Non-Linear Models

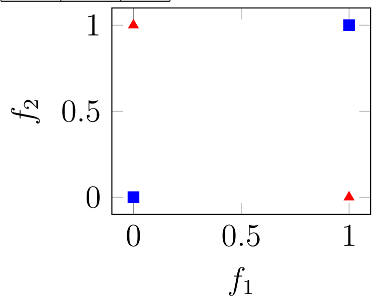
Jonathan May

September 6, 2024

1 Why Nonlinear Models?

The linear models we introduced appear to be very flexible, however they are limited in what they can capture. Specifically, because the equation $\theta \cdot f(x, y)$ is *linear*, classification cannot be successful if the data points, when plotted in their feature space, cannot be divided by a line (or, more generally, a hyperplane). The classic example of this is the *xor problem*. Consider this data:

f_1	f_2	y
1	1	a
1	0	b
0	1	b
0	0	a



This 2-label data set is class 1 iff binary features f_1 and f_2 are both on or both off and is class -1 otherwise. Try to draw a line that separates the data. It of course can't be done. You could of course introduce a new feature XOR(f_1, f_2) that explicitly captures this relationship and then the data would be linearly separable. But in general you don't know which combinations of features yield separability.

You could try a transformation that makes combinations of the weights. Define weights w_{11}, w_{21}, b_1 to map from the old feature space to a new feature g_1 and w_{12}, w_{22}, b_2 to map from the old feature space to a new feature g_2, such that

$$
\begin{aligned}
g_1 &= w_{11} f_1 + w_{21} f_2 + b_1 \\
g_2 &= w_{12} f_1 + w_{22} f_2 + b_2
\end{aligned}
$$

Let's use these as the weights:

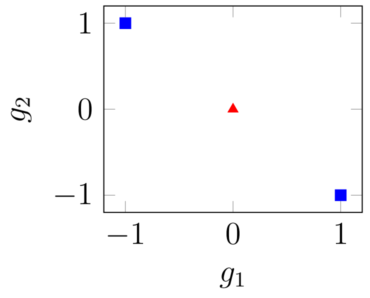
$$\begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 1 & -1 \end{bmatrix}$$

and

$$\begin{bmatrix} b_1 & b_2 \end{bmatrix} = \begin{bmatrix} -1 & 1 \end{bmatrix}$$

(It's no accident I set these up as a matrix)
That yields:

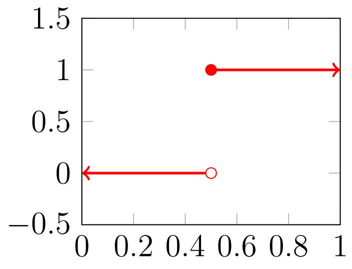
g_1	g_2	y
1	-1	a
0	0	b
0	0	b
-1	1	a



It's still non-separable! This should be no surprise; all a linear transformation can do is scale, transpose, and rotate the points; it can't distort them in a way that allows separability. Consider (cf. 7.2.1 in JM Aug 2024 ed.):

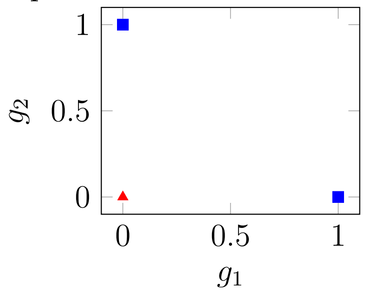
$$\begin{aligned} f &= W^{(0)}x + b^{(0)} && \text{(let's say the features were from a linear transform too)} \\ g &= W^{(1)}f + b^{(1)} && \text{(the above transformation in compact form)} \\ &= W^{(1)}(W^{(0)}x + b^{(0)}) + b^{(1)} \\ &= W^{(1)}W^{(0)}x + W^{(1)}b^{(0)} + b^{(1)} \\ &= W'x + b' && W' = W^{(1)}W^{(0)} \text{ and } b' = W^{(1)}b^{(0)} + b^{(1)} \end{aligned}$$

Still linear! So we'll multiply by a non-linear step function:



g_1	g_2	y
1	0	a
0	0	b
0	0	b
0	1	a

Separable!



The point of nonlinear transformations is to enable *recombinations* of features. We can make a linear combination of the new features and apply a nonlinearity to get yet another recombination. This can be done as many times as needed. What's nice about this is that *we don't need to specify complicated features* anymore – if we choose weights properly and use enough layers, we can capture any combinations of the input data.

1.1 Obtaining the weights

In logistic regression and perceptron, we used *gradient descent* of the loss on training data to set weights. We can use the same approach here, though the step function, being non-differentiable, isn't an appropriate nonlinear *activation function*, so we'll use a similarly shaped function that is differentiable at every point. First, let's define the model and the loss. Let $\mathbf{f}, y$ be the input feature vector of the input[1] $(1 \times d)$ and its label (a string) from a finite set of m labels. Let $\mathbf{H}$ of dim $(d \times v)$ be the weights matrix, and $\mathbf{b_H}$ be the bias vector[2] of dim $(1 \times v)$. The elementwise nonlinear activation function is $g()$. Thus to get the transformed vector (or 'hidden' features...or even 'hidden vector') $\mathbf{h}$:

$$\mathbf{k} = \mathbf{fH} + \mathbf{b_H}$$
$$\mathbf{h} = g(\mathbf{k})$$

What are d and v? That's up to you to some degree. $\mathbf{f}$, in particular, can be any features, such as the features we used in perceptron and logistic regression, but usually they're simply an arbitrary number of uninterpretable features tied to the vocabulary of the input $\mathbf{x}$.[3] For now assume they are given. If it helps, assume they are the features from the linear model walkthrough, i.e. the three-feature vector $(2, 1, 0)$.

[1]For those who are ready to jump to RNN, Transformer, etc., note that we're still using a fixed set of arbitrarily defined features. Even word embeddings will be introduced at the end.

[2]When to use a bias term? I don't know, and I see different formulations do different things. For example, E ch. 3.1 uses bias in both hidden and output layers, though he hides the bias term in the hidden layer. JM (Jan 2022) p.140 say "some models don't include a bias...in the output layer" and follow suit. I will use it in both places, explicitly.

[3]In other words, they are *word embeddings*, but we'll get to that shortly.

1.2 Nonlinear activation functions

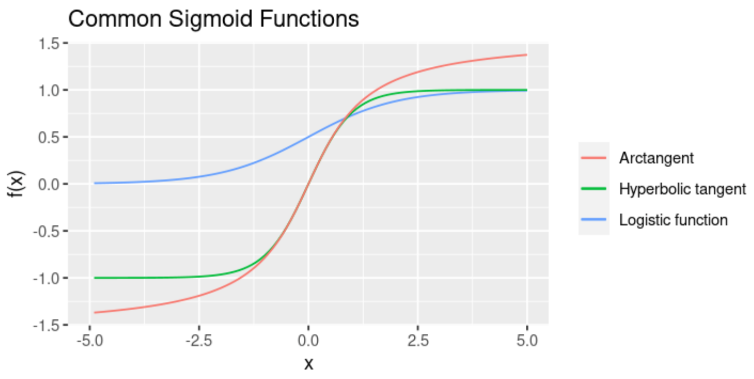
What about g? Before the 90s, the *logistic sigmoid* (often simply called 'sigmoid') function was used:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

In the 90s and 00s, neural network people recommended *hyperbolic tangent* or 'tanh', another sigmoid function, with a range from -1 to 1:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

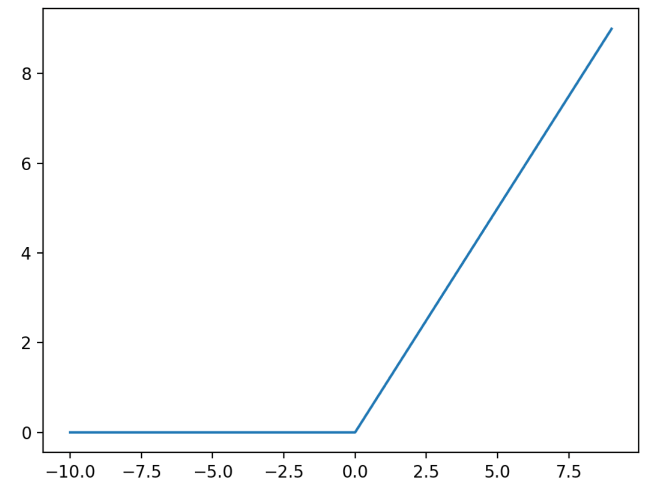
Here they are with the inverse of tanh (arctan) thrown in for good measure:[4]



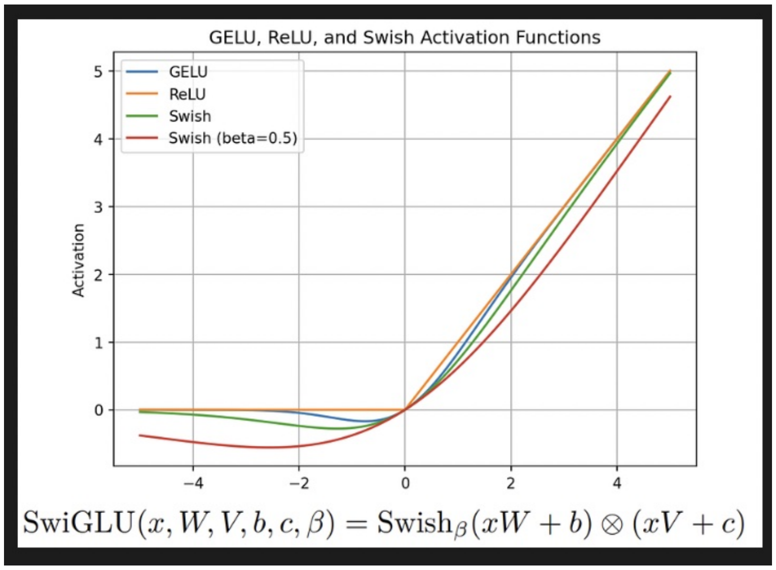
These resembled activation functions in neurons, the biological basis of ANNs. But one problem with these functions, especially as we started exploring deeper networks (i.e. repeated alternations of linear transforms and nonlinear activations) is that they *saturate*, i.e. the value goes to the extreme, where the slope is near zero, and then very little learning takes place. A nonlinear function that is a lot like a linear function but is still nonlinear is desirable. Enter the Rectified Linear Unit or ReLU:

$$ReLU(x) = \left\{ \begin{array}{ll} 0, & \text{if } x < 0 \\ x, & \text{if } x \geq 0 \end{array} \right\}$$

[4]Pic from `https://deepai.org/machine-learning-glossary-and-terms/sigmoid-function`



The gradient of ReLU is very easy to work with, and it turns out this function works very well in practice. Variants are sometimes used (Leaky ReLU: a small non-zero gradient is used for negative values; GELU: interpolation with a Gaussian distribution (erf) used in GPT1–3; Swish: a sigmoid with a hyperparameter; SwiGLU: combination of Swish and GELU used in LLaMa).



$$\mathrm{SwiGLU}(x, W, V, b, c, \beta) = \mathrm{Swish}_\beta(xW + b) \otimes (xV + c)$$

GELU on left, and Swish on right

1.3 Getting an output label

We now need to convert into the output space, which should be equal in length to m. For sentiment, let's assume the dimension is 2; positive and negative. We can just use a linear transformation for that, getting us the *logits*. Then, we use softmax again to get the probability of each output.

$$\mathbf{z} = \mathbf{hU} + \mathbf{b_U}$$
$$\mathbf{o} = \text{softmax}(\mathbf{z})$$

The loss ℓ is again the cross-entropy loss H,[5] which is defined for one data item $(\mathbf{f}, y)$ as

$$H_{\mathbf{f},y}(p,q) = -\sum_{y' \in \mathcal{Y}} p(y'|f) \log q(y'|f)$$

where the distribution $q(y'|f)$ may be represented by $\mathbf{o}$ and the true distribution $p(y'|f)$ is taken to be one-hot at y,[6] reducing H to $-\log(o_y)$.

Thus, ℓ ends up being

$$\ell = -\log(\mathbf{o}_y)$$

i.e. the negative log of the probability of the correct answer (denoted o_y to note that member of $\mathbf{o}$ corresponding to choice y).

Having calculated ℓ, we update each set of parameters $(\mathbf{H}, \mathbf{b_H}, \mathbf{U}, \mathbf{b_U})$ by the opposite of the gradient of ℓ with respect to that variable, i.e:

$$\mathbf{H} \leftarrow \mathbf{H} - \lambda \partial\ell/\partial\mathbf{H}$$
$$\mathbf{b_H} \leftarrow \mathbf{b_H} - \lambda \partial\ell/\partial\mathbf{b_H}$$
$$\mathbf{U} \leftarrow \mathbf{U} - \lambda \partial\ell/\partial\mathbf{U}$$
$$\mathbf{b_U} \leftarrow \mathbf{b_U} - \lambda \partial\ell/\partial\mathbf{b_U}$$

where λ is a learning rate. Now how are these partials determined? We start at the loss equation itself and use simple calculus:

$$\ell = -\log(o_y)$$
$$\partial\ell/\partial o_y = -1/o_y$$

Now consider the definition of o_y itself; we can use the chain rule and the local derivative of o_y with respect to z, though softmax is a slightly tricky function to take a derivative of:

$$\partial\ell/\partial z = \partial\ell/\partial o_y \times \partial o_y/\partial z$$
$$o_y = \frac{\exp(z_y)}{\sum_i \exp(z_i)}$$

To calculate $\partial o_y/\partial z$ we will make use of the derivative rule for quotients:

[5] Not to be confused with matrix $\mathbf{H}$

[6] Is this realistic? No! But it's convenient. Sometimes 'label smoothing' is used to make this more realistic; I might mention it.

$$(\frac{a(x)}{b(x)})' = \frac{b(x)a(x)' - a(x)b(x)'}{b(x)^2}$$

It is helpful to consider the application of this rule to $\partial o_y / \partial z$ in two cases: when $i = k$ and when $i \neq k$. Remember that even though o_y is a scalar, z is a vector, so we're calculating $\partial o_y / \partial z_i$ for every member z_i of z.

$$\begin{aligned}
[\partial o_y / \partial z]_{i \neq y} &= \frac{(\sum_{i''} \exp(z_{i''}) \times 0) - (\exp(z_y) \exp(z_i))}{(\sum_{i'} \exp(z_{i'}))^2} \\
&= -\frac{\exp(z_y)}{\sum_{i'} \exp(z_{i'})} \frac{\exp(z_i)}{\sum_{i'} \exp(z_{i'})} \\
&= -o_y o_i \\
[\partial o_y / \partial z]_y &= \frac{\sum_{i''} \exp(z_{i''}) \exp(z_y) - \exp(z_y)^2}{(\sum_{i'} \exp(z_{i'}))^2} \\
&= \frac{\exp(z_y)}{\sum_{i''} \exp(z_{i''})} \frac{\sum_i \exp(z_i) - \exp(z_y)}{\sum_{i'} \exp(z_{i'})} \\
&= o_y(1 - o_y)
\end{aligned}$$

Now we can multiply $\partial \ell / \partial o_y = -1/o_y$ with $\partial o_y / \partial z$ to get $\partial \ell / \partial z$:

$$\partial \ell / \partial z = \begin{cases} o_y - 1 & i = y \\ o_i & \text{otherwise} \end{cases} \tag{1}$$

Implementation note! Once you have o, if you represent the truth as a one-hot embedding T, then $\partial \ell / \partial z = o - T$. Try to see why!

We next continue on down to find the gradient of ℓ with respect to U and b_U, which are actual parameters we want to learn. We use the definition of z in terms of these variables and what we have previously learned:

$$\begin{aligned}
\partial \ell / \partial U &= \partial \ell / \partial z \times \partial z / \partial U \\
\partial \ell / \partial b_U &= \partial \ell / \partial z \times \partial z / \partial b_U \\
z &= hU + b_U \\
\partial z / \partial U &= h \\
\partial z / \partial b_U &= 1
\end{aligned}$$

We can simply multiply $\partial \ell / \partial z$, which is the complicated value in Equation 1, by either h or (the vector) 1, as noted above. Here, it's worth noting that we want to get the shapes of our gradient matrices right and that we want to deal with *batches* of training samples properly.

Imagine that we are updating parameters after seeing one training instance of a two-way classification problem ($m = 2$) with three features ($d = 3$) and 50 hidden units ($v = 50$).[7]

[7] Typical values would be in the hundreds for d and v and in the thousands for m.

Then, $\partial\ell/\partial z$ is a (1 x 2) vector. h is a (1 x 50) vector, and we want to update U, which is a (50 x 2) matrix. Thus we take $h^T \times \partial\ell/\partial z$ to get the right shape. However, note that in general, we do not update after a single training instance; rather, there may be some t items in the *minibatch*. So in fact $\partial\ell/\partial z$ is a (t x 2) matrix and h is a (t x 50) matrix. $h^T \times \partial\ell/\partial z$ still yields a (50 x 2) matrix but it is actually the sum of t individual loss calculations. The point of batch updating is to take a per-item average. Thus, the proper update for U is to subtract (the learning rate times) $\frac{h^T \times \partial\ell/\partial z}{t}$. Similarly, to update b_U, it is important to actually multiply $\partial\ell/\partial z$ by a length-t ones vector, which amounts to summing each dimension of $\partial\ell/\partial z$ along the batch axis, then divide by t.

If you've gotten this far, the rest should be straight-forward. We will need $\partial\ell/\partial h$, which is of course $\partial\ell/\partial z \times \partial z/\partial h$; the former term is in Equation 1 and is (t x 2), the latter is simply U, which is (50 x 2). We calculate as $\partial\ell/\partial z \times U^T$ to get a (t x 50) result for $\partial\ell/\partial h$.

We can now move on to the hidden layer; let's assume g is ReLu.

$$\begin{aligned}
\partial\ell/\partial k &= \partial\ell/\partial h \times \partial h/\partial k \\
h &= \text{ReLU}(k) \\
\partial h/\partial k &= \begin{cases} 1 & k \geq 0 \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\partial\ell/\partial H &= \partial\ell/\partial k \times \partial k/\partial H \\
\partial\ell/\partial b_H &= \partial\ell/\partial k \times \partial k/\partial b_H \\
k &= fH + b_H \\
\partial k/\partial H &= f \\
\partial k/\partial b_H &= 1
\end{aligned}$$

We update H, a (3 x 50) matrix with -$\partial\ell/\partial H$; The dimensions of $\partial\ell/\partial k$ are (t x 50), the dimensions of $\partial k/\partial H$ are (t x 3); thus we form $\partial\ell/\partial H = \frac{(\partial k/\partial H)^T \times \partial\ell/\partial k}{t}$. Similarly, we update b_H, a (1 x 50) vector with $-\partial\ell/\partial b_H$; we multiply $\partial\ell/\partial k$ by a t-length ones vector, which sums its values along the t-sized axis, then divide by t.

1.4 Word embeddings

Previously, we let $\mathbf{f}$, with dimension d, represent a set of arbitrary features. A more common approach is to instead use a fixed sequence of some n (let's say 20) words and represent each word in the vocabulary by an e-dimensional feature vector. This fits in nicely with our set of equations. Let E be a $|V| \times e$ matrix (often called an *embedding table*). Informally, we assign an index for each word in the vocabulary from 1 to $|V|$. Let the input be $j_1, j_2, ...j_n$ where each j_i is a *one-hot vector*, i.e. if j_i represents 'salamander' and the index for that word is 48, then $j_i = 0, \ldots, 0, 1, 0, \ldots, 0$ consisting of 47 0s, a 1, and then 49,952 0s. Then we redefine x as $Ej_1; Ej_2; \ldots; Ej_n$, a ne-length vector. Ej_i can be thought of as an 'embedding' of 'salamander' in $e << |V|$-space. Backpropagation is extended to update E as well[8]. We'll next look at why these embeddings are interesting in their own right.

[8]There is generally no bias term for the word embeddings