaches (but never quite reaches) 1. (As $x \to +\infty, y \to 1$.)

5.4 The Transfer Function in Action: Some Examples

During neural network *operations*, we use the transfer function itself. Thus, we gather the inputs to a given node (sum the various inputs, which are the outputs from nodes in the preceding layer, and multiply each of these inputs by their corresponding *connection weight*. These summed and weighted inputs collectively form the input to a given node. Then, we apply the transfer function to that input. (The author likes to think of this as "pushing" the input "through" the transfer function, which is a useful anthropomorphism.) The result of this is node's output, also sometimes called the *activation* of that node.

To reiterate a previous point, the role of the transfer function here is to ensure scaling of the inputs to a more reasonable range. Thus, if the inputs are a large value - say, 10, then the resultant node activation is not very much different than if the inputs are very large, say 100. Just to work this a little further, let's do this calculation, for the case of the sigmoid transfer function, where we let $\alpha = 1$.

We'll call the summed inputs into the node $NdInput$, for *node input*. Just to have notational consistency, let's say that we're working with the inputs from the input layer nodes to the first hidden layer node, which we call H_0.

Note: we're counting our nodes and all other vector and matrix elements using the Python convention of starting the count with 0, rather than with 1. Thus, we also dub our inputs into this node with the appropriate subscript, so we'll use $NdInput_0$ as the summed total inputs to H_0.

For the case where $NdInput_0 = 10$, then $\exp(-10) = 0.0000454$, and $H_0 = F(NdInput_0) = {}^{1}/_{1.0000454} = 0.999955$.

For the case where $NdInput_0 = 100$, then $\exp(-100)$ is = 3.72 times 10 to the power of *negative* 44, which is a very small number, and adding 1 to it is just incrementally over 1.

Thus, 1 over that number (which is incrementally over 1) is a number that is just incrementally *under* 1.

While there's a difference between this number and 0.999954, that difference begins in the fifth significant figure after the decimal. Clearly, not enough to make a big difference.

Just to scope this a little more, let's obtain the transfer function output when the input is 1 instead of 10. In this case, we have $\exp(-1) = 0.3679$,

and $1/1.3679 = 0.7310$. Referring back to Figure 5.4, we that ...

5.5 The Derivative of the Sigmoid Transfer Function

To compute the derivative of the sigmoid transfer function, as with almost every derivative that we'll take in both this and the following chapters, we'll use the *chain rule from differential calculus*.

AUTHOR'S NOTE for further work: Write additional text. Introduce chain rule in simple form.

AUTHOR'S NOTE for further work: Write derivative for the transfer function in chain-rule form.

The derivative of the sigmoid transfer function is shown in Fig. 5.5.

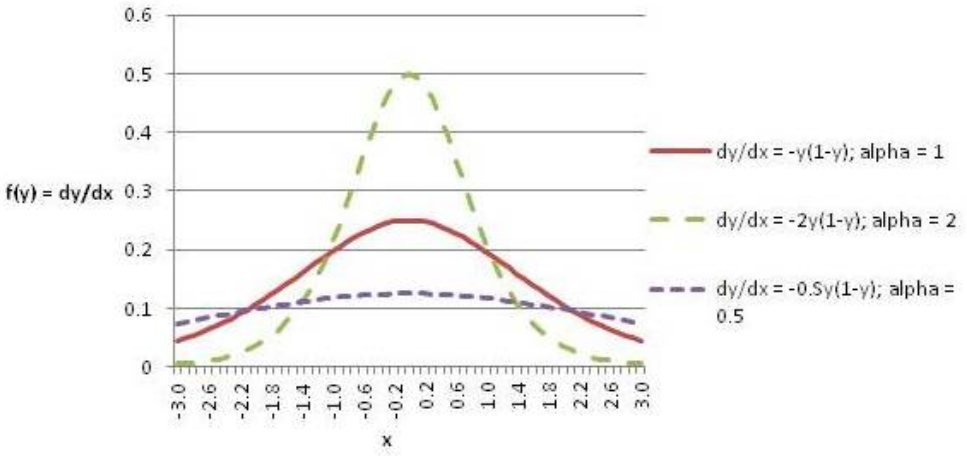

Figure 5.5: The derivative of the sigmoid transfer function .

We'll need to compute the derivative of the transfer function as part of the backpropagation algorithm. We'll do this now, and mentally stash the results for future use.

The derivative is computed as:

$$\begin{aligned} \partial y/\partial x &= -(-\alpha)(1 + exp(-\alpha x))^{-2} exp(-\alpha x) \\ &= \alpha(1 + exp(-\alpha x))^{-2} exp(-\alpha x). \end{aligned} \tag{5.2}$$

As a side note, those of you who are mathematically inclined will note that we've been using the terminology of "partial derivatives" throughout, when really, there is only a single variable (x) in the equations that we have just used. The reason for using the partial derivative nomenclature is that we'll shortly be deriving the entire backpropagation algorithm, which makes extensive use of the chain rule, and which also is expressed in terms of partial derivatives. Thus, we're just setting ourselves up for an easier usage later.

We want to simplify the results that we've just obtained, and to do so, let's first study a substitution. We write an equation with the same form, but with a simpler variable; we'll let $c = exp(-\alpha x)$. Thus, our simpler equation reads

$$\partial y/\partial x = (1+c)^{-2}c = \frac{c}{(1+c)^2}. \tag{5.3}$$

Note that we've dropped the leading coefficient α; again, that's just so we can concentrate on working with the form of the equation; we'll put it back in before we're done.

The mathematics of this equation are such that (if you know what you're doing in advance), you can express the derivative of the transfer function in terms of the original transfer function. To do this, we first note that the denominator of the original transfer function is now squared, compared to the original, as was shown in Eqn. 5.1.

As an intermediate step, we rewrite the equation so that the denominator is in two separate terms

$$\partial y/\partial x = \frac{1}{1+c} * \frac{c}{1+c}. \tag{5.4}$$

The first term on the right-hand-side is indeed the original transfer function; recall that

$$y = \frac{1}{1+exp(-\alpha x)} = \frac{1}{1+c}. \tag{5.5}$$

Thus, we'll simply substitute y for the first term on the right in Eqn. 5.4 to obtain

$$\partial y/\partial x = y * \frac{c}{1+c}. \tag{5.6}$$

Our job is half-done; we now want to express the last term on the right of Eqn. 5.6 in terms of y (the original transfer function), and not c. To do this,

we rewrite the numerator of this last term, both adding and subtracting 1; because we're not changing the total value of the numerator, this is allowed.

$$\partial y/\partial x = y\Big[\frac{1+c-1}{1+c}\Big]. \tag{5.7}$$

Now we can split the resulting last term on the right-hand-side into two parts:

$$\partial y/\partial x = y\Big[\frac{1+c}{1+c} - \frac{1}{1+c}\Big]. \tag{5.8}$$

We simplify the first term within the brackets, and for the second, we notice that we have once again obtained the expression for y, so that we obtain

$$\partial y/\partial x = y\big[1-y\big]. \tag{5.9}$$

Finally, we re-introduce the term α, which we had dropped earlier in order to focus on the simplifications. (This was when we went from Eqn. 5.2 to Eqn. 5.3.) This gives us our final result

$$\partial y/\partial x = \alpha y\big[1-y\big]. \tag{5.10}$$

We've now obtained the partial derivative of the transfer function in terms of its dependence on its (single) variable, x.

The derivative of a function is its slope. Since our transfer function is monotonically increasing, its slope is always positive. Thus, like the transfer function itself, we expect the derivative to also be consistently positive.

Thus, we expect that the derivative of transfer function will be approximately bell-shaped; it will have very small positive values to the far left and the far right, and achieve a value of 1 in the center.

We notice that it behaves exactly as we predicted.

5.6 Impact of a Bell-Shaped Transfer Function Derivative

In the previous section, we went through the math to obtain the derivative of the transfer function, and found that it was bell-shaped. We also identified the impacts of this bell-shaped function in neural network operation; we found

that it "squished" (author's term) the output into a reasonable range, e.g., between 0 and 1. (Instead of from negative to positive infinity.)

However, the impact of this "squishing" is that the *derivative* of this function is approximately 0 when the inputs to the transfer function are very numbers; either large positive or large negative. Further, just as the output of the transfer function itself was not much different when the inputs where, for example, 10 and 100, the values of the *derivative* of the transfer function are also very small, when there significant differences between the inputs (but both inputs are large; either large positive or large negative). This has a very substantial impact on training the neural network.

We'll postpone further discussion on this until the end of the next chapter, because we want to see how the transfer function's derivative plays a role in the training algorithms. Then, it will become very clear that - although the training method that we'll study (backpropagation) is indeed capable of training a neural networ, it has limitations. These led to the "neural network winter" of the late 1990's and early 2000's, and prompted the search for a method that would overcome these limitations.

That method, using the Boltzmann machine, has been the basis for *deep learning*.

However, that part of the story will wait until much later in this book.

Bibliography

[1] M. Jordan, "Why the logistic function? a tutorial discussion on probabilities and neural networks," tech. rep., Massachusetts Institute of Technology, 1995.

[2] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proc. ICML'10; Proc. 27th International Conference on Machine Learning*, (Haifa, Israel), pp. 807–814, June 21 - 24 2010.