Transformer decoding in fifty lines of pseudocode

Bob Carpenter, Flatiron Institute, bcarpenter@flatironinstitute.org October 16, 2023

Overview

This note provides pseudocode for a decoder-only transformer as used by OpenAI's generative pre-trained transformers (GPT) version 2. Version 3 uses sparse multi-head attention (each head looks at a subset of the tokens in the history). Version 4's architecture was not released.

The first part presents the decoder pseudocode assuming the parameters (`alpha`, `betas`, `gammas`, `delta`) have been estimated. We simplify to single-head attention in the first pass and circle back to multi-head attention after presenting pseudocode for prompt completion and training.

What is a language model?

Tokenization

First, we will assume that language is a sequence of discrete symbols, which we will call *tokens*. For written language, words or characters are obvious choices, but large language models tend to use token sets of roughly $50,000$ tokens that correspond to subwords. Let's see how that sentence we just wrote is tokenized by the GPT API (available online at `https://platform.openai.com/tokenizer`):

```
These days, large language models tend to use token sets
 of roughly $50,000$ tokens that correspond to subwords.
```

The colored boxes indicate the tokens. Most words and the spaces before them make up a single token, but note that the word "subwords" was split into a token consisting of a space and "sub" and the token "words" with no leading space. The punctuation items make up their own tokens, but notice that `$` shows up tokenized with and without a preceding space. Adding prefixes can change tokenization, e.g.,

```
discombobulatedness    antidiscombobulatedness
```

Language modeling

Let Tok be the set of tokens and $\text{Tok}^* = \{\text{tok}_1, \ldots, \text{tok}_N : N \in \mathbb{N}, \text{tok}_n \in \text{Tok}\}$ be the set of token sequences. A language model will assign probabilities to elements of Tok^*.

We will use higher-order Markovian language models that generate the next token based on the identity of the previous tokens. For GPT-3, the maximum history size is $2^{12} = 4096$ tokens. That is, we will assign conditional probabilities $p(\text{tok}_N \mid \text{tok}_1, \ldots, \text{tok}_{N-1})$, which is the probabilty of generating tok_N having seen tokens $\text{tok}_1, \ldots, \text{tok}_{N-1}$. Such a model is said to have order N. The probabilty of a sequence is then defined by the chain rule:

$$p(\text{tok}_1, \ldots, \text{tok}_N) = \prod_{n=1}^{N} p(\text{tok}_n \mid \text{tok}_1, \ldots, \text{tok}_{n-1}).$$

Training and generation

We use the log of the training sequence probability as the *objective function* to maximize during training. We fit with *maximum likelihood estimates* of parameters (aka weights) θ, given M training token sequences with lengths N_m,

$$\widehat{\theta} = \arg\max_{\theta} \prod_{m=1}^{M} p(\text{tok}_1, \ldots, \text{tok}_{N_m} \mid \theta)$$

This form of model is said to be *autoregressive* because it predicts the next item in a sequence based on the previous items. To generate a sequence, we start by generating a token at random according to $\text{tok}_1 \sim p(\cdot)$, the distribution over tokens. Then we generate $\text{tok}_2 \sim p(\cdot \mid \text{tok}_1)$ and so on.

Notation

If `A` is an $M \times N$ matrix, then we will use `A[m]` or `A[m, 1:N]` for the m-th row, which is an N-dimensional row vector. Ranges will be read inclusively, so that, for example, `1:5` denotes the sequence $1, 2, 3, 4, 5$. We use `simplex(T)` to denote a vector in $[0, \infty)^T$ that sums to unity. Given a type `T` we write `T[N]` for an N-dimensional array of objects of type `T`. We will use `.*` for elementwise product.

Sizes

```
T: number of tokens        K: key/query size      A: attention layers
N: history length          V: value size          L: feedforward net width
```

Decoder

```
DECODE(tok:     int<lower=1, upper=T>[N],           // 0 <= N <= MAX_TOKENS
       alpha:   matrix(T, V),
       betas:   { query: matrix(V, K),
                  key: matrix(V, K),
                  value: matrix(V, V) }[A],
       gammas:  { 1: vector(L),  2: matrix(L, V),
                  3: vector(V),  4: matrix(V, L) }[A],
       delta:   matrix(T, V) ):                                    simplex(T)
-------------------------------------------------------------------------------
for n in 1:N:                                       // embed input
    xs[n, 1:V] = alpha[tok[n], 1:V] + POS(n)            // embedding of token n
for a in 1:A:                                       // A attention layers
    xs = ATTEND(xs, betas[a])                           // update tokens jointly
    for n in 1:N:                                       // update tokens individually
        xs[n, 1:V] = FEED_FORWARD(xs[n, 1:V], gammas[a])  // with shared NN
return LOGISTIC_REGRESSION(xs[N, 1:V], delta)     // next token probs
```

Scaled Dot-Product Attention

```
ATTEND(x:           matrix(N, V),
       beta.query:  matrix(V, K),
       beta.key:    matrix(V, K),
       beta.value:  matrix(V, V)):                              matrix(N, V)
-------------------------------------------------------------------------------
for n in 1:N:
    q[n, 1:K] = x[n, 1:V] * beta.query               // optionally add intercept
    k[n, 1:K] = x[n, 1:V] * beta.key
    v[n, 1:V] = x[n, 1:V] * beta.value
for n in 1:N:
    lp[1:n-1] = [q[n] * k[1]', ..., q[n] * k[n-1]']   // dot product log probs
                / sqrt(V)                             // scaled by sqrt value size
    lp[n:N] = -inf
    p[1:N] = SOFTMAX(lp[1:N])                         // attention probs
    u[n, 1:V] = SUM(n' in 1:N) p[n'] * v[n', 1:V]     // weighted avg of token values
    y[n, 1:V] = STANDARDIZE(u[n, 1:V] + x[n, 1:V])    // add in to out (non-center)
return y
```

Positional embedding

```
POS(n:  int<low=1,up=N>):                                                vector(V)
-------------------------------------------------------------------------------
for i in 1:(V / 2):
    r = n / N**(2 * i / V)     // exponent ranges from 2/V to 1
    u[2 * i - 1] = sin(r)
    u[2 * i] = cos(r)
return u
```

Feed-forward neural network

```
FEED_FORWARD(x:       vector(V),
             gamma.1: vector(L),  gamma.2: matrix(L, V),
             gamma.3: vector(V),  gamma.4: matrix(V, L)}):             vector(V)
-------------------------------------------------------------------------------
u[1:L] = gamma.1 + gamma.2 * x              // first layer
w[1:L] = GELU(u)                            // non-linearity
y[1:V] = gamma.3 + gamma.4 * w              // second layer
return STANDARDIZE(x + y)                   // layer norm input + output
```

Logistic regression

```
LOGISTIC_REGRESSION(x: vector(V), delta: matrix(T, V)):        simplex(T)
-------------------------------------------------------------------------------
log_probs[1:T] = delta * x
return SOFTMAX(log_probs[1:T])
```

Helper functions

```
STANDARDIZE(u: vector(V)): vector(V)                 // aka layer norm
    return (u - mean(u)) / standard_deviation(u)

SOFTMAX(u: vector(V)):  simplex(V)
    return exp(u) / sum(exp(u))

GELU(u: vector(V)):  vector(V)
    return u .* Phi(u)               // Phi() std normal cdf (sigmoid)
```

Text completion

Given a sequence of input tokens, generate a response sequence of tokens. We assume a special "stop-generating" token `END_TOKEN` $\in 1:T$. The decoder can accept any size input, so we allow any size input to complete.

```
COMPLETE_TEXT(toks: int<low=1, up=T>[N'],
              int<low=0> max_tokens,
              ...decoder params...):                          int<low=1, up=T>[]
-------------------------------------------------------------------------------
toks_out = []
while (True):
    while (toks.size() > N) toks.pop_first()            // trim to <= N tokens
    prob = DECODE(toks, ...decoder params...)           // next token probs
    next_tok = categorical_rng(prob)                    // gen. next token randomly
    if (next_tok == END_TOKEN): return toks_out         // return if end token
    toks_out.push_last(next_tok)                        // append to output
    if (toks_out.size() == max_tokens): return toks_out  // return if max tokens
    toks.push_last(next_tok)                            // add next token to end
```

Parameter estimation (aka "learning")

Parameter estimates of the decoder parameters, all of which are real valued and unconstrained, proceeds by an approximate optimization using stochastic gradient descent (SGD) with the negative log density as an objective, using momentum as in the Adam optimizer. The input is a ragged array of tokens representing a total of `I` training sequences, each of length `J[i]`. `MAX_HISTORY` is determined by decoder.

```
LOG_DENSITY(toks: int<low=1, up=T>[I, J[1:I]],
            ...decoder params...):                                  real<up=0>
-------------------------------------------------------------------------------
log_density = 0
for i in 1:I:
   history = []
   for j in 1:J[i]:
       history_toks = []
       next_tok = toks[i, j]
       probs = DECODER(history_toks, ...decoder params...)
       log_density += log(probs[next_tok])
       if (history_toks.size() < MAX_HISTORY): history_toks.pop_first()
       history_toks.append_last(next_tok)
return log_density
```

The functions are typically coded in a system supporting GPU-based automatic differentiation, such as PyTorch Paszke et al. (2019) or JAX Bradbury et al. (2018), which allows the computation of the derivative of the output of this function (the log density) with respect to the decoder parameters. In machine learning, this is called "back-propagation," the general form of which is known as "automatic differentiation."

Dropout during parameter estimation

In practice, when training the neural networks involved, a random subset of nodes is selected to be "dropped out" during each iteration of optimization. In the simplest case, there will be a probability for dropping nodes and whether a node is dropped will be determined independently in each iteration. Dropout provides a form of implicit regularization, as originally described by Srivastava et al. (2014). Vaswani et al. (2017) used dropout when training transformers and this practice has continued with GPT.

Multi-head attention

In GPT-2, a total of H attention models, called "heads" are used in parallel. Each starts from the full value, but produces a value of size V / H. These H values are concatenated to get back to a value of size V. Residual structure and standardization are as before. The final affine layer was not in the original transformer architecture.

```
MH_ATTEND(x:           matrix(N, V),
          beta.query:  matrix(V, K)[H],
          beta.key:    matrix(V, K)[H],
          beta.value:  matrix(V, V/H)[H],
          tau:         matrix(N, N),
          rho:         vector(V)):                                  matrix(N, V)
-------------------------------------------------------------------------------
for h in 1:H:                                                  // parallel heads
    for n in 1:N:
        q[n, 1:K] = x[n, 1:V] * beta_query[h]                  // q, k, v vary by head
        k[n, 1:K] = x[n, 1:V] * beta_key[h]
        v[n, 1:V/H] = x[n, 1:V] * beta_value[h]
    for n in 1:N:                                              // loop unchanged
        lp[1:n-1] = [q[n] * k[1]', ..., q[n] * k[n-1]']
                    / sqrt(V)
        lp[n:N] = -inf
        p[1:N] = SOFTMAX(lp[1:N])
        u[h, n, 1:V] = SUM(n' in 1:N) p[n'] * v[n', 1:V]
for n in 1:N:
    z[n, 1:V] = concat(u[1, n, 1:V], ..., u[H, n, 1:V])     // concat results
w[1:N, 1:V] = tau * z + rho * [1 ... 1]                      // affine transform
for n in 1:N:
    y[n, 1:V] = STANDARDIZE(x[n, 1:V] + z[1:V])             // residual + std
return y
```

Sparse attention

GPT-3 adds sparse, multi-head attention where each of the multiple attention heads restricts attention to a subsequence of the full token history.

From decoders to chatbots

As presented, we have a very sophisticated text completion engine. To turn a decoder-style transformer as we have presented into a chatbot, OpenAI proceeded in two stages. First, the text stream is marked up (with text) to indicate "user" and "assistant" turns with an additional "system" prompt prefixed to user queries.

1. fine tune (additional training) on human-generated answers to example prompts
2. have GPT generate multiple answers to prompts
3. humans rank GPT's output and provide ordinal scores (e.g., 1–5)
4. rankings used in reinforcement learning with human feedback (RLHF)
    - requires training a reward model on the side

The goal is to "align" a large language model to be (a) helpful, (b) truthful, and (c) harmless. These are subjective notions, the implementation of which will depend on the human-generated answers and human-generated rankings.

History and further reading

Language models were introduced by Shannon (1948), who explored estimating and generating from 3rd order character models and 2nd order word models.

The transformer architecture was introduced by Vaswani et al. (2017), who used an encoder-decoder architecture for translation. It actually simplifies some of the architectures it replaced, hence the paper title, "Attention is all you need." The encoder embedded the text to translate and it was available as additional context during decoding.

Phuong and Hutter (2022) also provide pseudocode for transformers. Their pseudocode is more finely factored and general than what I have presented here. They provide pseudocode for multiple applications of transformers including the original encoder/decoder machine translation application, the BERT architecture, as well as the autoregressive, decoder-only architecture of GPT.

Other implementations

There are several useful complete implementations of the transformer architecture.

- Python with TensorFlow (OpenAI): `https://github.com/openai/gpt-2`,
- Python with PyTorch (Andrej Karpathy): `https://github.com/karpathy/nanoGPT`,
- C++ (Georgi Gerganov): `https://github.com/ggerganov/llama.cpp`
- C (Febrice Bellard): `https://bellard.org/nncp/`, and
- Stan (Daniel Lee): `https://github.com/bayesianops/gpt-tutorial/`.

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. (2018). JAX: composable transformations of Python+NumPy programs.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035.

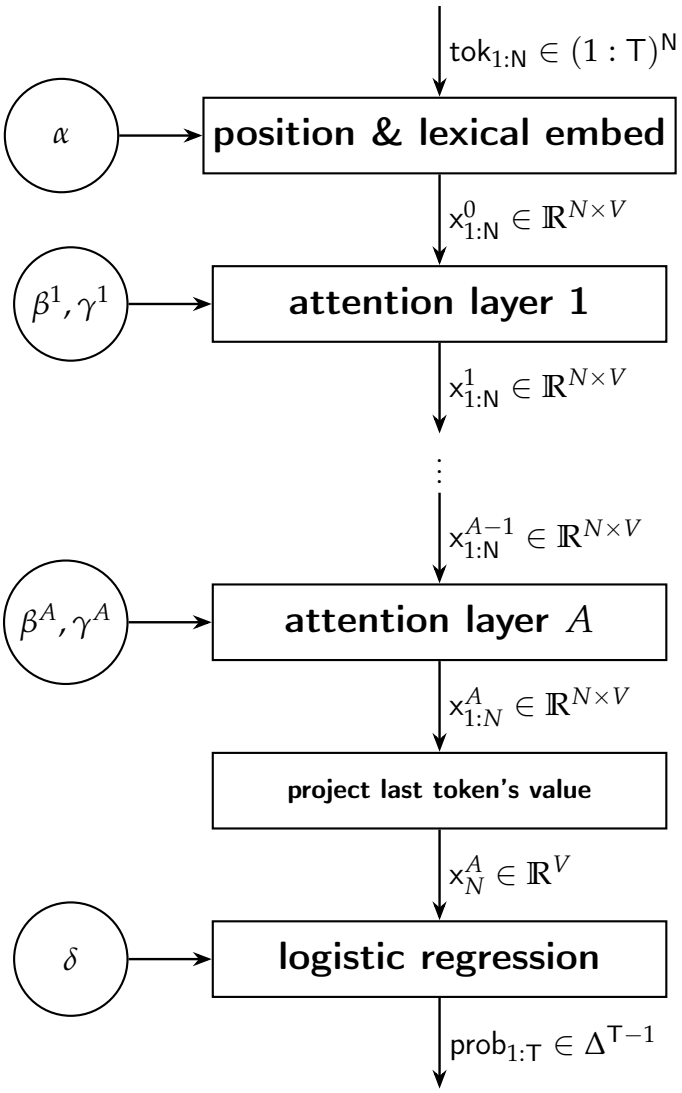
Phuong, M. and Hutter, M. (2022). Formal algorithms for transformers. *arXiv*, 2207.09238.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.

Decoder diagram



Attention layer diagram