**Neural Networks**

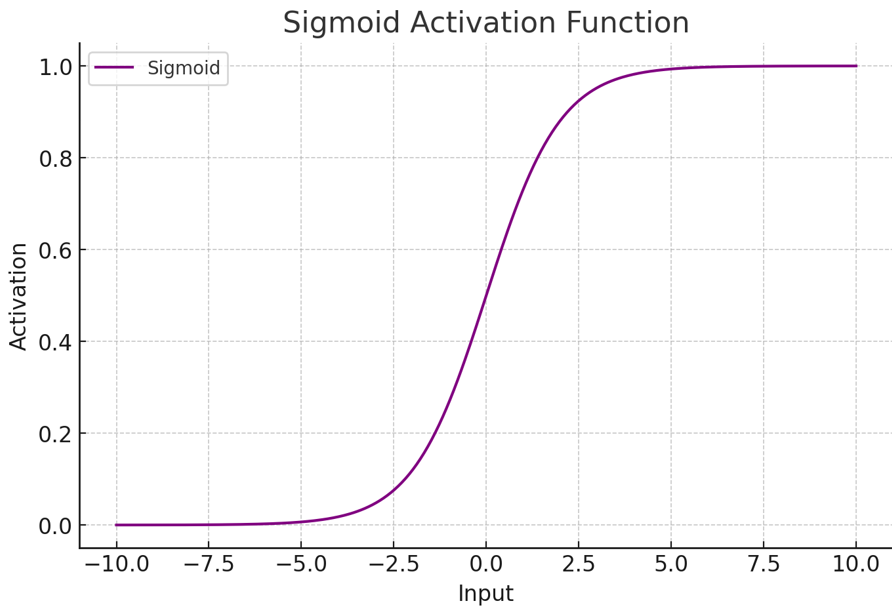
On the other hand, we can use neural networks to model classification schemes. For example, we’ll recall one of the data tables we’ve used before.

Table

Description automatically generated

We can turn any regression network into a binary classification network by adding a sigmoid function to the end of the output.





The activation functions we use for the other A’s to the left don’t have to be sigmoids. They are often one of the regression activation functions covered earlier. Codebasics guy says that the hidden layer activation functions are better served by the tanh activation function.



But then he says even better is Arelu or Asoftplus, actually. A simple perceptron classification network would be:

A diagram of a diagram

Description automatically generated

The output of this network would be:



which we’ll note is just a simple logisistic regression function. A more complicated one is:

A diagram of a network

Description automatically generated

(Networks don’t have to have descending orders of connections – I just randomly drew it that way) The output of this network would be just as delineated in the previous file for the case of regression.



where,



Presumably, we take the cutoff probability between ‘yes’ classifications and ‘no’ clasifications to be f = 0.5. But the correct choice would be pinned down by cross-validation. To find the optimal weights and biases, we need to know the loss function. And this is as before. We use log-loss function:



where i denotes the row, Yi = 0/1 is the outcome, and f(**x**i) is the predicted probability (**x**i is a vector variable **x**i = x1i, x2i, x3i).

We can generalize our neural network to accommodate outcomes which are not strictly binary. Perhaps an outcome could be ternary – could rate a movie as ‘good’, ‘fair’, ‘poor’. And we could modify the neural network above to accommodate this. We’d start with a regression-like network that outputs three variables. But then we’d tack on a so-called ‘softmax’ activation function, instead of a sigmoid activation function.

A diagram of a network

Description automatically generated

The softmax activation function kind of a generalizaion of the sigmoid activation function. It takes in n arbitrary real number values, and outputs these values to the range (0,1). And it preserves the relative order of these numbers, since if y2 > y1, then ey\_2 > ey\_1. Well, it is in fact what we use for the logistic function (see notes) when we have more than two outputs.



Anyway, since the numbers are scaled to (0, 1), we can interpret them as representing the probability of the input being associated with any of the classes f1,2,3,…n. Note there is nothing super special about the σsoftmax function. We could use another other that makes the same mapping. The mathematical output of our illustration would be (f is now a 3 element column vector f = (f1 f2 f3)T):



where,



and X = (x1, x2, x3)T. Again, A acts row by row. But…now we have to update the loss function to incorporate more than just two (1/0) possibilities. So…first let’s rewrite our LL in a form that makes the generalization more plausible. So to start, we have:



Now let’s call f(xi) = f1(xi), the calculated probability of a yes outcome. And we’ll write 1 - f(xi) = f2(xi) is the probability of a no outcome. And let’s write Yi = p1(xi), the probability of a yes outcome, and then 1 – Yi = p2(xi) will be the probability of a no outcome. p1(xi) and p2(xi) are either 1 or 0 of course. In this language we can say,



It then seems plausible that if we had 3 possible outcomes, with associated probabilities p1,2,3(xi) and fitted probabilities f1,2,3(xi), that we could generalize our loss function to:



And the generalization to, say, m possible outcomes it evident. Note p1,2,3(xi) will be 1 for one of the outcomes, and 0 for the other two. Since two of the outcomes is always zero, we can simplify our expression. Let the outcome which is certain to occur be denoted with the subscript outcome, and the other two with not-outcome. Note outcome could be any *one* of 1, 2, 3, and not-outcome would be the other two. Then we can write:



So we have:



This formula would apply equally well to the previous binary output case (case with single output node).

**Interpretation?**

On the CodeBasics website, I saw a nice video on interpreting neural networks. Consider we’re trying to ascertain the identity of an animal, and we have, say, three options. The relevant features might be: nose, eyes, mouth, torso, legs, represented by numbers, for the sake of discussion.

A diagram of a machine

Description automatically generated

A neural network with a hidden layer might function like this. The Nose, Eyes, Mouth features might combine to determine something about the Face, like the identity of the animal based on just the Face. And the Torso, Legs features might combine to determine something about the Body, like the identity of the animal based on the Body. And then the the Face and Body results could be combined to make an overall determination of the identity based on assessments based on Face and Body alone. The weights on the Nose, Eyes, Mouth lines, say, would be proportional to the relative importance of that feature to ascertaining the identity of the animal based on the Face. The bias in the Nose, Eyes, Mouth line might have something to do with overall, say, a priori, probability of it being one animal vs. the other, regardless of any feature information. And similarly with the Torso and Legs features apropos Body.

And it could be that be information about the Mouth actually has something do to with the determination based on the Body. In that case, we’d add another line, connecting Mouth and Body,

A diagram of a machine

Description automatically generated

Anyway, that’s just an interpretation. But the actual workings of a neural network are a bit more mysterious I think. And people have said that often times the hidden layer has no obvious physical meaning.

**Example**

So consider this data table, which we’ve seen before of course.

|  |  |  |  |
| --- | --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Age** | **Loves Cool as Ice** |
| Yes | Yes | 7 | No |
| Yes | No | 12 | No |
| No | Yes | 18 | Yes |
| No | Yes | 35 | Yes |
| Yes | Yes | 38 | Yes |
| Yes | No | 50 | No |
| No | No | 83 | No |

I did a classification perceptron with three input nodes, one activation function in hidden layer, and one output node. Yes/No was encoded as 1/0. Used the sigmoid activation function of course.

A diagram of a diagram

Description automatically generated

and got these results,

A screenshot of a computer program

Description automatically generated with medium confidence

So fitting function is:



Can see the fitting function does a good job fitting the data, which is fairly impressive given we have only four fitting parameters. It’s interesting that the neural network is so much simpler than the decision tree. But this probably isn’t always the case. At the very least, we’ll note that there is apparently no correspondance between number of hidden layers in a neural network and number of layers in a decision tree. One more interesting observation. When we worked this out via Decision Tree, we found that LS was most highly correlated with the outcome. And in our Perceptron, we see that LS has by far the highest weight. And we saw that in our Decision Tree, Age, and LP were the next most highly correlated features. Don’t know if it’s a coincidence, but Age and LP have the next highest (signed) weights in our Neural Network. So maybe we can think of the weight assigned to a feature as proportional to the information gain of that feature. And maybe the bias can be thought of as being proportional to the overall probability of the output being 1, regardless of feature.



**Example**

Consider this simpler example,

|  |  |  |
| --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Loves Cool as Ice** |
| Yes | Yes | No |
| No | Yes | Yes |
| Yes | No | No |
| No | No | Yes |

We could try to model with a simple perceptron,

A diagram of a block with a hidden layer

Description automatically generated

and we find,

A screenshot of a computer program

Description automatically generated with medium confidence

So the perceptron model works really well here.

**Example**

On the hand, there are cases where it won’t. Consider:

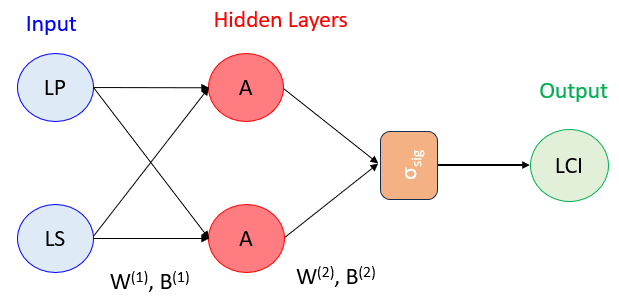
|  |  |  |
| --- | --- | --- |
| **Loves Popcorn** | **Loves Soda** | **Loves Cool as Ice** |
| Yes | Yes | No |
| No | Yes | Yes |
| Yes | No | Yes |
| No | No | No |

Then we get:

A screenshot of a computer program

Description automatically generated with medium confidence

Problem is that in the former case, the 1 and 0 outcomes are separable by a hyperplane in LP-LS space. But in the second example they are not. And a single sigmoid function can only make a distinction between points that can be separated by a single hyperplane. To get a good prediction in the second case we need a more sophisticated model. So I tried this one.



And I actually used A = σsig for the inner activation function, which isn’t really recommended. But then running the program, we get,

A screenshot of a computer program

Description automatically generated with medium confidence

So that works! And our output function is:



Another option is to create a two-valued prediction network. Could do,

A diagram of a software algorithm

Description automatically generated

Using the A = *softplus* activation function, I get this:

A screenshot of a computer program

Description automatically generated with medium confidence

So not good. But there are fewer d.o.f. in this network, so maybe not surprising. But if I do the following, kind of strange network in retrospect,

A diagram of a diagram

Description automatically generated

using A = softplus activation function again, I get much better results:

A screenshot of a computer program

Description automatically generated with medium confidence

which is really good! But it’s interesting that when I use the sigmoid activation function on this two-valued network, I get really crappy results.

**Exploring the Model and Hyperparameters**

At least in sklearn, there’s not a lot of wiggle room besides adjusting the network architecture and activation functions. Otherwise, we can change the learning rate, tolerance, and the solver.

**Hyperparameter: hidden\_layer\_sizes**

Let’s consider how this parameter affects things. The default is hidden\_layer\_sizes = (100,), which is for us, 2 input nodes, 100 hidden layer nodes, and 1 output node. But let’s do hidden\_layer\_sizes = (1,). Then this will just be a perceptron, which is basically a logistic regression function. This is its performance on the pure data sets. I also scaled the data first – seems to like that. And I’ve set the activation function to A(x) = x (activation = “identity”).

A diagram of a red and blue diagram

Description automatically generated A diagram of a diagram with red and blue dots

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence A diagram of red and blue circles

Description automatically generated

I would expect it to get the linear guy correct, and also to approximate the quadratic roughly the way it does. The triple class guys could’ve been better. And here it is on an impure (10% outliers) dataset,

A diagram of a red and blue diagram

Description automatically generated A diagram of a diagram with red and blue dots

Description automatically generated

A diagram of a graph

Description automatically generated with medium confidence A diagram of a diagram with circles and numbers

Description automatically generated with medium confidence

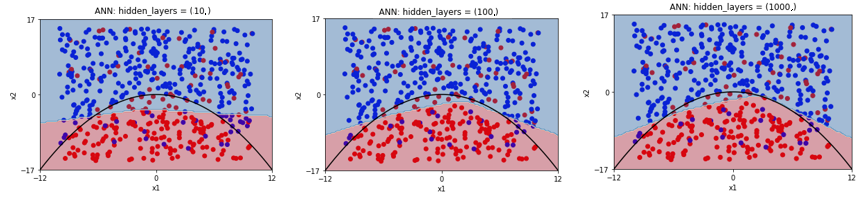
For the binary, performance is like the logistic regression guy, as expected. Will note that I had to decrease the tol → 10-5, and max\_iter → 105, to get it to converge in the linear case. For the trinary, I expected something like logistic regression, but I’m getting worse, even if I decrease the tolerance to superlow levels.

Now let’s do hidden\_layers\_size = (n1,) with n1 = 10, 100, 1000. And I’ll make the activation = “relu”, so that we can get non-linear behavior.

A diagram of a red and blue graph

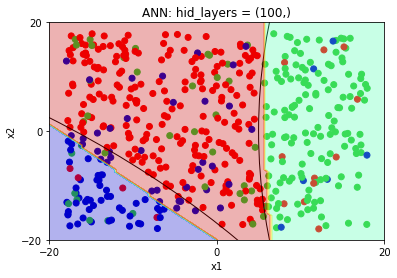
Description automatically generated

So increasing n1 doesn’t seem to confer much benefit. For the quadratic case, we get:



And again, gains seem to be minimal. For the linear triple case, we have:

A diagram of a diagram of different colors

Description automatically generated with medium confidence  A diagram of a diagram of a layer

Description automatically generated with medium confidence

Gains are minimal, past n = 100 I guess. Now let’s look at the triple circle case,

A diagram of a graph

Description automatically generated with medium confidence A diagram of a diagram of a cell

Description automatically generated with medium confidence A diagram of a diagram of a cell

Description automatically generated with medium confidence

And we get much better behavior in the circle case, even past n1 = 100. Now let’s go deep, instead of wide. Let’s consider hidden\_layers\_size = (10,10), (10,10,10), (10,10,10,10). We have:

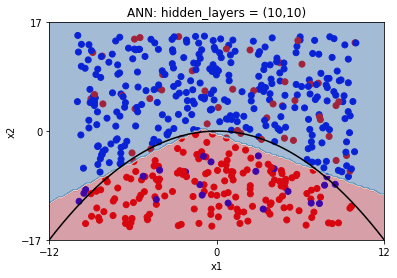
A diagram of a red and blue diagram

Description automatically generated A diagram of a red and blue diagram

Description automatically generated A diagram of a red and blue diagram

Description automatically generated

Our results are quite good. And there doesn’t seem to be any particular reason to go beynd (10,10) architecture. What about quadratic?

 A diagram of a red and blue dotted diagram

Description automatically generated A diagram of a diagram of a red and blue dotted diagram

Description automatically generated with medium confidence

These results are pretty good too. Would say by (10,10,10), we’re pretty much there. In fact, increasing it from 3×10 to 4×100 = (100,100,100,100) didn’t really make much improvement. I have read somewhere that nth order polynomial regression requires n+1 layers. And that seems to be the case for nth order polynomial classification boundaries too. So maybe a general rule is you need the layers to be a width a couple times greater than the input width. And you need the layers to be deeper than the order of polynomial curviness you’re looking to model. Now let’s look at the triple linear case,

A diagram of a graph

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated with medium confidenceA diagram of a graph

Description automatically generated with medium confidence

Get’s a little better with 10×10, but probably no need to go beyond. And now the triple-circle,

A diagram of a diagram of different colored circles

Description automatically generated with medium confidence A diagram of a diagram of a number of circles

Description automatically generated with medium confidence A diagram of a diagram of different colored circles

Description automatically generated with medium confidence

Not getting much improvement after 104, as it turns out. So looks like, again, for certain curvatures, there are diminishing returns for building deeper and deeper neural networks. FWIW, doing 1004 didn’t really do anything more. And 105 started noticeably overfitting.

**Hyperparameter: activation**

So all the previous examples (except the hidden\_layers\_size = (1,) guy) were using the relu activation function. What happens if we use something else? I’ll use *tanh*.

A diagram of a diagram of a diagram

Description automatically generated with medium confidence A diagram of a diagram of a red and blue circle

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated

A diagram of a layer of a cell

Description automatically generated with medium confidence A diagram of a diagram of a number of circles

Description automatically generated with medium confidence A diagram of a diagram of different colored circles

Description automatically generated with medium confidence

I’m surprised it did that well, honestly. Interestingly, the logistic activation function gave terrible results, with the specified tolerance I’ve been using (tol = default, or sometimes 10-5) Now let’s try the softplus activation function. This one isn’t available to sklearn’s library. So have to use TensorFlow. Actually had to increase the patience (see below) to p = 50, and increase the step size. Eventually I even had to turn off patience (eliminate the call back argument). Apparently softplus always likes to start in a local extremum, often a local minimum, and it takes a while to get out of it.

A diagram of a graph

Description automatically generated with medium confidence A diagram of different colored circles

Description automatically generated A diagram of different colored circles

Description automatically generated

**Hyperparameter: batch size**

The batch\_size default is 200, for sklearn, I think. And for TensorFlow it’s 32. And it’s the number of sampled rows of data that are used when updating the weights via sgd or Adam. The epochs are the number of times the algorithm runs through all of your data. So if you have 100 rows, batch\_size = 40, and epochs = 250, then you’ll update weights 3 times per epoch (once with the first 40 rows, again with the next 40 rows, and again, usually, with the next 20 rows), for a total of 3·250 = 750 times. Apropos TensorFlow, looks like often times you’ll want to increase the batch\_size above its default 32. Seems like it gets stuck in a false minimum a lot, which might happen if its not sampling enough of your data at once. But the sklearn and TensorFlow algorithms are supposed to shuffle the data after every epoch so that this kind of thing is less likely to happen. Here’s using the relu activation function and batch\_size = 50,

A diagram of a diagram of different colored circles

Description automatically generated with medium confidence A diagram of a graph

Description automatically generated with medium confidence A diagram of different colored circles

Description automatically generated

yeah.

**Hyperparameter: learning rate, epochs, min\_delta, patience**

The defaults for these parameters in sklearn is *α* = 0.001, *epochs/max\_iter* = 200, *min\_delta/tolerance* = 0.0001, and *patience/n\_iter\_no\_change* = 10. For TensorFlow it’s *α* = 0.001 (set in nn.compile method), *epochs* (set in nn.fit method) = no default, *min\_delta* (set in callbacks.EarlyStopping method) = no default, and *patience* (set in callbacks.EarlyStopping method) = no default.

*α* adjusts how much weights and biases are updated in the direction of the gradient after each batch is evaluated. *epochs* sets the number of iterations over which all the data is run through the algorithm (epochs×batches would be the number of times the weights and biases are updated). *min\_delta* sets the minimum value by which the estimated total loss function should diminish each epoch. If it fails to do so after a given epoch, then it has *patience* epochs from that point forward to diminish by that amount before training is terminated.

Consider the following blue loss function in parameter space. Then let’s discuss the influence of those parameters on our progress from A to F.

A graph of a graph

Description automatically generated with medium confidence

First, α, the arrow, has to be small enough to navigate the terrain seen here. If it is too large, then it might just skip completely over point F to point G. And then, since the sign of the gradient will have reversed, it might go right back to point A, or B or something. So α has to be small enough. Seems like α = 0.001 often does the trick. If the loss is jumping up and down kind of, then I’d think α is too large. If α is the right size, then I think the loss would be steadily decreasing or increasing. Even still, you’d probably want to get away with the largest α that you can, to expedite progress. So if loss decrements are really small, then might increase α.

If α is small enough, then *e*(pochs) has to be large enough. Often I’ve needed e = 1000, 10 000. Otherwise, we might just stop at point B, or point C, or E, for that matter. If the loss has been steadily decreasing for all epochs, then I’d guess we just need to increase *e*, to go, say, from B to C, or E to F. But if it’s been steadily increasing, then maybe decrease e, to go, say, from G to F. On the other hand, even if the loss is increasing, like when you’re going from C to D, you might want to increase the epochs to see if it’s just in a local, but not global minimum.

I guess in principle, just varying *α* and *e* should suffice, as long as you’re keeping track of the loss yourself. If loss is jumpy, then decrease α. If not dropping fast enough, increase α. If still dropping by end of training, increase e until it goes back up, and you’ve located the minimum. But you can also set δ = min\_delta, and p = patience to have TensorFlow monitor the loss. I guess you’d want δ = not too large, or you’d stop before you got to any local minimums, since as you get closer to a minimum, the decrement in the loss will drop further and further. On the other hand, you don’t want it too small, as, I imagine there are a lot of unpictured bumps and wiggles on the way from, say, E to F. And so you might in fact go up a little bit, or sideways, on your way to F. That would be a good reason to have the patience window set to p = not too small. Furthermore, you might not even want to get all the way to the minimum, as that might overfit your training data. But what if you’re trying to get over the hump, i.e., from C to F? Then you’d need a really large p (I had to increase p to 50 at one point, and at some point had to just turn it off because I kept bouncing around a local, but not true, minimum that I had to get out of). But that same really large p would probably allow you to far beyond F as well, maybe even up to G.

Might consider doing bagging classifier with ANN.