**Neural Network Library Work Notes**

The key point of the library is using the matrix multiplication technique for calculating derivatives, at least up to third order but definitely don’t hard code that. Actually it’s 2nd+1st order, up to 2 in input variable and then 1 in model parameter.

I’d love to use Dr. Turchanin’s approach of describing an entire NN by a single weights matrix, but I believe that is inconsistent with the matrix multiplication technique. At least the technique was not derived that way, so the first implementation will not use it. Instead, each layer is a matrix.

For expediency, there is going to need to be a way for a user to indicate which derivatives they want available for a model, which will be used to limit (the pikj, aikj and more importantly) the gradient of derivatives calculation. Actually, the way is “autodetect” – there will be a class method to call a derivative or 2nd derivative with respect to the model inputs, that will be called from a loss function, and that call will enable a flag to calculate q, p, a, b, c, d, e, f. It will also check if it was previously enabled and if not it’ll kick off calculating those for all of the model layers; this catches the first behavior.

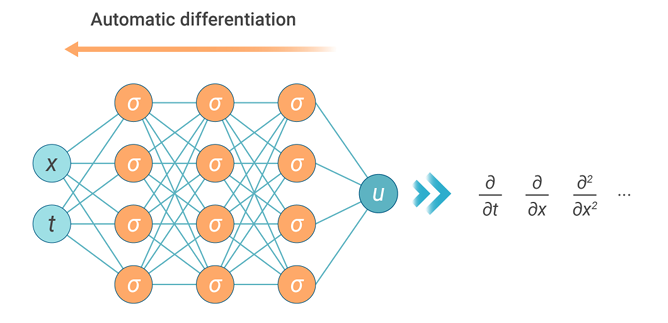


Figure from Wang Chen Zhang 2024, showing what I’m trying to change; the arrow is going to go in the other direction.

BTW, a matrix is row-column. Aij is the ith row and the jth column.

Model Class Data:

optimizer, calculate\_p\_list (and bdef), calculate\_a\_list (and c), nodes (by layer list), activation\_function (by layer list), wxyzpqvabcdef

Model Class Methods:

init, add\_layer, compile, set\_optimizer, apply\_gradient, load, save, ddx, d2dx2, ddw, evaluate, calculate\_xy, calculate\_data

How big are B and C?

*biklmnj* for each residual, so batch size (e.g. 1000 = 210) \* width (2e5) \* {1 per input whose first or second derivate is used} (2e1) \* width \* width \* layers (2e2), \* layers. Totals to 2e30. Using float16, each is a byte. So 1 GB. Not really unhappy with that. Does sound like a lot to calculate each epoch, but at least it’s not a TB or anything, and the earlier layers are largely unpopulated.

I know that if loss = (T’’(x)-0)2 then the gradient of the loss is 2T’’ dT’’/dw, but how does the model know the gradient of the loss?

One big design question is: do I make the inputs be layer 0, and include that 0th layer in all of qxyzqvpabcdef? I think I do, and I take advantage of the fact that the lists can be different size so the 0th layer is [] in most cases.

While the notation used in Expressions of Derivatives v2 has the layer being the last index, that is likely not how we want to build the data structures. The last index should be the residual index (for everything other than W, which is not residual-dependent). The first index should be the layer. Actually, it varies. Individual data structure considerations. Captured in v3.

W was from-to-layer. Now is layer-to-from. Also is layer-Cnode-Pnode; same thing.

x was node-layer. Now is layer-node-residual.

y was node-layer. Now is layer-node-residual.

z was node-layer. Now is layer-node-residual.

Q was current layer node – previous layer node - layer. Now is layer-Cnode-Pnode-residual.

P was node\_c – input layer node – layer. Now is layer-Inode-Cnode-residual.

V was node-[W]-layer. Now is layer-node-W-residual = layer-node-Wlayer-Wto-Wfrom-residual.

A was node-input layer node-layer. Now is layer-Inode-Cnode-residual.

B was node-input layer node-[W]-layer. Now is layer-Inode-Cnode-Wlayer-Wto-Wfrom-residual.

C was node-input layer node-[W]-layer. Now is layer-Inode-Cnode-Wlayer-Wto-Wfrom-residual.

D was Wto-Inode-layer. Now is layer-Inode-Wto.

E was Cnode-Inode-[W]-layer. Now is layer-Inode-Cnode-Wlayer-Wto-Wfrom-residual.

F is as V; layer-node-Wlayer-Wto-Wfrom-residual.

Strategy is going to be to build this with full calculations, then to replace big 0-matrices with 0s or [] and include structure checking in the calculations.

Strategy is to start with list comprehension-style calculations, and eventually change to matrix multiplication.

Strategy is to write the formulas in the general sense, and then put an “if j==0” block at the front of it. J==0 block written.

Strategy is to write the formulas in a general sense, and then recognize that layer j depends only on the values of layer j-1, so instead of having full data structures representing all of the layers, retain the previous layer and build the next layer from it. That allows the simplification of the j==0 block as we can define the previous layer immediately as the inputs.

Strategy is to assume 1st and 2nd derivatives are used for every variable, and then revise the code to utilize the lists. Already revised.

Next action: test data structures. Use nodes = [3 2 2 1].

All of the equations were written for a single residual. I intended to, and in test do, have multiple residuals. Where is the residual layer? By the current definition of y, it is y[j][res][i]. Given that the future plan is to turn y[j] into y\_previous and y\_current, maintaining only the two, this would make [res] in this position reasonable. The other choice, of course, is that [res] is the very last index always, so that if these calculations can operate on lists then the calculations as written would need no modification. This is my preferred choice, as I’d like to get to matrix multiplication.

Then again, reading up on [list comprehension speeds vs numpy array speeds](https://stackoverflow.com/questions/31598677/why-list-comprehension-is-much-faster-than-numpy-for-multiplying-arrays), list comprehension may actually be better for this library:

Creation of numpy arrays is much slower than creation of lists:

In [153]: %timeit a = [[2,3,5],[3,6,2],[1,3,2]]

1000000 loops, best of 3: 308 ns per loop

In [154]: %timeit a = np.array([[2,3,5],[3,6,2],[1,3,2]])

100000 loops, best of 3: 2.27 µs per loop

There can also fixed costs incurred by NumPy function calls before the meat of the calculation can be performed by a fast underlying C/Fortran function. This can include ensuring the inputs are NumPy arrays,

These setup/fixed costs are something to keep in mind before assuming NumPy solutions are inherently faster than pure-Python solutions. NumPy shines when you set up *large* arrays *once* and then perform many fast NumPy operations on the arrays. It may fail to outperform pure Python if the arrays are small because the setup cost can outweigh the benefit of offloading the calculations to compiled C/Fortran functions. For small arrays there simply may not be enough calculations to make it worth it.

That’s not what we’re doing, so we may be best sticking to list comprehension.

Assuming we keep list comprehension permanently, it may still be best to put [res] as the last index because we have some logic for when a [k] doesn’t need to be calculated – doing that once instead of len([res]) times may be best.

Debugged j=0, j=1; there’s a structure bug in j=2.e. It turns out the equation is correct, but there is a deltanj that would prevent a data size mismatch in the math that is occurring in the code. Thus the delta has to be built in as an “if” statement so that the mismatched data (that would get multiplied by 0) doesn’t get calculated. There is a second delta in e that needs to be corrected. Done and validated. There is one delta in v that should be evaluated and corrected if necessary. It’s a deltaim and they can have different dimensions, so it needs to be fixed. Done. There’s a double in c as well.

C has another error, specifically that self.a was never defined! Defined, debugged.

Program runs completely for a particular model shape. Trying a larger model, found new bugs. In e, there is some question over whether the q is jri or jir. Jri seems correct. Jir had been previously implemented and worked – have to go back and test the small model with jri. Then in c also. Now both models work. That big model sure takes a while…

At this point we have only tested data structures, we have not validated calculations of anything. For that, we’ll need to fix the weights and do the calculations manually.