GPT Utility Manual

link to the application : https://spectrograph.uk/transformer.html

To train a GPT encoder, the type of question one would ask is, how should the query matrix in level 3, together with the error E in matching training data, affect the key matrix in the attention step of level 5?

The first authors of a back-propogation machine hadn't included in their paper that they just use the chain rule, and gradient descent. However, a notion of gradient descent can depend on choosing an error function to take the gradient *of*, and converting its differential to a flow depends on a Riemannian matric on the weight space. By default, one has tended to use a Euclidean metric, but both the intrudction of softmax in attention layers and the notion of layer normalization suggest that other possible metrics could simplify or generalize the GPT architecture.

The model in 'Attention is all you need' introduces its own simplification, of letting the data at every level be represented by a matrix. At the beginning, each word of a sentence is encoded as a sequence of real numbers, the sequences can be adjusted to include position encoding, and still one begins with a matrix.

It does seem acceptable to use, on space of output matrices, the negative Euler flow, or, what corresponds in coordiantes x_{ij} with the sum $-\sum x_{ij}dx_{ij}$. This flow, acting on all possible error matrices, gives $E * dE$ if we denote by dE the four-tensor which is just the matrix whose entries are the differential forms dx_{ij}. The operation $*$ here is the multiplication of corresponding components, and we have

$$-E * dE = -trace(E^t dE)$$

with respect to more ordinary matrix multiplication.

In an encoder chain with one attention head per layer and no layer normalization, for example, the operation of the GPT in the i'th layer is

$$F_i(w, X) = m(a(n_i^{-1/2} X_i Q_i K_i^t X_i^t) X V_i + C b_i) W_i + C f_i \qquad (1)$$

where $w = (K_i, Q_i, V_i, W_i, b_i, f_i)$ is the tuple made of the query, key, value matrices and C is a column vector with entries of 1 so that the last parts denote two affine transformation; and m is the operation which sets any negative matrix entry to zero while a is softmax. (Note we absorb the linear part of an affine transformation acting on the right into V_i as we may).

Then if we train against an answer $f(X)$, we have $E = f(X) - F(w, X)$. For fixed X, or vieweing X as constant, this is a map $W \to V$ from the weight space to the answer space.

Although there is no natural pullback of vector-fields, the map is nothing like a local isomorphism as W has higher-dimension, even without choosing any error function *or* metric, we can pull back the Euler form along the error map, we are pulling back $trace(E^t dF(w, X))$ where F is the composite of the F_i, if we are only training a sequence of encoders.

If we simply apply d formally to a product of a copy of (1) for each layer, using Leibniz rule, we can finish the calculation. Just like for traces of ordinary matrices, with these four-tensors we expand out using the distributive rule, and then cyclically rotate each term to bring a letter dW to the end, where W is one of the matrices which are the component of the weight space.

That is, we can use the rule that even for matrices of forms applied to matrices of numbers,

$$trace(A\ d(B)\ C) = CAtrace(dB).$$

When we need to calculate $dm(X)$ for X any matrix, we write this

$$dm(X) = mm(X, dX)$$

where mm sets each entry of dX to zero when the corresponding entriy of X is negative. Note now that mm is linear with respect to addition in the second argument, so when X is a polynomial expression, we may continue to apply Leibniz rule in the second argument of mm. When we need to calculate $d(a(X))$ with a the softmax function, we find this is $aa(X)dX$ where $aa(X)$ is a matrix made using the entries of X and the exponential function.

These operations actually are not enough to rewrite the pullback of $E^t dE$ to the weight space without one additional move. It is that

$$trace(A\ mm(B, C)) = trace(\ mm(B^t), A)C\ ).$$

That is, when we have differentials trapped in the second argument of mm we can use this trick to flip the mm function onto the complementary part of the word.

In the utility, matrices are initially represented as symbolic entities (we just use nested arrays in the DOM with an extra key denoting the operation), later, in a part being written now, we will use ordinary matrix multiplication routines to make the calculation do-able, so we may chat with the utility.

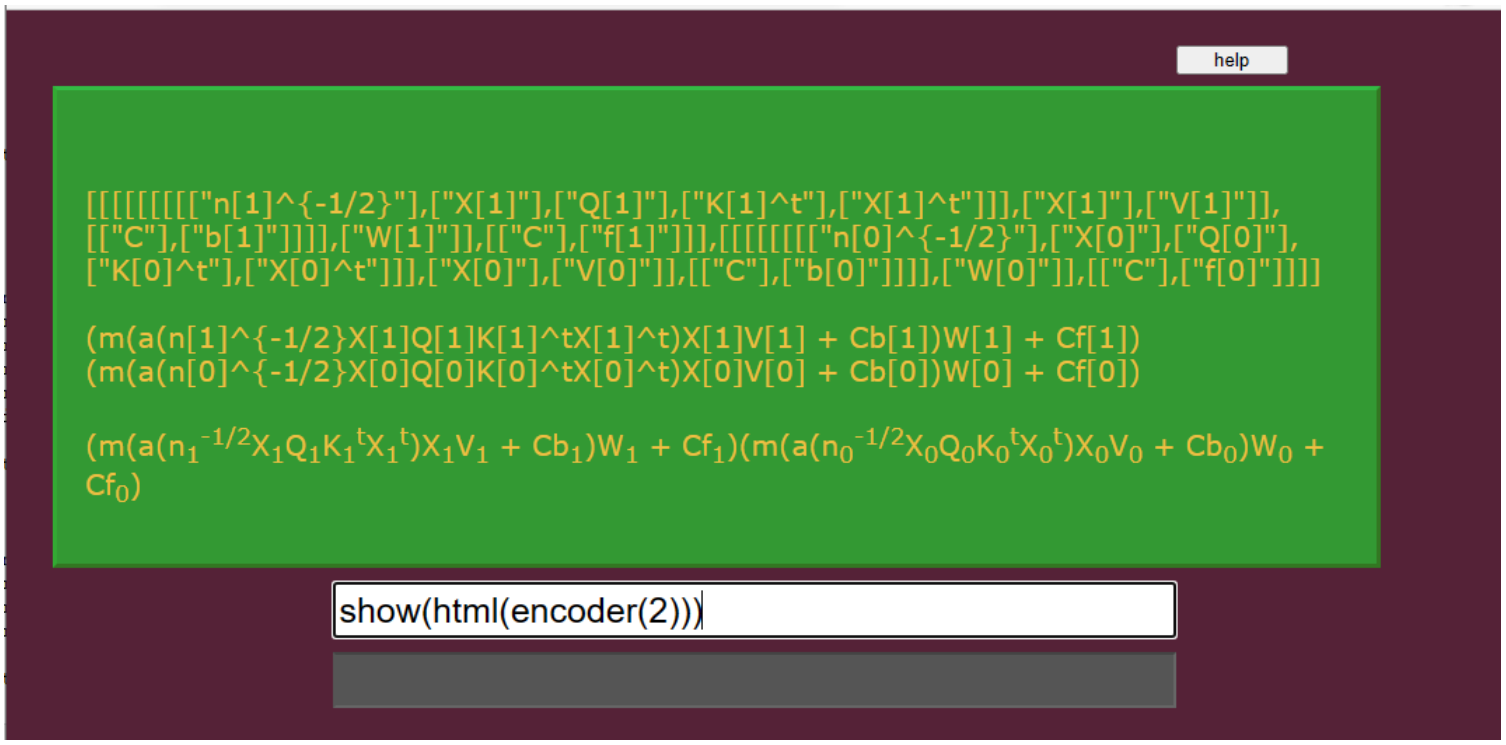
To create a matrix object A in the DOM we write $A = [\text{“}A\text{”}]; a.op =$ “” and now A has three manifestations, we can see the matrix in the display by writing

show(A)
show(string(A))
show(html(A))

If we want to describe the product or sum of two such objects we write this as otimes(A,B) or oplus(A,B). Unlike other things here, the preceding o does not relate to tensor products, it stands for ‘object’. For A, B variables as above, for instance $\otimes(A, B)$ is the array $[\text{“}A'', \text{“}B'']$ and we set $\otimes(A, B).op = \text{“}.''$.

Using these conventions, the architecture of a standard encoder is abbreviated by the function called 'encoder,' such that for example encoder(2) shows a composite of two layers. Here is a screenshot of the utility doing that.

```
show(encoder(2)
show(string(encoder(2))
show(html(encoder(2))
```



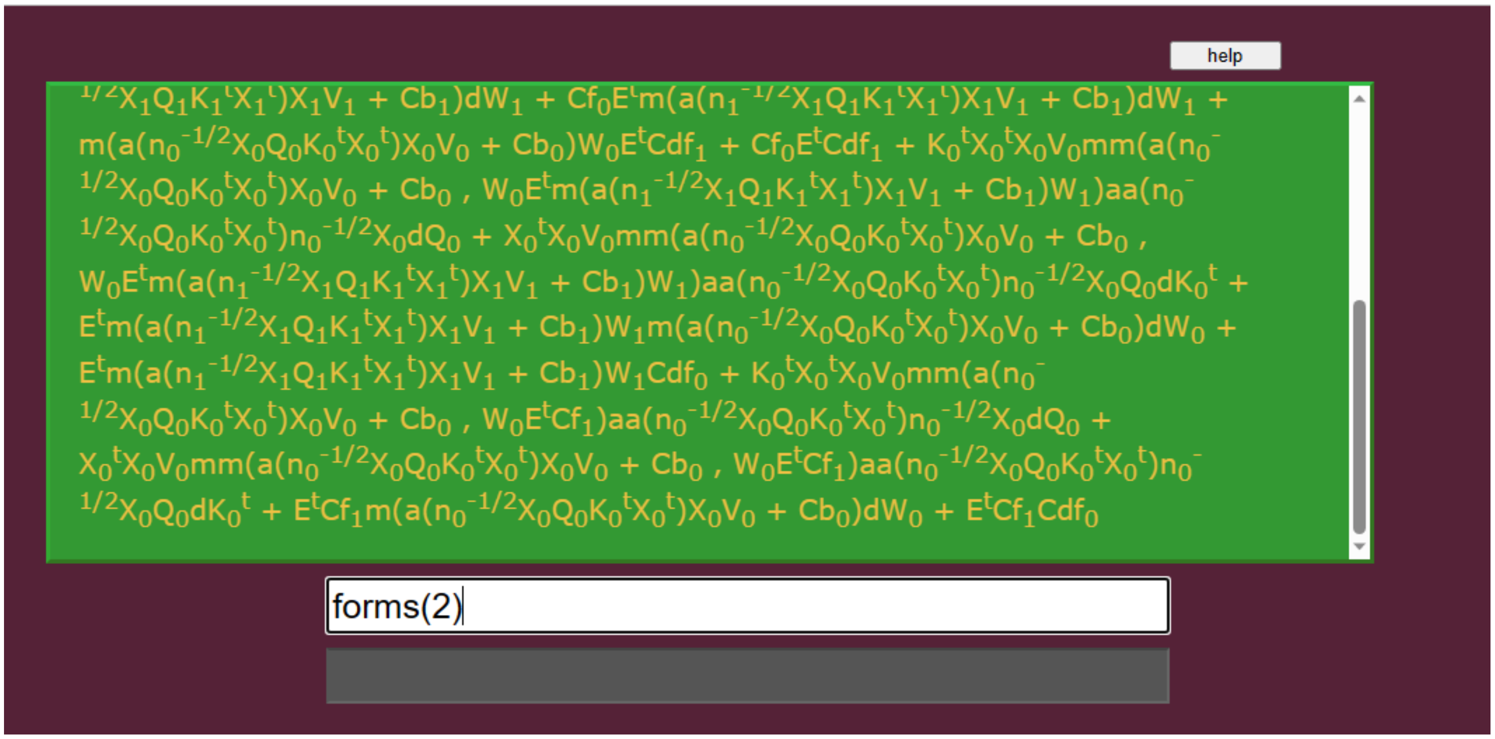
```
[[[[[[[[["n[1]^{-1/2}"],["X[1]"],["Q[1]"],["K[1]^t"],["X[1]^t"]]],["X[1]"],["V[1]"]],
[["C"],["b[1]"]]]],["W[1]"]],[["C"],["f[1]"]]],[[[[[[[["n[0]^{-1/2}"],["X[0]"],["Q[0]"],
["K[0]^t"],["X[0]^t"]]],["X[0]"],["V[0]"]],[["C"],["b[0]"]]]],["W[0]"]],[["C"],["f[0]"]]]]

(m(a(n[1]^{-1/2}X[1]Q[1]K[1]^tX[1]^t)X[1]V[1] + Cb[1])W[1] + Cf[1])
(m(a(n[0]^{-1/2}X[0]Q[0]K[0]^tX[0]^t)X[0]V[0] + Cb[0])W[0] + Cf[0])
```

$(m(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1 + Cf_1)(m(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)W_0 + Cf_0)$

```
show(html(encoder(2)))
```

The operation d applies to any of these using the transformation rules above. When they are all combined, they yield the explicit formula for $-E^t dE$, the whole form in html can be displayed by using the shortcut forms(2).



$^{1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)dW_1 + Cf_0E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)dW_1 +$
$m(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)W_0E^tCdf_1 + Cf_0E^tCdf_1 + K_0{}^tX_0{}^tX_0V_0mm(a(n_0{}^{-}$
$^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0\ ,\ W_0E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1)aa(n_0{}^{-}$
$^{1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0dQ_0 + X_0{}^tX_0V_0mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0\ ,$
$W_0E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^t +$
$E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1m(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)dW_0 +$
$E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1Cdf_0 + K_0{}^tX_0{}^tX_0V_0mm(a(n_0{}^{-}$
$^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0\ ,\ W_0E^tCf_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0dQ_0 +$
$X_0{}^tX_0V_0mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0\ ,\ W_0E^tCf_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-}$
$^{1/2}X_0Q_0dK_0{}^t + E^tCf_1m(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)dW_0 + E^tCf_1Cdf_0$

```
forms(2)
```

I'm cheating a bit by hitting up-arrow to bring command back in the window, actually I have it scroll out of view when you press 'enter' so you can give the next command.

Anyway, this is too large to fit on the screen, so to see just one summand we can write, for instance forms(2,4).

The tensor at the end dK_1^t, is a matrix of differential forms made from the coordinates of the key matrix at layer 1, and it has a single matrix as a coefficient, we see here the expression for that matrix, it involves the two-place function mm – we've made a change for simplicity with respect to the transformation rule, so I can tell you the correct definition of $mm(A, B)$ is to set the entries of B

to zero which are in the position of negative entries of A^t, – and the dimensions will all line up so this makes sense. The function $a(\ )$ is softmax, and $aa(\ )$ is a single variable function also built using exponentials of its one matrix argument, such that $da(X) = aa(X)dX$.

It might help to explain the algorithm if we write down the definition of the function forms(i). It is

show(html(cycle(shiftmm(flatten(expandmm(expandAll(form(encoder(i))))))))

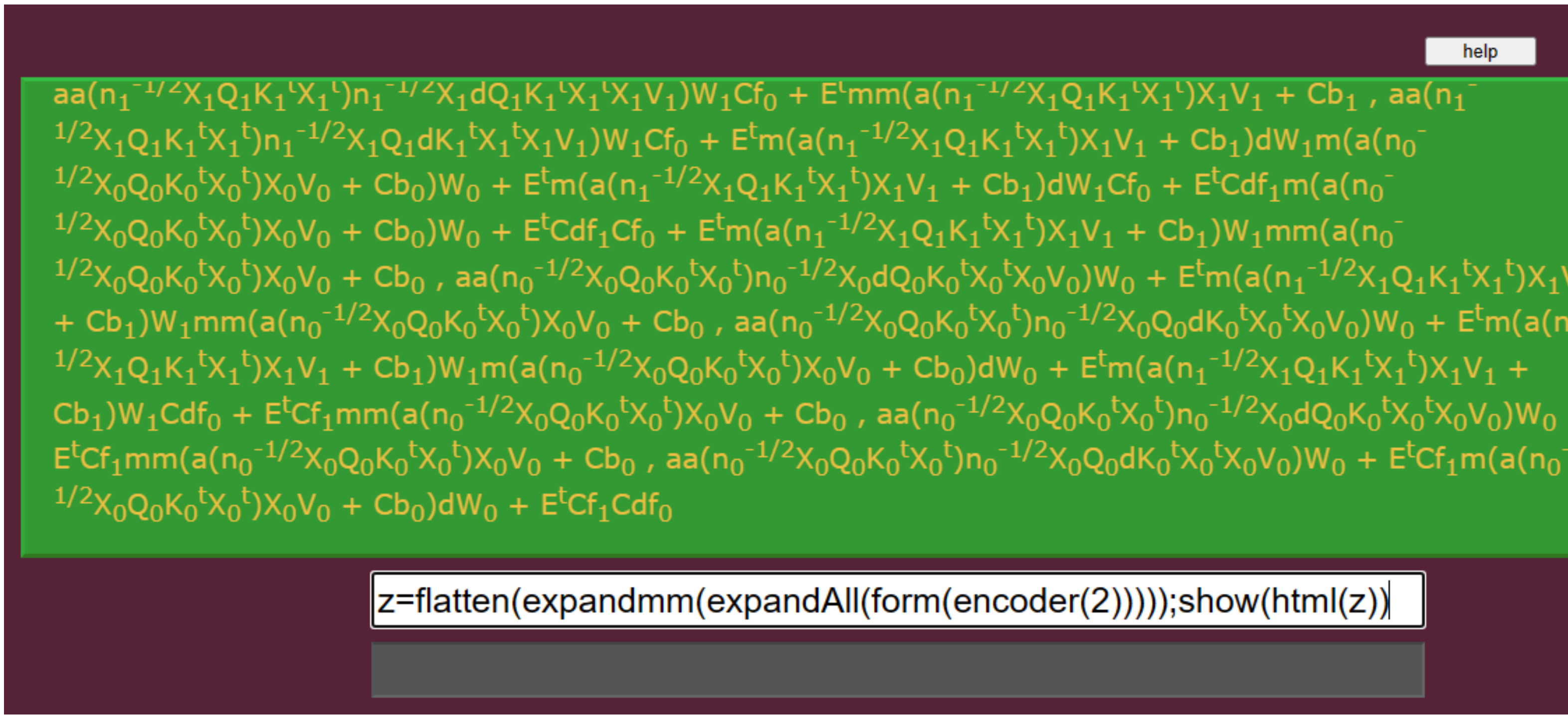
We already explained how encoder(i) describes the action of i layers of encoding, now $form$ is just defined such that

$$form(E) = -E^t E.$$

The function expandAll() just does multiplying out in the group algebra of the free monoid. We can test this by making some variables which we just do in the website DOM as

A=["A"];A.op="";B=["B"];B.op=""; Y=otimes(oplus(A,B),A), Z=otimes(Y,Y)
and we can write

showHTML(Z)
showHTML(expandAll(Z))



$aa(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)n_1{}^{-1/2}X_1dQ_1K_1{}^tX_1{}^tX_1V_1)W_1Cf_0 + E^tmm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1 , aa(n_1{}^{-}$
${}^{1/2}X_1Q_1K_1{}^tX_1{}^t)n_1{}^{-1/2}X_1Q_1dK_1{}^tX_1{}^tX_1V_1)W_1Cf_0 + E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)dW_1m(a(n_0{}^{-}$
${}^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)W_0 + E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)dW_1Cf_0 + E^tCdf_1m(a(n_0{}^{-}$
${}^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)W_0 + E^tCdf_1Cf_0 + E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1mm(a(n_0{}^{-}$
${}^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0dQ_0K_0{}^tX_0{}^tX_0V_0)W_0 + E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V$
$+ Cb_1)W_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0)W_0 + E^tm(a(n$
${}^{1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 + Cb_1)W_1m(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)dW_0 + E^tm(a(n_1{}^{-1/2}X_1Q_1K_1{}^tX_1{}^t)X_1V_1 +$
$Cb_1)W_1Cdf_0 + E^tCf_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0dQ_0K_0{}^tX_0{}^tX_0V_0)W_0$
$E^tCf_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0)W_0 + E^tCf_1m(a(n_0$
${}^{1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0)dW_0 + E^tCf_1Cdf_0$

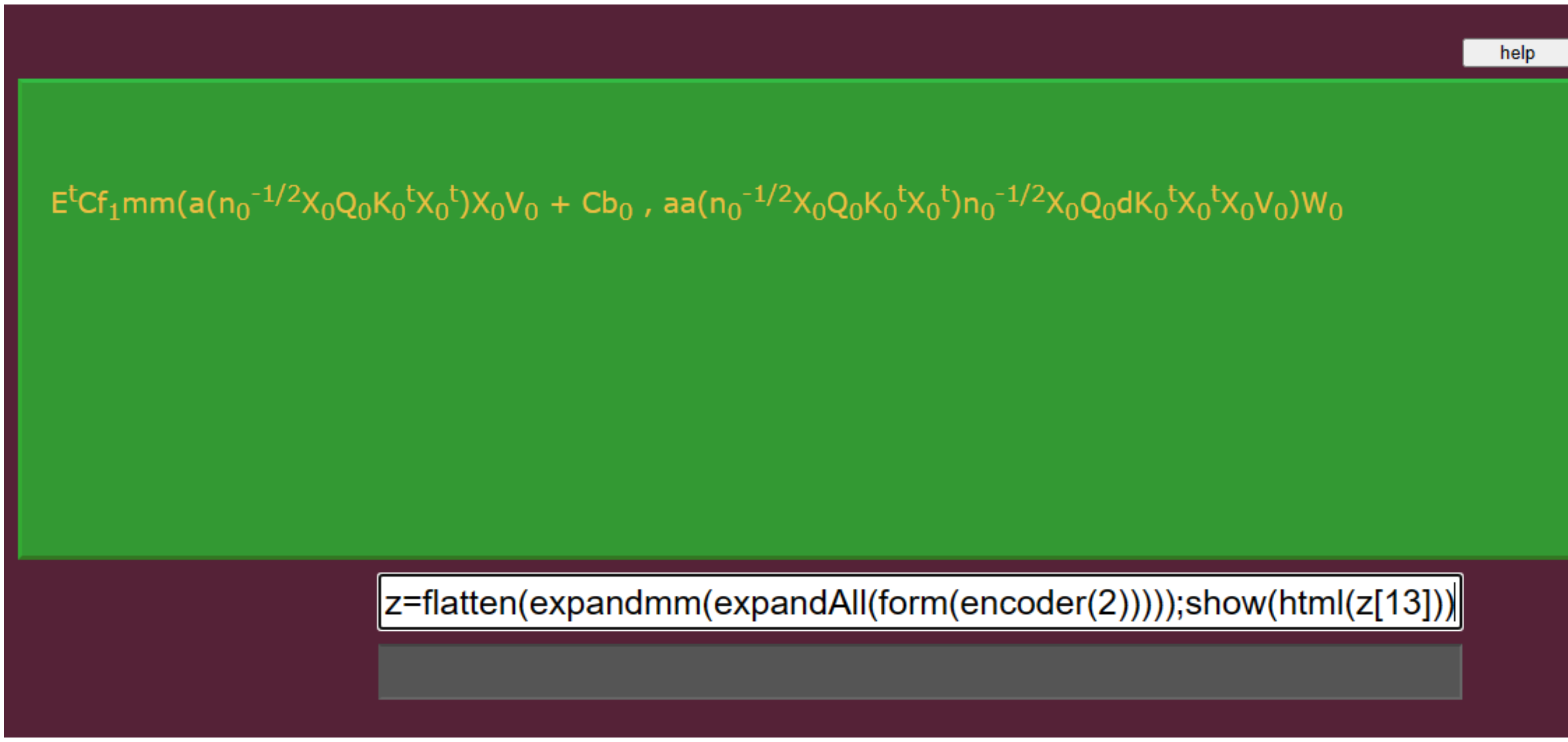
```
z=flatten(expandmm(expandAll(form(encoder(2)))));show(html(z))
```

Let's look at the action of expandmm, we take the object

$$flatten(expandmm(expandAll(form(encoder(2)))))$$

Here expandmm is just the continued expansion using the fact that the second argument of mm commutes with sums.

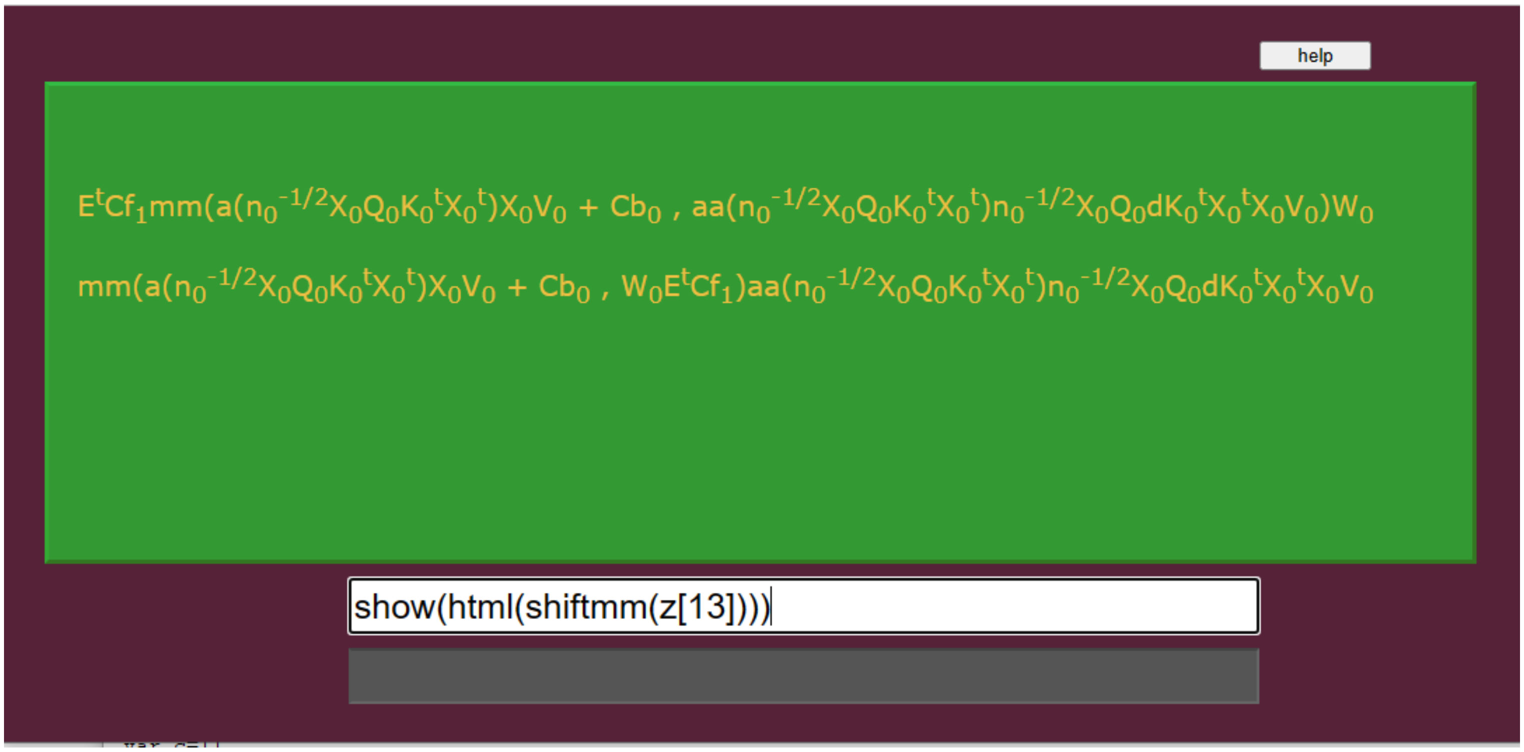
Let's only work on the part visible without scrolling, for now, the grey box below the command window is for errors and logging, if we write $log(z.length)$ we see z has 16 terms, so let's just look at one of the last ones, $show(html(z[13]))$



$$E^tCf_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0\ ,\ aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0)W_0$$

```
z=flatten(expandmm(expandAll(form(encoder(2)))));show(html(z[13]))
```

We see that dK_0^t occurs within the second argument of mm, so we cannot yet access the coefficient of dK_0^t, but this is where we apply shiftmm, we could apply it to the whole form, but let's just apply it to this part, so we write $show(html(shiftmm(z[13])))$ giving

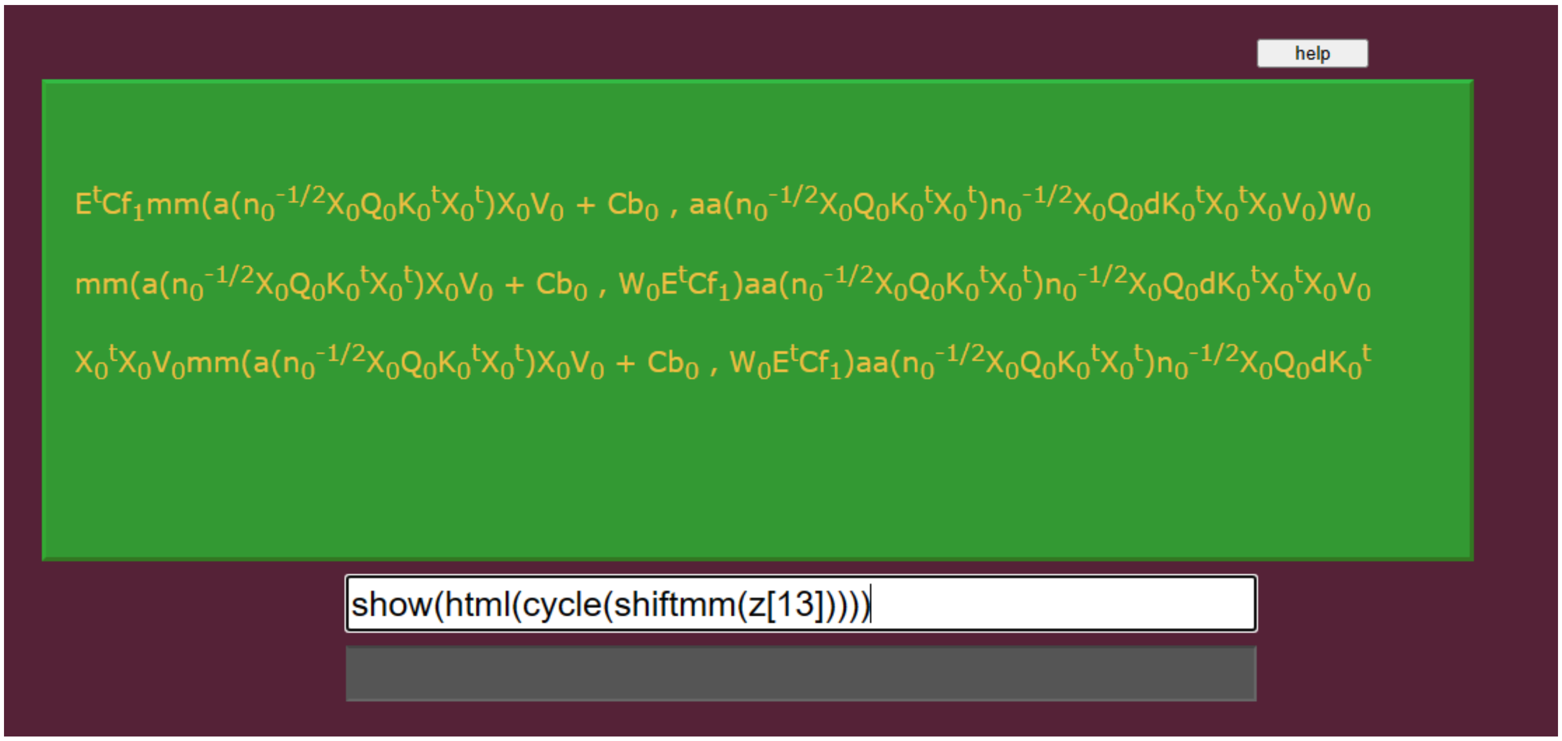


$E^tCf_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0)W_0$

$mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , W_0E^tCf_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0$

```
show(html(shiftmm(z[13])))
```

The differential dK_0^t has been removed from the argument of the function, and now we may apply the cycle operator



$E^tCf_1mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0)W_0$

$mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , W_0E^tCf_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^tX_0{}^tX_0V_0$

$X_0{}^tX_0V_0mm(a(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)X_0V_0 + Cb_0 , W_0E^tCf_1)aa(n_0{}^{-1/2}X_0Q_0K_0{}^tX_0{}^t)n_0{}^{-1/2}X_0Q_0dK_0{}^t$

```
show(html(cycle(shiftmm(z[13]))))
```

And we see that the order four tensor is written as a product of a matrix which we know, times a matrix of one-forms, which are the differentials of in this case the attention key of the zero'th encoding stage. The only weight from the next stage that affects this coefficient matrix is f_1 which is the row vector which adds a translation on one side of the max activation step in the next stage. And the transpose of the error matrix occurs in the middle of this coefficient matrix word, but, the point is, every factor in this word is known now by direct calulation.

I have written the entries of $aa(X)$ for any matrix X in the prequel to these notes, as I mention, each entry is given by a simple formula in terms of the entries of X and their exponentials.

By recognizing that pulling back differential forms is canonical, and that particular symbolic manipulations allow us to express the pullback in a normal form, we have a chance to understand the future role of a Riemannian metric on the weight space, and other conceptual simplificaitons to the GPT architecture.

If tensorflow is efficient about using sparse operations, this answer should agree with the answer given by tensorflow's gradientTape operation starting with the error function which is half the sum of squares, and the Euclidean metric on weight space. However, just having the final Jacobian matrix at the end may not give us as good insight as having a symbolic expression of the form which can be dualized againsts a choice of Riemannian metric to give a flow, or understood symbolically or theoretically.

While I'm thinking about this, I do want to say, I have not put in the code the step of transposing the first argument of mm when we do the flip, and yet in these calculations the flip is always done, and so we should merely re-interpret the dfefinition of mm such that $mm(A, B)$ sets the entries of B to zero in the positions of where entries of the *transpose* of A are negative. Since B here is a four-tensor, I need to explain the conventions, but if you think of B sometimes as a 'matrix of one-forms' then those very matrix entries indices which the phrase implicitly refers to are the ones which need to correspond, now, under transposition.

There are several other clarifications I wanted to make, however I can't now think what they are.

In any case, when I have finished connecting these matrices to the literal matrix multiplication routines, we can examine individual matrix coefficients during training, and have some detailed canonical insight into the mechanics of training a GPT.