**Neural Networks: Regression**

Decisions in animal brains are made by neural networks. Apparently primary neurons take in some sort of stimulus input, which is then transmitted to an adjacent layer of neurons. These adjacent neurons can receive signals from many of the primary neurons simultaneously. Their superposition determines the response of the adjacent neuron. If higher than some threshold, it will fire; othewise not. And this adjacent layer of neurons will then send some signal to the next layer of neurons, etc., until an output/decision is finally rendered. Assuming this decision making process has some utility, we can try to make an artificial analogue. So neural networks seem to basically have the same versatility as decision trees. ChatGPT says they’re less prone to overfitting, and can model stuff better in general. But it might be biased. I’ve heard other people say that when data is heterogeneous, like in the table below, then decision trees often do better. Data is heterogeneous when it combines different categories of data, like age, sex, dosage. Homogeneous data would be something like all numbers, say stock prices on successive days, or all words, like the words in a sentence. In the homogeneous case, it makes more sense to model the output via linear combinations of the data variables, like 0.5×stock price on day one – 1.2×stock price day on two, as opposed to the heterogeneous case where we’d have something like 0.5×gender – 1.2×dosage. An advantage of regression networks, and linear/polynomial regressions too, is that they would tend to do better with fewer data points than decision trees I think. This is because all realistic functions in nature must be continuous/differentiable, I’d think, and linear/polymial/neural network regressions satisfy this minimum criterion, and so we cannot have huge leaps in output over just a small change in input.

Okay. Well let’s say we had a regression table,

Table

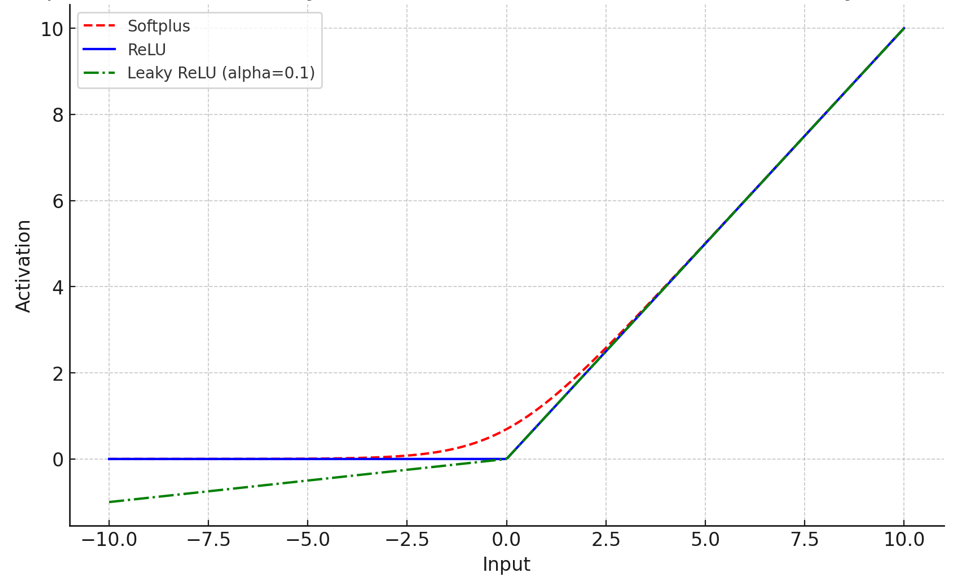
Description automatically generated

We can make a neural network that takes as input the values from these categories x1 = Dosage, x2 = Age, x3 = Sex (1,0), and outputs a numerical result, a prediction, f, for the Drug Effectiveness, y. The simplest neural network we can make is a *perceptron*, which looks like this:

A diagram of a diagram of a hidden layer

Description automatically generated

The set of input nodes corresponds to some row of the input data. Each input node corresponds to a given column, and outputs its value into the arrow. Each arrow carries a presumably unique weight (could be zero) which multiplies that value. Where arrows (could be just one arrow ‘meeting’) meet up at a node, we sum the values×weights and add a bias term (could be zero). This total value x = w(1)1x1 + w(1)2x2 + w(1)3x3 + b(1) is what goes into the activation node A. Then the activation node outputs A(x), where A is the activation function. There are three popular choices, plotted below:



The equations for these guys are:



The other one is (lrelu = leaky-relu):



(the 0.1 is negotiable) And I imagine that, especially for regression, we could make up other activation functions if we wanted to. For instance, I’ve noticed



works pretty well, but obviously we can only get linear behavior out of this one, whereas the previous guys will accomodate non-linear behavior. Anyway, so the output coming out of the activation function would be, vis a vis the network above:



And then the next line adds a weight and bias to this, so the total output going to f would be:



Note sometimes people don’t conventionally presume a weight and bias term going from the last (only, in this case) activation function to f. Adding the weight, bias seems to be the convention for SKLearn’s neural network stuff, whereas TensorFlow doesn’t do it this way. Then we would optimize the unknown parameters W, and B by minimizing the loss function, which we’d take to be the SSE, as usual:



where yi is the outcome pertaining to the ith row, and fi is the corresponding prediction. As a practical matter, we’d probably use gradient descent. Now some(often)times, such a simple neural network won’t be sophisticated enough to capture the complexity of the relationship between independent and dependent variables. So we can add more so-called ‘hidden layer’ nodes between the input and output. So we could have something like this:

A diagram of a network

Description automatically generated

Same basic rules as before apply here. But we’re going to put this in matrix form. The relationship between the values output by the first activation layer and the input layer is this. The values that are output of each node in the first activation layer are the activation function of the value input into that node. The value input into a node is the sum of the weights×values of the input layer of nodes, plus a bias. In other words, the four values coming straight out of the four activation nodes are:



Note the subscripts on the weights are to be read *backwards*. wab connects the ath node on the right to the bth node on the left. Kind of awkward, but this enables us to write this in matrix notation as:



where,



only remember that A acts on the matrix in a row by row manner, which is to say, e.g., A([x1, x2]T) = [A(x1), A(x2)]T. Then these output values are multiplied by weights and fed into the next layer of activation nodes, with a bias included for each node in the that next layer (i.e., where the arrows meet). And then they are fed into the activation function to get the next output. In other words, the two values coming out directly from the two activation nodes in the second activation layer are:



which we could write in matrix notation as:



where,



And then finally we come to the output. We have:



which we could write in matrix notation as:



where,



So for this neural network, the final output, in terms of the bare input, is:



Let’s work all of this out explicitly,



FWIW, we could also construct a regression network with multiple outputs,

A diagram of a network

Description automatically generated

The output would be:



where f is a three element column vector f = (f1, f2, f3), and the weights/biases are:



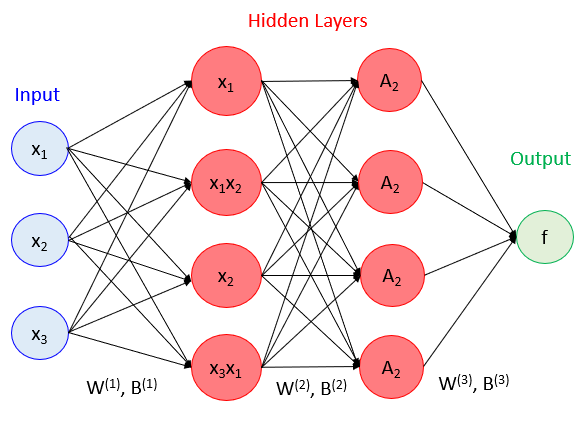
So basically, constructing a neural network for regression is just kind of a visual way of constructing a very complicated regression function – one which is a composite of a composite of a composite…of a composite of the activation function. I’m not sure why a composite of a composite, etc., of a function is a good prospective regression model. But whatever.

Parenthetical point: you might want to include a bias with each ‘wire’. This would make it a little cleaner mathematically and conceptually, but would introduce extra d.o.f., which shouldn’t be in any computer code, as by experience, this makes it much harder for it to converge to an answer. So if have two wires running into a node, could write w1x1 + b1 + w2x2 + b2 → w1x1 + w2x2 + b.

Another point: on the other hand, it might be helpful, diagrammatically, to add a little disconnected arrow pointing into each internal node and the final output node, to represent the bias that each node adds to the input. But that really clutters up the diagram, so I left it out.

**Interpretation?**

I talk about interpretation in the Neural Network Classification file, but I’ll say something here too.



It’s often the case that hidden layers have many more neurons than the input layer. We might think of the width of the layer as having something to do with the number of important features. And this might be more, or less, than the input layer. In the Classification file, I talk about how it can be less. Basically, it could be less if we can combine all the input variables into fewer relevant features. This is kind of like what PCA and NNMF, etc., do. And if we can combine into fewer features, we’d want to, to eliminate noise. But the hidden layer size can also be more than the input layer size. Recall the Categorical Regression file. In that file we find that to model all the slopes and things, we need to split certain features by other features, e.g., split Total\_Bill by Gender when modeling tips, to get Total\_Bill\_Male, and Total\_Bill\_Female. And this involved multiplying all columns by the categorical variable columns, extending our array of features to potentially a *lot* more than the starting dataframe. So likewise we might envision this expansion of the hidden layer as doing something similar. We might also compare to quadratic regression, which also required expanding the dataframe to include products of numerical variables. Generally, they say the width of the layers is proportional to how many features we need to model it properly. And the depth of the layers is proportional to how curvy the data needs to be.

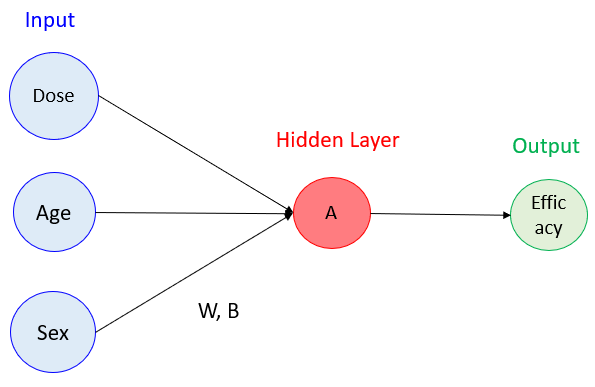
**Example**

For example, consider this regression table.

Table

Description automatically generated

I modelled it with a perceptron, with *no* weight/bias on the last leg.



I wrote a small program to use gradient descent to find the optimum parameters for a perceptron with 3 input nodes and an output node (no weights/bias on that last line going fro A to f in this case). Encoding Female/Male as 1/0, and using the softplus activation function, I found,

A screenshot of a computer program

Description automatically generated with medium confidence

So our f is:



Can see that our function is pretty good. But then, we’d expect it to be since we have only four data points and four unknowns. So we should be able to get an exact fit, more or less. Seems like we should also see from the weights and typical values of the variables, which variables are most important. If we wanted to do this ourselves, i.e., determine the best weights and bias, then we’d have to form the general regression function:



where x1,2,3 = dose, age, sex, and fill into the loss,



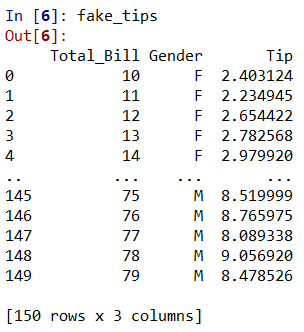
And take derivatives w/r to w1, w2, w3, and b, setting each expression to zero,



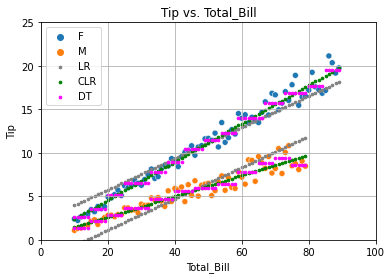
and solving for the unknown respective variables. Obviously this would have to be done numerically in practice – using Gradient Descent probably.

**Example**

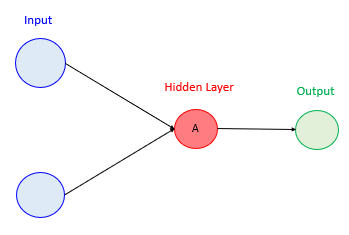
Might recall I did categorical linear regression on a fake\_tips dataset.



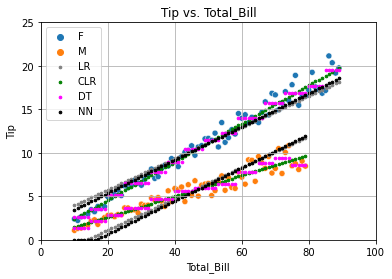
And we saw (by design) that the tip rate did depend on gender. And when we did machine learning models, we had to do categorical linear regression (green) to model it properly as ordinary linear regression (grey) couldn’t change slope based on category.



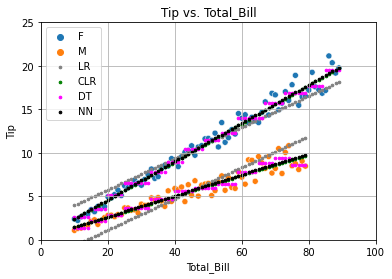
We can reproduce the linear model with a simple neural network with layers = [input=2, hidden=1, output=1].



We find,



But we can do better than this by expanding the hidden layer dimensions beyond 1 to, say, 10. This effectively increases the number of ‘features’ we’re using, like categorical linear regression does, and so allows the NN to properly distinguish slopes based on gender category. This is what we get for layers = [input=2, hidden=10, output=1] (and *lots* of experimentaion on batch\_size, alpha, epochs, etc.)



As we can see, now the NN regression overlays with the categorical linear regression just about perfectly. So take away is that expanding the hidden layers means we’re basically expanding the feature space. But should also say that the number of hidden layers in our NN was 10, whereas the number of ‘hidden layers’ in our categorical regression was 3 (or 4). And when I set hidden=3 or 4 in the NN, I didn’t get results better than hidden=1. So I’d say there is not an exact correspondance between ‘hidden layers’ in categorical regression and hidden layers in NN regression.

**Scaling**

Seems the gradient descent algorithms would *really* prefer that the data is scaled to between 0 and 1 (or so) first. So say we did this. Let X be our initial data, and f our initial target variable. Then let X´ = SX(X), and f´ = Sf(f) be our scaled variables. And our regression function for the scaled variables would be, say:



and we can put this in terms of the unscaled variables,



So we’d define a scaler for X and scaler for f. Do the regression. And then feed in the scaled Xtest, and then compute the reverse scaled ytest. Seems can often get away with just scaling the X data.

**Vanishing/Exploding Gradients Problem**

In practice it seems that convergence is an issue. Scaling data first helps and is a must. Also, using LeakyReLU activation functions, instead of ReLU or softplus, seems to help. But otherwise, the each time I run the program I seem to end up with possibly vastly different functions. This seems to be due to the fact that the initial weights and biases that the SGD or Adam optimizers use are random. But really, we should be converging to the same result regardless of the starting point. But often…this doesn’t seem to happen. So is neural net regression really useful?

Have noticed that it seems to stabilize a little if do architecture where there are many more hidden layer nodes than nodes in the input layer. So maybe something like, if doing two variable regression: 2, 20, 20, 1, instead of 2, 2, 2, 1. Let’s go back to:

A diagram of a network

Description automatically generated

The output is:



When optimizing the model, we’d use gradient descent, say. So we’d minimize SSE = Σ(fi-yi)2. The derivatives are, in matrix-y notation,



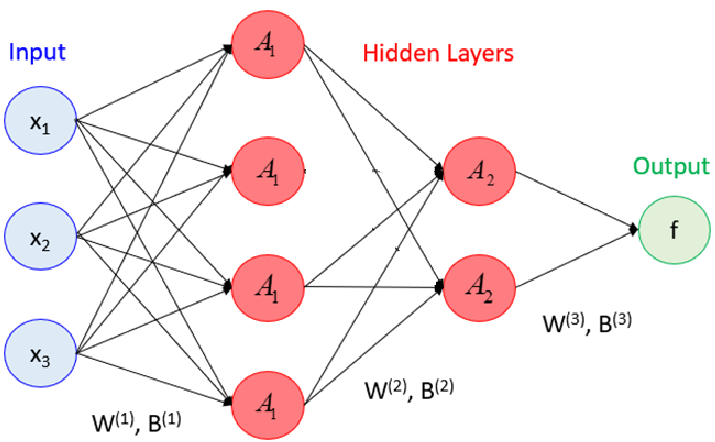
where [ ] is short for [W(2)A1(W(1)X + B(1)) + B(2)], and ( ) is short for (W(1)X + B(1)). And we’d have, from gradient descent:



If we number the hidden layers from left to right, then can see that the gradient for the weights W(m), biases B(m) in layer m depend on all the weights after layer m, i.e., to the right of layer m. So if those weights to the right are ever small, then the step size, α{ }, for W(m) and B(m) will be small too. And in fact, as can see from above, they get smaller the smaller m gets, i.e., the further the hidden layer is positioned to the left. And so there will be hardly any change in them. This is called a *vanishing gradient problem*. This might be why the model is more stable, when have large layers – because the weights/biases aren’t changing as much? On the other hand, if those weights, biases are large, then the step size for W(m), B(m) will be huge, and too large, as we won’t be able to fine tune the model. And the step sizes will get larger and larger the smaller m gets, i.e., the further and further the hidden layer is positioned to the left. This is called an *exploding gradient problem*. So we can see these two issues will be a problem when we have more and more layers in our network.

**Dying Neurons Problem**

This primarily affect RELU activation functions. When such a neuron’s input is pushed into negative territory, its output becomes 0. In regards the network above, if the second from the top neuron in the first hidden layer went dead, then we’d effectively have this network.



Let’s go back to the general formula for the output and cross out the terms that would be eliminated if that activation function dies.



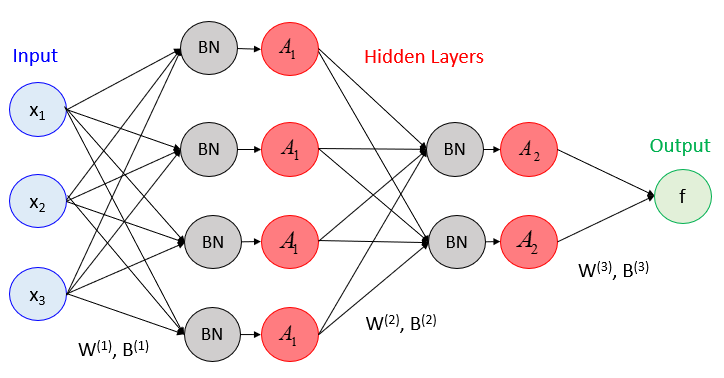
We see that once this happens, f no longer depends on w12(2), or w22(2) – the weights heading out of that A1. But also the weights and biases feeding into it, w21(1), w22(1), w23(1), b2(1) also get eliminated from f. Having been eliminated from the output, gradients w/r to these parameters will have been eliminated too. This is because, say we’re trying to minimize the SSE ~ (f – y)2, then to update the weights, biases, we’d do ∂SSE/∂w, ∂SSE/∂b, which numerically, would be accomplished by taking the present weights w and biases b, incrementing them slightly from their present values and calculating ΔSSE/Δw, ΔSSE/Δb. But if that A1 in question is outputing 0 presently, then it will still output zero (unless the input is right at the origin) for small deviations in w2,(1,2,3)(1) . And it will certainly still output zero for small deviations in w(1,2),2(2). Thus, small changes in these parameters will not change f, and so will not change SSE, and so the gradient with respect to them will be 0. Thus these parameters will not be updated, and we will continue to have the zero-output problem for that A1 until the very end of training. So basically, once they go dead, they stay dead.

**Some Remedies for Gradient Problem, Dying Neuron Problem**

Proper weight and bias initialization can mitigate these issues. One effective approach is called **Kaiming/He initialization**. In this approach, the initial weights (and biases?) of a hidden layer are drawn from a p.d.f. with average 0 and variance 2/n, where n is the number of inputs into the layer. The shape of the p.d.f. is typically normal or uniform, and I’m not sure it matters which. So if the input into a node is xi(2) = Σj=1n wijxj(1), then the variance of the input xi(2) would be var(xi(2)) = Σj=1n var(wij)[xj(1)]2 ~ nvar(w)[x(1)]2. Now of course this xi(2) and other in the same hidden layer (2) would then feed into the next layer (3). So the variance input xi(3) of the next hidden layer would be commensurately larger. So the variance grows to the right. But if we shrink the variance of our p.d.f. from which we draw the weights by 1/n, then we’d have:

var(xi(2)) ~ var(w)[x(1)]2. And at every layer, the variance would always be of around this order. Thus the initial inputs at every layer will be similarly distributed.

This will fix the problem in the short term, but deep into training, it will probably return. An effective way to forestall that problem is **batch normalization**. The approach is to put a batch normalization layer before every hidden layer. This standardizes the input into the activation functions, keeps it from getting super small, or super large.



It works like this, I think. The weights and biases in the network are initialized. And then the first batch of data is sent through. Each BN takes the data, xi(1) that successively comes to it in the batch and normalizes, scales, and shifts it according to:



And then it feeds this data xi(2) into the activation function A1,2 as applicable. For the first batch, I think the parameters are initialized as follows: mean μ = 0, std deviation σ = 1, scale factor γ = 1, shift factor β = 0. So the data is initially unchanged from xi(1). After the batch data has been run through the network, each γ and β, along with the rest of the weights and biases, are treated as learned parameters and the Loss gradient with respect to them calculated, and they are updated.

When the next batch is sent through, the data coming into BN is rescaled again according to:



γnew and βnew are the learned parameters calculated from gradient descent (Adam or whatever) in the last batch. μnew and σnew are the updated mean and standard deviation of the input data. μnew and σnew are calculated a little differently than one might think. There is a recurrence relation relating the new μ and σ to the previous μ and σ. It looks like this:



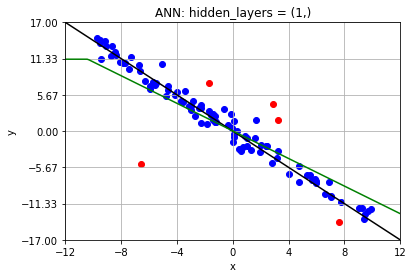
where μbatch and σbatch2 are the previous batch’s xi(1) input mean and variance, and α is a hyperparameter typically set to between 0.9 and 0.99. Next this scaled data is fed into the activation function neurons, and the loss calculated. Then as before, the weights, biases, γ’s, and β´s are updated. Then the process is repeated with the next batch. Each BN neuron will zero in on a different final μ, σ, γ, β, as each’s input will be different.

**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,), but let’s do do hidden\_layer\_sizes = (1,). I also scaled the data first – seems to like that. And I’ve set the activation function to A(x) = x∙θ(x) (activation = “relu”), which is the default anyway.

 A graph with red and blue dots

Description automatically generated

Had to increase max\_iter to 10000 for the linear guy. Now let’s do hidden\_layers\_size = (n1,) with n1 = 10, 100, 1000. And I’ll again make the activation = “relu”.

A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated A graph of a line graph

Description automatically generated with medium confidence

So we see that we get better results-ish as we increase n1. We basically remove that lip of the relu activation function. And so we’re basically doing straight linear regression. We could’ve switched out our activation from relu to identity (linear), and this would’ve been pure linear regression. But one problem we see is that it treats the outliers the same as the normal noise. How could we tune those guys out? For the quadratic case, we get:

A graph with red and blue dots

Description automatically generated A graph with a line and a dotted line

Description automatically generated with medium confidence A graph with a line and a line

Description automatically generated with medium confidence

Gain here is minimal. 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 graph with a line and a dotted line

Description automatically generated with medium confidence A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated

These results are a little funky. 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. 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. Interesting that it’s still, by and large, ignoring the outliers, which is good. I would wonder though, what kind of architecture would we need, for it to try to get the outliers too?

**Hyperparameter: activation**

So all the previous examples were using the relu activation function. What happens if we use something else? I’ll use *tanh*.

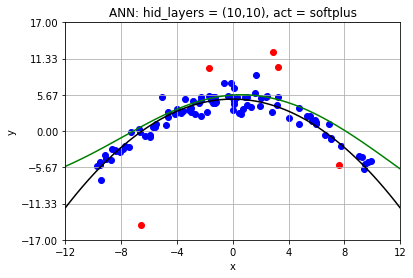
A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated

Looks a little better, but still. Interesting that it’s still, by and large, ignoring the outliers, which is good. I would wonder though, what kind of architecture would we need, for it to try to get the outliers too? 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. Note I had to increase the learning rate to α = 0.006, and run it for 5000 epochs.

 A graph with red and blue dots

Description automatically generated A graph with red and blue dots

Description automatically generated

The 10×10 looks the best, even though the other ones actually have slightly lower loss. The 10×10 would probably do better in cross-validation.

**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. Going back to the 10×10×10 architecture, with relu activation, and doing a couple different batch sizes,

A graph with a line and a line graph

Description automatically generated with medium confidence A graph with a line and a line graph

Description automatically generated with medium confidence A graph with a line and a dotted line

Description automatically generated with medium confidence

They look about the same. Maybe batch\_size = 50 is best. When using the tanh activation function, the difference was a little more dramatic, and could see that you probably don’t want to use batch\_size = data\_frame\_size = 100 (in this case). 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. For instance, it could sample the left half the data and try to approximate it by an upward line, and then the right half of the data, it try to approximate it via a downward line. And maybe it will just kind of see-saw back and forth between these options. But clearly we need a parabola, not a line, and it can only see that if it is looking at a large enough sample of the data. On the other hand, the sklearn and TensorFlow algorithms are supposed to shuffle the data after every epoch so that this kind of thing wouldn’t happen. But I have noticed that changing batch\_size has an effect on convergence, often times.

**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 decreasng 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. But that same really large p would probably allow you to far beyond F as well, maybe even up to G.

**Thoughts**

Might consider doing bagging regressor with ANN.

How could we enforce a regression constraint, like curve has to go through the origin? We can do this with the linear regression program, but how would we do with neural network?