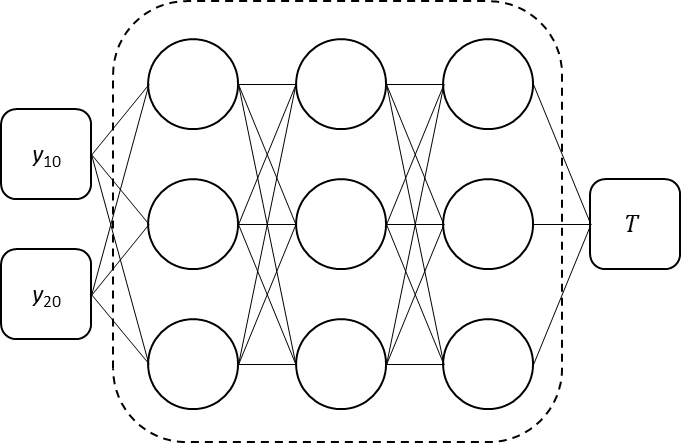
**Tensor Expressions of Algebraic Expressions of Derivatives**

First introduce slightly new notation. The outputs of a node within an NN is given the letter *y*. The inputs to an NN are the *y* of the 0th layer.

Consider a 2-input, 1-output NN with three nodes per hidden layer.



Denote weightings as *w*from-to-layer.

Denote the output of a node as *y*node-layer­.

Denote the argument going into a node as *x*node-layer.

**Inference**

For each layer *j*, there is a weight matrix *Wj* where *wikj* is the weightingrelating the output a node in the (j-1)th layer to the input of a node in the *j*th layer.

For each layer *j*, calculate an input vector ***x****j* where *xij* is the input into the *i*th node in the *j*th layer.

Using tensor notation suppresses the summation signs. Actually, don’t do that yet.

As layers calculate, we could populate more vectors ***x***j and could combine them into a matrix *X*, but I’m not certain that’s necessary, so let’s leave it out.

For each layer *j*, calculate an output vector ***y****j* where *yij* is the output of the *i*th node in the *j*th layer. Using a tanh activation function, for example,

Again these could be compiled into a matrix *Y*, but let’s not.

**Derivative**

For each layer *j*, calculate a vector ***z****j* expressing the derivative of the activation function of each node with respect to its input, as a function of the activation function output.

Note that nodes with other activation functions will have different expressions for *zij*. This vector can be calculated and stored numerically, as *yij* is already calculated.

For each layer *j*, calculate the derivative of each node’s output with respect to the outputs of the previous layer’s nodes. Denote this matrix Qj where:

This matrix can be calculated and stored numerically.

Now let’s build another forward-running artifact.

For the first layer (1), calculate the derivative of each node’s output with respect to the outputs of the 0th layer, e.g. the model inputs. Call this matrix P. For the first layer this will resemble Q but that won’t be a trend.

For the 2nd layer, make an extended matrix that includes the derivatives of each node’s output with respect to the outputs of the 0th layer.

I don’t like that expression, let’s rewrite it.

Much better. This elucidates the general form:

For each layer, build a matrix *Pj*. This includes the output layer. We now have a recursive feed-forward way of calculating the first derivative of the model output with respect to each of the model inputs simply through vector multiplication at most.

Next steps are to enable: d/dw, d2/dx2, d2/dxdw and d3/d2xdw. I sure hope there’s something iterative in here that enables further derivatives to be calculated, but for now these will fill the need.

**Gradient**

One method to calculate d/dw, that’s relatively heavy, is to make a matrix at each layer that captures the derivative of the output of each node with respect to each of the weightings in all previous layers. Each layer will have a progressively larger matrix to calculate.

A same-size-every-layer approach is a 4th rank tensor for each layer, *Vj*, where

The restriction we put on our model by required layered construction is *viklmj* = 0 for *m* > *j*.

Calculating the v for *m* = *j* is fairly easy:

Given that the weights are all independent of each other,

So summing over *m* is just replacing *m* with *k*;

These are calculated from already-calculated terms.

What about for *m* = *j* – 1?

Okay, that’s pretty ugly, but only because we went too far. Notice:

Where these *Vj* tensors are being built iteratively, so we all of the *viklm(j-1)* are already known numerically. Also we can make one more change to expedite calculations:

Have to think through *m* = *j* – 2 to ensure we have the correct iterative form.

So more generally, for m < j,

There. Gradient wasn’t so bad.

**Next up: 2nd derivative with respect to model inputs**

For each layer *j*, calculate a matrix *Aj* where *Aikj* is the second derivative of the output of node *i* with respect to the output of node *k* in the 0th layer, i.e. input *k*:

For the tanh activation function

Substituting this in,

Write the 1st derivative of an input with respect to the *k*th model input:

Write the 2nd derivative of an input with respect to the *k*th model input:

Substitute these in above:

So if the 2nd derivatives of the outputs of the *j* – 1 layer have been calculated, the derivatives of the outputs of the *j* layer can be calculated from known quantities. The derivatives in which we’re interested for PINNs are the derivatives with respect to the model inputs, e.g. the outputs of the 0th layer. So

**Gradient of 1st derivative**

It will be necessary to calculate the gradient of expressions that include the first derivative with respect to model inputs. This, sadly, adds another rank to the tensor that needs to be calculated, as the derivative may be calculated with respect to any model input. 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 calculation.

The gradient of a first derivative with respect to model inputs is:

Okay, so that was dense. Here were the steps: definition, substitution of p, substitution of equation for p, chain rule, {substitution for q and substitution of b}, chain rule, {substitution of z and dw/dw}, {chain rule and sum over delta}, reordering. The final equality is one that includes only already-calculated terms.

Note that the substitution of z is explicitly dependent on the activation function and will not always have this form or this derivative.

**Gradient of 2nd derivative**

It will be necessary to calculate the gradient of expressions that include the second derivative with respect to a single model input.

This is long and the chain rule is about to make it much wider, so deal with each term individually. The first term is:

(One supporting substitution we used.)

The second term is:

I see no point in writing these out together here; there is nothing that can be advantageously grouped.

Looking at the gradients of the first and second derivatives, when the gradient of the second derivative is going to be needed, the gradient of the first derivative is going to be needed. There may be common terms between them that can be pre-calculated to consolidate/expedite calculations:

With these substitutions,

Note that the expression for a is explicitly dependent on the activation function, as are a couple of other steps. For each activation function that will be enabled in this library, these expressions must be rigorously derived. That makes everything preceding this note “the tanh section” of the document.

Note that there is a little more manipulation that could have been written; d = p/z, and then maybe factor a y out of f, but there was no significant benefit in making these changes at this time.

This is enough to implement the whole library, using only tanh activation functions. Additional calculations will be necessary to enable different activation functions, like the linear one we frequently have for the last layer, but we can anticipate that tanh will be the workhorse of the hidden layers and thus we can use tanh-only models for the first implementation, in order to speed-test this against existing methods. If it isn’t faster for tanh, there’s no purpose in deriving equations for other activation functions.