**Tensor Expressions of Algebraic Expressions of Derivatives**

Since we write the loss function as D(Y) with D being a differential operator acting on Y, the output of the NN, and since the nodes are tanh(x) with

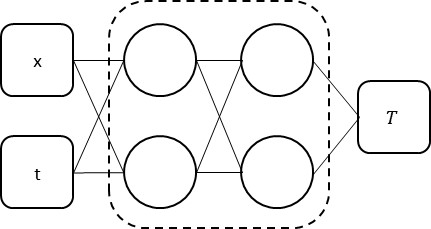
Or

Can we write as NN22(x) which is an NN that has weightings analytically relatable to the weightings of NN1, such that we don’t need to take derivatives to calculate ?

This could all be groundbreaking, or could be what TF has always done under the hood, except that it doesn’t assume z = (1-y2).

(Hey: How does it matter that , if at all?)

Consider a 2-input, 1-output NN with two 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.

Denote the input variables to the NN in two ways, as *x* and *t*, and as *x*inputnumber meaning *x*1 = *x* and *x*2 = t.

In this case we choose to have a linear activation function on the output layer. (Layer 3)

The loss term for the heat equation is

The output of the NN is

The weightings are independent of the input variables, so the derivative with respect to time is

Now use the derivative relationship (and the chain rule) to write this as:

Given that the numeric value of has already been determined, we can write

Noting that the inputs to the nodes at the second layer (*xn*2) are functions of time, take their derivatives. Focus on the first term

Focusing on the first term of that expression

Expanding that back out to the entire expression

All of these values are known; we can take the derivative of Temperature with respect to time, at any residual, without differentiating.

While this is calculable, it can be simplified. Observe the repeating of in two of the terms and of in two of the terms. And a corresponding similarity in the middle of each term.

Denote *q*from-to-layer = *w*from-to-layer\**z*to-layer

Using this notation, we can rewrite

Or on a single line as

**That was the easy part, this is the hard part**

Here we make several changes to how this expression is written. First, reverse the order of multiplication within each term:

Second, write the input variable *t* as *x*2 and write the output variable T as *y*13, then add some colors to highlight consistently-appearing indices:

Next, use tensor notation where a repeated letter of index within a product implies the sum over all values that index could have (, for instance). For clarity, use red and green colors to indicate indices where all combinations will be summed.

Now the notion that we’ve summed over these indices is clear and the expression compacts to:

It is hard to be satisfied by this, as it has a *w* sticking out. To make that a *q*, it would have to be multiplied by a *z*;

Note that the output layer was chosen to have a linear activation function:

The derivative of *y*13 with respect to its argument (*x*13) is:

Substituting this in,

The beginning of this derivation would have the *z* and *q* no matter what activation function were used for nodes in the output layer.

The derivative can now be written:

which is still incomplete. Note that there are multiple possible inputs and multiple possible outputs, and there will be need to take derivatives of any outputs with respect to any inputs:

This is the complete expression for a network with inputs *x*j and outputs *y*i3, e.g. a network with two hidden layers and one output layer.

But what if there were more layers in the network? Suppose there was another hidden layer such that

The first step of the derivative would be

Which of course simplifies and generalizes to

And so on. The fully general form is useless, but here it is:

Where *in* means “index 1, index 2, index 3…”

**Huh? So what?**

Next up is matrix algebra. Consider the matrix multiplication AB = C

Calculating the first-first term of *C*,

Which generalizes to

Now make substitutions / definitions

And multiply

Meaning that this summed product of the two *q* can be written as a matrix.

What of E = ABD = CD?

So the summed product of three *q* can be written as a matrix. And by induction the summed product of any number of *q* can be expressed as a matrix. If we write the matrix form a *q* as

And we make a matrix of the derivatives of the outputs with respect to the inputs

Then

And this we can code!

This matrix contains the derivatives of all of the outputs with respect to all of the inputs and can be evaluated at each residual point through simple matrix multiplication of quantities that were already calculated (while calculating the output for that residual). Note that the letters in this variable name are all capitalized to indicate that this is a matrix, and note also that this is *one* variable whose name has four letters arranged as a fraction; we should probably give it its own name, something like “*J”* for Jacobian. But we didn’t.

**Other Z**

Some other common activation functions include softplus and sigmoid. For softplus

Not quite as simple an expression but easily calculable from *y*.

For sigmoid

**Higher order derivatives**

This is a little less fun, and doesn’t end up nearly as compact as the first derivative. Only looking at the 2nd derivative here, but the method should apply to higher order derivatives as well. Build the derivatives iteratively. For some node

We are using the hyperbolic tangent activation function so the first two derivatives with respect to its argument are:

and

The important fact is that these are expressible for the activation function specific to this node, especially as a function of the node’s value. Extending what was done before, define

And

Given these definitions, we can write the derivative with respect to an arbitrary input as

This can quickly be iteratively built into the general form shown before, but the utility of that is lost. Instead, continue by taking the second derivative;

The goal of this exploration is to enable efficient calculation of the derivatives, which can be built iteratively. For convenience and to illustrate the constructs for calculation, introduce two new matrix sets:

And

The objective is to calculate specific terms within *GN*, meaning the second derivative of the model outputs with respect to various inputs. Calculating any term of *GN* requires calculation of all of the terms of *GN*-1 so build *G* and *J* up iteratively.

That’s it. There are several clever things that can be done to impact the form just a little, but they won’t end up reducing the amount of calculation that needs to be performed.

Wait. This is the exact process for evaluating a feed-forward network, isn’t it? Consider *J*. For a given *j*, each layer has *k* nodes

Where the activation function is

And the weightings are the same as in the original neural network! So while you have a main neutral network *NN* with tanh activation functions, *J* is calculated by using the same weightings in a network with activation functions calculated from the values of the nodes in *NN*.

What about *G*? Nope. Nothing so practical, as every layer needs inputs from the *J* matrix. In fact, file this under the category of “clever things that can be done to impact the form just a little, but that won’t reduce the amount of calculation that needs to be performed.

**The derivative of the loss with respect to the weightings**

To train the model, a loss function is defined (for the differential equation in particular) and the gradient of the loss function with respect to the weightings is used in some form of gradient descent. The loss function for the heat equation is

This can be written more generally as

But we won’t do that.

The gradient of the loss with respect to the weightings is comprised of partial derivatives of the sort

In the *J* and *G* notation from above, this is

The derivatives of *J* are

At this point, everything gets complicated and long, but not complex or hard to fathom. The first term, for instance has to be calculated by back-propagation or built up in a feed-forward manner. In this case it is specifically useful to note

[Continue from here. Explain how to build this up iteratively in a way that builds all derivatives simultaneously.][This isn’t done, read below.]

The second term simplifies quickly, as all weights are independent:

The third term [continue].

The derivative of *G* is [continue].

At this point, it became questionable if this would all compact to some form that could be easily calculated by a multiplication. But then a new epiphany occurred: it doesn’t matter if this loss term’s gradient is easy to calculate, as there will be other loss terms that are unlikely to have expressions of these kinds, thus the use of the gradient tape to calculate the gradient on the combined loss will remain necessary. But there may still be value in eliminating the gradient tapes used to calculate the derivative and 2nd derivatives necessary for the loss calculation.

**The Simpler Simplification**

Applying the advice of “use more neural networks”, the PDE loss function doesn’t have any derivatives. Instead the PDE loss term is (okay, the error term is…)

Where *A* is a NN construct that represents the time derivative of temperature and *C* is a NN construct that represents the 2nd derivative of temperature with respect to position. Given a NN construct *B* that represents the derivative of temperature with respect to position, not involved in the PDE loss term, these NN constructs are guaranteed to eventually represent the derivatives through other loss (error) terms:

This means that for any of these NNs we only need the general form of the first derivative, which is calculable as a matrix multiplication.

We never need the 2nd derivative, which has to be built iteratively.

Old versions of the code had 2nd derivatives and 1st derivatives, and then needed to take a gradient, resulting in 3 nested Gradient Tapes. Given that the derivatives can be calculated as an algebraic expression of known quantities, only one gradient tape is necessary, for the gradient of the losses with respect to the weights.

Much as before, it may be possible to calculate the gradient of some loss terms with respect to the weightings algebraically. None of the loss terms have 2nd derivatives, which simplifies this, but most loss terms involve weightings from two or more different NNs, such that the gradient with respect to a weight in one NN depends explicitly on values of weights in the same NN and in another NN. There is unlikely to be an advantage to working out an expression for these, as shown by this brief beginning for calculating the derivative of with respect to a weighting in *B*:

I’m not going to continue to work this because my intuition tells me that this will not simplify much, and that collectively we’ll be better off using a gradient for taking the gradient with respect to the weights of the NNs using the built-in functions of Tensorflow.

**But what about hard enforcement?**

Hard enforcement is done as

Where *g* is equal to the specified values at the specified points and *f* is equal to zero at the specified points. Here the is represented by an NN, and the *f* and *g* could be either provided for the specific problem or potentially represented by NNs if a couple of issues can be worked out about those. Regardless of representation, *f* and *g* are generally meant to be trained prior on fixed (specified) data prior to training . But what does this do to the construction of the derivative?

For all three of these functions, (one output so *i* = 1),

These could be calculated prior to training for *f* and *g* except that the residual points for training are not known prior to training (they are randomized each epoch), so using this technique for expressing derivatives as algebraic expressions of known quantities requires the *f* and *g* (for each of the NNs) to be calculated and their derivatives to be calculated at each epoch. Doing this with hard enforcement presumably has the same speed advantage (or disadvantage) as doing the same just for *T* without hard enforcement, it’s just more calculations per epoch. Those more calculations were already baked into the hard enforcement decision, so the speed advantage (if any) of using the algebraic expressions of derivatives should be independent of the decision to use hard enforcement of constraints.