Sivchenko_translate
13 часов назад

Клон ChatGPT в 3000 байтах на C, основанный на GPT-2

3.5K

Машинное обучение*, C*, Программирование*, Занимательные задачки, Искусственный

Перевод

Автор оригинала: Nicholas Carlini

Эта программа представляет собой свободную от зависимостей реализацию GPT-2. Она загружает матрицу весов и файл BPE из оригинальных файлов TensorFlow, токенизирует вывод при помощи простого энкодера, работающего по принципу частотного кодирования, реализует базовый пакет для линейной алгебры, в котором заключены математические операции над матрицами, определяет архитектуру трансформера, выполняет инференс трансформера, а затем очищает вывод от токенов при помощи BPE-декодера. Всё это — примерно в 3000 байт на C.

Код достаточно эффективно оптимизирован — настолько, что малый GPT-2 на любой современной машине выдаёт отклик всего за несколько секунд. Чтобы этого добиться, я реализовал KV-кэширование и применил эффективный алгоритм перемножения матриц, а также добавил опциональный OMP-параллелизм.

Взяв это за основу, можно создать некий аналог Chat GPT — при условии, что вас не слишком волнует качество вывода (объективно говоря, вывод получается просто ужасный… но решение работает). Здесь есть некоторые глюки (особенно с обработкой символов в кодировке UTF-8), а для эксплуатации модели размером XL с широким контекстным окном может потребоваться ~100 ГБ оперативной памяти. Но, если вы просто набираете текст в кодировке ASCII при помощи

малого GPT2, то такая модель должна нормально работать примерно везде.

Я выложил весь код на GitHub, поэтому можете свободно брать его там и экспериментировать с ним.

Эта программа состоит из следующих основных блоков (каждый из них сопровождается соответствующим кодом): Библиотека для базовой матричной математики (700 байт) Быстрое перемножение матриц (300 байт) Слои нейронной сети (300 байт) Модель-трансформер (600 байт) Частотное кодирование (400 байт) ввод/вывод (200 байт) Загрузка весов (300 байт) Загрузка данных для частотного кодирования (300 байт)

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<math.h>
int U,C,K,c,d,S,zz;char*bpe;typedef struct{float*i;int j,k;} A;void*E
#define N(i,j)for(int i=0; i<j; i++)
A o(int j,int k,int i){float*a=E;E+=S=4*j*k;memset(a,0,S*i);A R={ a,j
#define I(R,B)A R(A a,float k){ N(i,a.j*a.k){ float b=a.i[i]; a.i[i]=
I(l,b/k)I(q,b+k)I(u,1./sqrt(b))I(z,exp(b))I(r,a.i[(i/a.k)*a.k])I(P,(i
#define F(R,B)A R(A a,A b){ N(i,a.j*a.k){ a.i[i]=a.i[i]B b.i[i]; } re
F(V,+)F(v,*)F(H,/)F(at,+b.i[i%a.k];)F(mt,*b.i[i%a.k];)A X(A a){A R=o(
A g(A a,A b){A R=o(a.j,b.j,!c);{for(int i=c;i<d;i++){for(int j=0;j<b.
V(o(R.j,R.k,1),R);}A J(A a,int b,int j,int k){A R={ a.i+b*j,j,k} ;ret
#define G(a,i)at(g(a,f[i+1]),f[i])
A m(int j,int k){j+=!j;k+=!k;A a=o(j,k,1);fread(a.i,S,1,fp);return p(
int t;int Y(char*R){if(!*R)return 0;int B=1e9,r;N(i,5e4){if(bpe[999*i
int main(int S,char**D){S=D[1][5]+3*D[1][7]+3&3;K=12+4*S+(S>2);U=K*64
bpe=malloc(1e9);fp=fopen(D[2],"r");unsigned char a[S=999],b[S];N(i,5e
fp=fopen(D[1],"r");A\
 x[999];A*R=x;N(i,C){N(j,12)*R++=m(U+U*(j?j^8?j^11?0:3:3:2),U*((j%8==
while(1){char W[1000]={ 0} ;int T;strcat(W,"\nAlice: ");printf("\n%s:
```

```
while(1){E=n;T=d+32-d%32;c*=!!(d%32);A O=o(T,U,1);N(i,d){N(j,U)O.i[i*
if(bpe[S*999]==10)break;printf("%s",bpe+S*999);fflush(stdout);}}}
```

Контекст: ChatGPT и трансформеры

Тем, кто в танке — напомню, что ChatGPT — это такое приложение. В нём вы можете общаться с так называемой «(большой) языковой моделью» как с собеседником-человеком. Она удивительно хорошо поддерживает разговор, а GPT-4, новейшая модель, лежащая в основе ChatGPT, вообще невероятно впечатляет.

В данной программе на C воспроизведено поведение ChatGPT, но при помощи гораздо более слабой модели GPT-2, появившейся ещё в 2019 году. Несмотря на то, что номер версии у неё всего на 2 меньше, чем у GPT-4, возможности их просто несопоставимы — зато модель GPT-2 является опенсорсной. Поэтому с ней и будем работать.

GPT-2 — это тип моделей машинного обучения, так называемый «трансформер» . Такие нейронные сети принимают на вход фиксированную последовательность слов, после чего прогнозируют слово, которое должно идти следующим. Вновь и вновь повторяя эту процедуру, можно генерировать при помощи трансформера лексические последовательности произвольной длины.

В этом посте я не собираюсь делать настолько полный экскурс в машинное обучение, после которого вы бы уяснили, *почему* трансформер спроектирован так, а не иначе. Далее просто будет описано, как именно работает вышеприведённый код на C.

Разбор кода C

Начнём: матричная математика (700 байт)

Вся «картина мира» нейронной сети заключена в матричных операциях. Поэтому для начала нам потребуется соорудить библиотеку для работы с матрицами, потратив на это минимум байт.

Вот совершенно минимальное определение матрицы:

```
typedef struct {
  float* dat;
  int rows, cols;
} Matrix;
```

Для начала отметим, что, пусть нам и требуется реализовать целый ряд различных операций, все они делятся на два принципиальных «типа»:

1. Матрично-константные операции (напр., прибавить 7 к каждой из записей в матрице)

2. Матрично-матричные операции (напр., сложить соответствующие матричные записи)

Благодаря такому сходству можно воспользоваться макросами и вытянуть некоторую общую логику в метапроцедуру, в которой прописано, например, как обращаться с парами матриц. В таком случае детали конкретных операций будут зависеть от реализации.

Чтобы сделать это на C, определим функцию

```
#define BINARY(function, operation)
```

как

```
Matrix FUNCTION(Matrix a, Matrix b) {
  for (int i = 0; i < a.rows; i++) {
```

```
    for (int j = 0; j < a.cols; j++) {
      a[i*a.cols + j] = a[i*a.cols + j] OPERATION b[i*a.cols+j];
    }
  }
  return a;
}
```

Что, например, позволит нам написать

```
BINARY(matrix_elementwise_add, +);
BINARY(matrix_elementwise_multiply, *);
```

и предусмотреть возможность расширения до полной операции, в рамках которой происходило бы поэлементное сложение или перемножение двух матриц. Определю ещё несколько операций, понять которые не составляет труда:

Теперь остановимся на принятых в C `#defines`. В сущности, это просто расфуфыренные регулярные выражения. Поэтому, когда запустим такой `#define`, в программе на самом деле произойдёт

```
a[i*a.cols + j] = a[i*a.cols + j] OPERATION b[i*a.cols+j];
```

что в случае с умножением расширится до

```
a[i*a.cols + j] = a[i*a.cols + j] * b[i*a.cols+j];
```

На первый взгляд этот код выглядит довольно запутанно — например, что здесь делает эта точка с запятой? Но, если вы просто замените код регулярным выражением, то увидите, как он расширяется до

```
  a[i*a.cols + j] = a[i*a.cols + j] + b.dat[i%a.cols] ; b[i*a.cols+j]
```

Поскольку второе выражение здесь ничего не делает, этот код фактически эквивалентен

```
a[i*a.cols + j] = a[i*a.cols + j] + b.dat[i%a.cols] ; b[i*a.cols+j];
```

a[i*a.cols + j] = a[i*a.cols + j] + b.dat[i%a.cols] ; b[i*a.cols+j];

(ПОЛЬЗУЙТЕСЬ ЯЗЫКАМИ С КАЧЕСТВЕННЫМИ МАКРОСАМИ. LISP НЕ ВСЕГДА ЛУЧШЕ C!)

Быстрое перемножение матриц (300 байт)

Базовая реализация перемножения матриц совершенно проста: мы всего лишь реализуем тривиальные циклы с кубической вычислительной сложностью. (В моём примере с перемножением матриц нет ничего особенного. Если вы умеете быстро перемножать матрицы, то просто следите за кодом).

```
Matrix matmul(Matrix a, Matrix b) {
  Matrix out = NewMatrix(a.rows, b.rows);
  for (int i = 0; i < a.rows; i++)
    for (int j = 0; j < b.rows; j++)
      for (int k = 0; k < a.cols; k++)
        out.dat[i * b.rows + j] += a.dat[i * a.cols + k+k2] * b.dat[(


  return out;
}
```

К счастью, эту процедуру можно значительно ускорить, сделав её лишь немного умнее. Учитывая, как именно на большинстве компьютеров работают память и кэш, можно (значительно!) ускорить работу, если многократно читать один и тот же фрагмент памяти, а также записывать в него.

```
Matrix matmul_t_fast(Matrix a, Matrix b) {
  Matrix out = NewMatrix(a.rows, b.rows);
  for (int i = 0; i < a.rows; i++)
    for (int j = 0; j < b.rows; j += 4)
      for (int k = 0; k < a.cols; k += 4)
        for (int k2 = 0; k2 < 4; k2 += 1)
          for (int j2 = 0; j2 < 4; j2 += 1)
            out.dat[i * b.rows + j+j2] += a.dat[i * a.cols + k+k2] *


  return out;
}
```

Позже мы внесём ещё одно изменение в механизм логического вывода, а также добавим к перемножению матриц ещё один параметр, который позволит нам лишь частично умножать матрицу A на матрицу B. Это полезно в случаях, когда мы успели предварительно вычислить часть произведения.

Слои нейронной сети (300 байт)

Чтобы написать трансформер, потребуется определить несколько специфических слоёв нейронной сети. Один из них — это функция активации GELU, которую можете воспринимать как колдовскую.

```
UNARY(GELU, b / 2 * (1 + tanh(.7978845 * (b + .044715 * b * b * b))))
```

Также я реализую функцию, задающую нижнюю диагональ матрицы (после возведения значений в степень). Это пригодится нам для так называемого *причинного внимания*: мы собираемся учитывать только прошлое, но не будущее, поэтому при помощи данной функции устанавливаем нижнюю диагональ матрицы внимания в ноль.

```
UNARY(tril, (i/k<i%(int)k) ? 0 : exp(b/8))
```

Наконец, нам понадобится функция нормализации слоёв (ещё одна магическая вещь, о которой можете сами почитать подробнее, если захотите. В сущности, она нормализует среднее и дисперсию каждого слоя).

```
Matrix LayerNorm(Matrix a, int i) {
  Matrix b = add(a, divide_const(sum(a), -a.cols));
  Matrix k = divide_const(sum(multiply(
    add(NewMatrix(b.rows,b.cols,1),b), b)), b.cols-1);
  Matrix out = add_tile(multiply_tile(
    multiply(add(NewMatrix(b.rows,b.cols,1),b),
    mat_isqrt(add_const(k, 1e-5),0)), layer_weights[i+1]),
                        layer_weights[i]);


  return out;
}
```

Последний элемент модели — это линейная функция, просто перемножающая матрицы и плюсующая сдвиг (с разбиением на макрооперации — тайлингом).

```
#define Linear(a, i) add_tile(matmul_t_fast(a, layer_weights[i+1]), l
```

Архитектура трансформера (600 байт)

Решив все эти вопросы, мы, наконец, сможем реализовать наш трансформер всего в 600 байтах.

```
for (int i = 0; i < NLAYER; i++) {
  layer_weights = weights + 12*permute;


  // Вычисляем ключи, запросы и значение — всё за одну большую операц
  Matrix qkv = transpose(slice(Linear(LayerNorm(line, 4), 0), 0, T*3,


  // Освобождаем место для вывода из вычислений
  Matrix result = NewMatrix(DIM, T, 1);


  for (int k = 0; k < NHEAD; k++) {
    // Распределяем qkv на три вычислительные головы
    Matrix merge = transpose(slice(qkv, k*3, 64*T, 3)),
      // Получаем произведение запросов и ключей, а затем возводим ре
      a = tril(matmul_t_fast(transpose(slice(merge, 0, 64, T)),
                             transpose(slice(merge, T, 64, T))), T),
      // наконец, умножаем вывод softmax (a/sum(a)) на матрицу значее
      out = transpose(matmul_t_fast(divide(a, sum(a)), slice(merge, T
    // и копируем вывод в ту часть результирующей матрицы, где он дол
    memcpy(result.dat+64*T*k, out.dat, 64*T*4);
  }


  // Остаточная связь
  line = add(line,Linear(transpose(result), 2));


  // Функция активации и остаточная связь
  line = add(line, Linear(GELU(Linear(LayerNorm(line, 6), 8), 0), 10)
```

```
}


// Сбросить веса слоёв так, чтобы последний слой можно было взять за
layer_weights = weights;
line = LayerNorm(line, 12*NLAYER);


Matrix result = matmul_t_fast(transpose(slice(line, tmp-1, DIM, 1)),
```

Теперь озвучу один момент о логическом выводе в трансформерах, для вас, возможно, полностью очевидный. Когда вы уже вызвали модель, приказав ей сгенерировать один токен, вам не приходится перевычислять всю функцию для генерации следующего токена. На самом деле, для генерации каждого последующего токена требуется выполнить лишь минимум работы.

Дело в том, что как только вы вычислили вывод трансформера для всех токенов вплоть до N-го, вы можете повторно использовать почти весь этот вывод для вычисления N+1го токена (выполнив еще немного работы.)

Чтобы всё это реализовать, я выстроил все операции выделения памяти у меня в коде последовательно, в пределах одного и того же блока памяти. Так гарантируется, что при любой операции матричного умножения будет задействована одна и та же память. Соответственно, на каждой итерации цикла я могу не обнулять память перед тем, как задействовать её на следующей итерации, и в памяти уже будет содержаться результат предыдущей итерации. Мне просто потребуется выполнить вычисление для N+й строки.

Частотное кодирование (400 байт)

Проще всего выстроить языковую модель на базе последовательности слов. Но, поскольку общее множество слов, в сущности, не ограничено, для любой языковой модели требуется ввод фиксированного размера, где понадобится заменить достаточно редкие слова специальным токеном [OUT OF DISTRIBUTION] (вне обучающего распределения). Это нехорошо.

Да, существует простейшее средство, позволяющее с этим справиться — использовать модели, работающие «на уровне символов» и поэтому знающие только отдельные буквы. Но здесь же возникает проблема: фактически, такой модели придётся изучать значение каждого слова с чистого листа. Также из-за этого сузится реальный размер контекстного окна языковой модели, и коэффициент такого снижения равен средней длине слова.

Во избежание таких проблем модели вроде GPT-2 создают токены из «подслов». Некоторые слова могут быть токенами сами по себе, но редкие слова дополнительно дробятся. Например, слово «nicholas» можно подразделить на «nich», «o», «las».

Реализовать общий алгоритм для этой цели не составляет труда: берём слово, которое хотим токенизировать, и сначала разделяем его на отдельные символы. Затем ищем пары таких смежных токенов, которые можно было бы объединить, и если находим — объединяем. Повторяем процедуру до тех пор, пока вариантов для дальнейшего слияния не останется.

При всей простоте этот алгоритм, к сожалению, очень трудно реализовать на C, так как для него требуется многократно выделять память и отслеживать развитие древовидной структуры токенов.

Поэтому в таком случае мы превращаем достаточно простой алгоритм с линейной сложностью в алгоритм с потенциально экспоненциальной сложностью, зато пишем гораздо меньше кода. Базовая идея будет

раскрываться примерно так, как показано в этом C-подобном псевдокоде:

```
word_tokenize(word) {
  if len(word) == 0 { return (0, 0); }
  result = (1e9, -1);
  for (int i = 0; i < VOCAB_LEN; i++) {
    if (is_prefix(bpe[i]), word) {
      sub_cost = word_tokenize(word+len(bpe[i]))[0] + i + 1e7;
      result = min(result, (sub_cost, i));
    }
  }
  return result;
}
```

То есть, чтобы токенизировать слово, требуется проверить все возможные слова из словаря и узнать, не является ли оно префиксом актуального слова. Если является, то мы возьмём его в качестве первого токена, а затем попытаемся таким же образом рекурсивно токенизировать всё слово. Мы будем следить, какой вариант токенизации получается наилучшим (об этом судим по длине, ничейные результаты распределяем по индексу токена в словаре) — и именно его возвращаем.

Загрузка весов (300 байт)

Почти готово! Последнее, что нам осталось сделать — загрузить с диска в нейронную сеть фактические веса. Это на самом деле не сложно, поскольку веса хранятся в простом двоичном формате, который легко считывается в C. Фактически, это совершенно плоская сериализация 32-разрядных чисел с плавающей точкой.

Единственное, что требуется узнать — насколько велики различные матрицы. К счастью, это также легко выяснить. Каков бы ни был размер

модели GPT-2, архитектура у них у всех одинакова, и веса хранятся в одном и том же порядке. Поэтому нам всего лишь требуется прочитать с диска правильно оформленные матрицы.

Но вот напоследок ложка дёгтя. Слои нейронной сети не хранятся на диске в том порядке, в котором следовало бы ожидать: сначала слой 0, затем слой 1, далее слой 2. На самом деле, первым идёт слой 0, за ним слой 1, а потом слой .... ДЕСЯТЬ! (и далее слой 11, а за ним 12.) Дело в том, что при сохранении веса сортируются по лексикографическому принципу. А лексикографически «0» предшествует «1», но «10» предшествует «2». Поэтому нам потребуется немного поработать, чтобы переставить веса в правильном порядке. Для этого напишем следующий код

```
int permute;
tmp=0;
for (int j = 0; j < 10; j++) {
  if (j == i) {
    permute = tmp;
  }
  tmp++;
  for (int k = 0; k < 10*(j>0); k++) {
    if (j*10+k < NLAYER && tmp++ && i == j*10+k) {
      permute = tmp;
    }
  }
}
```

Загрузка данных для частотного кодирования (300 байт)

Чтобы фактически осуществить частотное кодирование, мы сначала должны загрузить с диска словарь с соответствующими байтовыми парами. В идеале хотелось бы иметь список всех слов из словаря, сохранённых в каком-нибудь вменяемом C-читаемом формате. Но

поскольку исходный файл (a) рассчитан на чтение в Python и (b) не предназначен для лёгкого синтаксического разбора минимальных байтовых фрагментов, нам здесь придётся потрудиться.

Логично предположить, что формат файла предусматривает просто список идущих друг за другом слов, но на самом деле информация в нём закодирована в виде списка байтовых пар. То есть, мы сможем прочитать в качестве одного токена не «Hello», а строку «H» «ello». Таким образом, нам следует объединить токены «H» и «ello» в один токен «Hello».

Другая проблема заключается в том, что файл представлен в сглаживающей кодировке UTF-8 (с оговорками) и ... на то есть причина. Все символы ascii, которые можно вывести на печать, кодируются сами по себе, а «невыводимые» символы 0-31 кодируются в формате 188+символ. Так, например, пробел кодируется в виде токена «Ġ». Но вот проблема: в кодировке UTF8 на диске символу «Ġ» соответствует 0xc4 0xa0. Поэтому при считывании нам придётся приложить усилия, чтобы преобразовать этот символ обратно в пробел.

Притом, что всё это, в принципе, ничто из этого не сложно, для всех этих операций требуется написать много кода. Это, конечно, раздражает, когда стремишься к максимальной компактности.

```
unsigned char a[tmp=999],b[tmp];
for (int i = 0; i < 5e4; i++) {
  int k = i*tmp;
  if (i < 93) {
    // Первые 92 токена – это просто символы ascii, выводимые на печа
    bpe[k] = i + 33;
    bpe[k+1] = 0;
  } else if (i > 254) {
    // Те, что сверх 254, взяты из файла BPE. Загружаем их
    fscanf(fp, "%s %s", a, b);
    strcat((char*)a, (char*)b);
```

```
    int j = 0;
  for (int i = 0; a[i]; i++) {
    // Кодировка UTF8 усложняет жизнь, поэтому обрабатываем её здесь
      bpe[k+j++] = a[i] ^ 196 ? a[i] : a[++i]-128;
    }
    bpe[k+j++] = 0;
  } else if (i > 187) {
    // Токены выше 187 — это не выводимые на печать символы asii в ди
    bpe[k] = i-188;
    bpe[k+1] = 0;
  }
}
```

Заключение

В самом деле, примечательно, как можно сконцентрировать в нескольких тысячах байт целые десятилетия в развитии машинного обучения. В сущности, здесь есть всё необходимое (кроме фактических весов модели), чтобы вы могли запустить любую современную нейронную сеть. Притом, что я реализовал этот проект в основном из интереса, он хорошо демонстрирует, насколько, в самом деле, *просты* нейронные сети.

Теги: gpt, трансформеры, C, нейронные сети

Хабы: Машинное обучение, C, Программирование, Занимательные задачки, Искусственный интеллект