Cyclic 4-tensor rotations underly the training of a GPT-style encoder

The basic architecture of GPT-style encoders nowadays involves a simple type of ‘neural’ step, composing affine transformations with the projection on the first ‘octant’ of a based vector-space, together with an attention step. The attention step is described in the paper “Attention is all you need,” and seems to occur in earlier papers.

Training invovles ‘gradient descent,’ and can be simplified conceptualy, and coded easily, by using the principle that instead we identify the Euler field with a differential one-form, via the Euclidean metric, and pull the form back to the weight space. We will give an easy calculation of the training step using rotations of cyclic matrix words.

Re-interpreting the form there as a flow again requires invoking the Euclidean metric on the weight-space, and it is my belief that various normalization steps, perhaps even the use of softmax in the attention step, could be combined and generalized if one allowed a Riemannian metric on the weight space.

We start with a vector-space V containing a finite set S, we think of the elements of S as words, or, sometimes, words together with information conveying their position in the sequence of words in a sentence.

In terms of particular coordinate functions on V, a well-used position encoding could add to each coordinate the x or y position of the minute, second, or hour hand of a progressing clock, for example. I’d rather be sure that the sum were direct, to avoid interference between word meanings and the position encodings.

If we write $V = \mathbb{R}^N$ then a sequence of n elements of V corresonds to an $n \times N$ matrix X, and the first attention step, in the case of a single attention head, is determined by three matrices, which could be learned, Q, K, V by sending X to $(\frac{1}{\sqrt{n}}XQK^tX^t)aXV$. We have taken to writing operators on the right, and a known as sofmax is the operation here of exponentiating the the entries of $\frac{1}{\sqrt{n}}XQK^tX^t$ and dividing each element by the sum of the elements in its row.

The entries of this matrix are $\frac{1}{\sqrt{n}}$ times the Euclidean inner products of the Qx with Ky when x, y rows of X. The softmax step makes many entries near zero, when it is not one imagines somehow that x 'pays attention' to y.

The neural step is just an affine transformation composed with projection on the first 'octant,' and another affine transformation. Since we have only a single attention head, the affine transformation is general if it is merely addition of a (the same) vector b_1 to each row, hence addition of $C = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} b_1$ to the matrix, as any linear transformation can be subsumed in V, and then we apply $(\ )m$, the truncation which acts at each point by a diagonal matrix with entries of 0 and 1, and then application of another affine transformation, so the whole encoder sends X to

$$((\frac{1}{\sqrt{n}} XQK^tX^t)XV + Cb_1)W + Cb_2 \qquad (1)$$

with Q, K, V matrices and b_1, b_2 row vectors.

We shall ignore the concept of adding X to this to provide a 'shortcut' and also the concept of normalizing the weights; the latter notion would be better understood if we were to use more general metrics.

Let's abbreviate the tuple (Q, K, V, W, b_1, b_2) by just the letter w and the set of all such tuples by the symbol W, an abuse of notation since we also use it for the name of a matrix. If we are trying to match a particular function of matrices $f : V^n \to V^n$, or more correctly $S^n \to V^n$, in the first instance with just this one layer of encoding, for each fixed $s \in S$ we write the negative Euler flow $-\sum x_i \partial/\partial x_i$ with x_i Euclidean coordinates on V and we identify this using Euclidean duality with the one-form $\sum x_i dx_i$. We pull this back along the error map

$$E : W \to V$$

$$w \mapsto f(s) - F(w, s)$$

with s fixed, and this gives us a differential 1-form on W.

There is a very conveient type of algebra which will let us do this. Starting with the query matrix Q, if the entries are considered to be variables q_{ij} we write dQ for the matrix of one-forms with entries dq_{ij}, and we do the same for K, V etcetera.

With $E = f(s) - F(w, s)$, we use the same convention for E, and pullback of our one-form along the error map to W is merely $-E{*}dE$ where the asterisk $*$ just means we multiply *correspnding* entries of E with *corresponding* entries of dE.

If we wanted to write this in terms of the action of matrices upon matrices of forms, it would be $E^t dE$.

Note that dE is actually an element of V^n tensor with its own cotangent space, so it is a 'tensor' which would be called 'rank four' in the language of tensorflow software.

But without using tensorflow we can calculate the compopnent of $dQ, dK, dV, dQ, db_1, db_2$ in this tensor. That is to say, we can write it as a matrix linear combination of these basic parts.

When we substitute (1) in the formula for E and expand using the distributive rule, we have a sum of matrix products. In each such product involving any of our weight matrices matrix X, we can collect the preceding and following terms to obtain a three-part expression AXB, and now we have interpreting A, B as constant

$$E * d(AXB) = trace(E^t d(AXB)) = trace(E^t A\ dXB)$$

The word trace here refers to the sum of the diagonal entries in a square matrix of one-forms. Nevertheless, just as for ordinary matrices, Without affecting this notion of trace we may cyclically rotate until dX is at the end, giving

$$trace(BE^t AdX) = (A^t EB^t) * dX$$

and now, the entries of $A^t EB^t$ are the coefficients of the corresponding entries of dX in $E{*}d(AXB)$. Whenever we have a sum of matrix words, we can use this easy principle to find the matrix of coefficients of any variable matrices in any of the words.

Let us state this as a theorem.

Theorem. Each matrix word in the expression for the error E when written in the form AXB where A is the part of the word before X and B is the part afterwards, contributes, to the pullback of the one-form corresponding to minus Euler derivation, precisely the sum of the differential one-forms which sit as the diagonal entries of the cyclically rotated 'rank 4' tensor $-BE^tA\ dX$

This applies when one of the matrices in the word represents projection on the octant, $()m$ because in a neighbourhood of any point (except if it is in a set of measure zero), we can interpret m as merely a fixed diagonal matrix with entries of 0 and 1.

To have enough tools to finish the calculation, I only need to explain how to deduce $d(Xa)$ from dX when $(\ )a$ is softmax.

We have

$$d(Xa)_{ij}) = \frac{1}{(e^{x_{i1}} + ... + e^{x_{in}})^2} \sum_k e^{x_{ij}+x^{ik}}(dx_{ij} - dx_{ik})$$

Then using the star operation of multiplying like terms and adding, for any constant matrix A we have

$$(A{*}d(Xa)) = \sum_{ij} a_{ij}(d(Xa)_{ij}) = \sum_i \frac{1}{(e^{x_{i1}} + ... + e^{x_{in}})^2} \sum_{jk} e^{x_{ij}+x_{ik}} a_{ij}dx_{ij} - a_{ij}dx_{ik}$$

The exponential coefficient being symmetric, we can interchange j and k appropriately

$$A{*}d(Xa) = \sum_{ij} a_{ij}(d(Xa)_{ij}) = \sum_i \frac{1}{(e^{x_{i1}} + ... + e^{x_{in}})^2} \sum_{jk} e^{x_{ij}+x_{ik}}(a_{ij} - a_{ik})dx_{ij}$$

$$= \sum_{ij} [\frac{\sum_k e^{x_{ij}+x_{ik}}(a_{ij} - a_{ik})}{\sum_k e^{x_{ik}}}] \cdot dx_{ij}$$

and we have calculated the coefficient of x_{ij} and the form is $A'{*}dX = A^t dX$ with A' the matrix whose entries are as given in the square brackets.

Note that the dot is more properly a tensor product sign.

Now we can apply this formula for any word containing $(X)a$. We write such a word in the form $A(Xa)B$ and the desired differential form is $trace(E^tAd(Xa)B) = trace(BE^tAd(Xa))$ and we substitute BE^tA in the formula above in place of A.

We actually will have occurrences of $(\frac{1}{sqrtn}XQK^tX^t)a$ in place of X and when the formula asks for dX we will really have $d(\frac{1}{\sqrt{n}}XQK^tX^t) = \frac{1}{sqrtn}(Xd(Q)K^tX^t + XQd(K)^tX^t + XQK^td(X)^t)$, note dX does not occur as X is considered constant here.

A very space-efficient training strategy for any composite of encoding steps is to combine these formulas in the obvious way to obtain our one-form on the overall weight space, which we re-interpret as a flow using the Euclidean matric. We could apply the flow for a short time corresponding to the error associated to each of a long sequence of correct and calculated answers.

I believe that the preference using the Euclidean metric at the last step to convert the differential form back to a flow is implicit in how we train Q and K separately, while in the actual encoding they are not used separately, only the product QK^t is ever used.

It is sometimes difficult to read tutorials about tensorflow, some of the operations are high-level and hence somewhat rigid. Through things like rotating cyclic words it is easy to reduce the tensor calculation to a matrix calculation and also undertand it that way.

When we train a sequence of encodings, a point of the space W will iclude entries for the $Q, K, ...$ of each stage, and the cyclic words will be longer, but the formula otherwise the same. Thus we have described explicitly the training strategy.

The actual strategy we would want to use would train not only on a series of encoding steps but also decoding, as the output of a series of encoding steps is a 'meaning,' and althogh we argue in another paper that we should not really think of this as encoding any actual meaning, still it shares with 'meaning' that there is no easy way we can describe for a large series of sentences what is the 'actual meaning' in any literal sense.

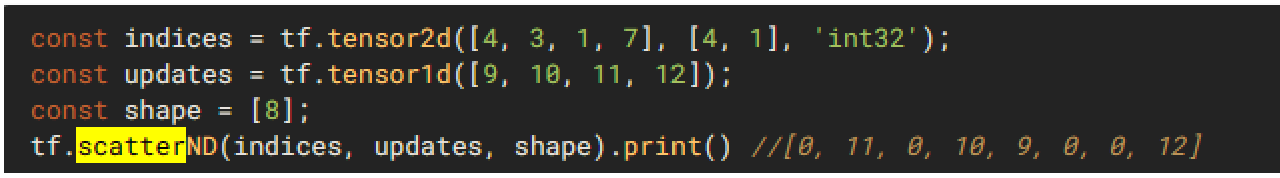
So although we can write down the training algorithm for just a series of encoders, it may be useful only to use this when the size of n decreases with successive layers, so the output is only something like an indicator of truthfulness, or kindness, or some other attribute of meaning where we might be able to find a store of 'true sentences' and 'false sentences.' Thus we could use the literal training strategy we've described to train just a series of encoders to recognize a notion of truth or reliability, for example. and only when a series of decoders (which function very similarly to the encoders) could we train on something like language translation.

I have omitted adding X to the right side of (1) to include a 'shortcut' around an encoding step. It might be worth considering what type of feed-through one should want to have when n decreases step-by-step.

An associated javascript at https://spectrograph.uk/transformer.html is under construction where we will train just a series of encoders to detect truth.

The algorithm

I considered encoding this in tensorflow – there is a google tutorial along those lines which has the user uninstall and install many packages. There is also tfjs.js, however tensorflow documentation and organization is horrible, to create a "rank" four tensor corresponding to dX for X a matrix one must set one entry to 1, the 'scattternp' function shows that to do this for a 'rank two' tensor requires an auxiliary rank one and rank two tensor, and tfjs.js crashes unless the type is set to integer... which is not possible upon object creation



```
const indices = tf.tensor2d([4, 3, 1, 7], [4, 1], 'int32');
const updates = tf.tensor1d([9, 10, 11, 12]);
const shape = [8];
tf.scatterND(indices, updates, shape).print() //[0, 11, 0, 10, 9, 0, 0, 12]
```

Yet, just as working on the blackboard, since it only needs to be done once for each particular model architecture, we can use regular expression rules. We start with $-EdE$ which we might call the negative Euler form, and write E as a finite sum of words. Each letter in one of the words is a constant matrix, a weight matrix, or its transpose, or is $(\ )a$ or $(\ )m$ applied to such a word.

Let's switch to writing a and m on the left since we'll be using javascript.

If we write $mm(A, B)$ for the matrix (or matrix of forms) B with the entries set to zero which are in the position where A has negative entries, then we have the valid rules

$$d\ m(A) = mm(A, dA).$$

and

$$mm(A, B + C) = mm(A, B) + mm(A, C).$$

Also, of course, we may use the associative and distributive rules and Leibniz rule acting on the second argument of mm, then, in order to 'flip' a factor which contains a differential out of the second argument we have the crucial rule

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

As for $a(\ )$ if we write $aa(A, X)$ for the matrix A' such that $d\ (AdX) = A'dX$ we have of course

$$d\ a(AdX) = aa(A, X)dX,$$

and when dX is a sum $B_i dC_i$ we have as just a consequence of associativity of matrix multiplication

$$d\ a(AdX) = \sum aa(A, X)B_i\ dC_i.$$

It follows that we can write a simple regular expression loop which will translate the string representing the minus Euler form, in human-readable format, $-E * dE$ into a sum for each weight matrix W of any level, of an expression AdW where A is a word consisting of the original matrices or their transposes or their images when $aa(\ , \ )$ or $mm(\ , \ )$ are applied.

The actual training algorithm, in each step, merely evaluates for each W the sum of products of known matrices and transposes and values under $aa(\ , \ )$ and $mm(\ , \ )$.

At some point we will also put in memoziation to avoid repeated calculations in calculating matrix products.

With the type of configuration we're considering, and the same is true for the configuration in “Attention is all you need,” when we expand $-E * dE$ as a sum, each weight matrix in each level only occurs in one of the terms of the sum. I do not know if this may be a reason why such configurations work well practically.

Note that we're assuming that the input matrix has already been position-encoded, we'll put in the standard position encoding with an exponential base chosen to be compatible with the values of n and N which the user chooses for the first layer.

As I say, this project is underway at what will be a small, perhaps two or three page javascript, which when the user chooses a configuration will display the human-interprable string for $-E*dE$ which also the training algorithm will read before it can begin training. It will be at https://spectrograph.uk/transformer.html