让我们编码 LLM

最后修改日期：2024 年 2 月 22 日

本文的完整结构化代码可以从github.com/waylandzhang/Transformer-from-scratch ↗ 下载。

在本文中，我们将从头开始实现类似 GPT 的转换器。我们将按照我上一篇文章中描述的步骤对每个部分进行编码。

那么让我们开始吧。

准备环境

安装依赖项

```
pip install numpy requests torch tiktoken matplotlib pandas
```

```
import os
import requests
import pandas as pd
import matplotlib.pyplot as plt
import math
import tiktoken
import torch
import torch.nn as nn
```

设置超参数

超参数是模型的外部配置，无法在训练期间从数据中学习。它们是在训练过程开始之前设置的，在控制训练算法的行为和训练模型的性能方面起着至关重要的作用。

```
# Hyperparameters
batch_size = 4  # How many batches per training step
context_length = 16  # Length of the token chunk each batch
d_model = 64  # The vector size of the token embeddings
num_layers = 8  # Number of transformer blocks
num_heads = 4  # Number of heads in Multi-head attention # 我们的代码中通过
learning_rate = 1e-3  # 0.001
dropout = 0.1 # Dropout rate
max_iters = 5000  # Total of training iterations
eval_interval = 50  # How often to evaluate the model
eval_iters = 20  # How many iterations to average the loss over when evalu
device = 'cuda' if torch.cuda.is_available() else 'cpu'  # Instead of usir

TORCH_SEED = 1337
torch.manual_seed(TORCH_SEED)
```

准备数据集

和我们的示例一样，我们将使用一个小数据集进行训练。该数据集是一个包含销售教科书的文本文件。我们将使用该文本文件训练可以生成销售文本的语言模型。

```
# download a sample txt file from https://huggingface.co/datasets/goendali
if not os.path.exists('sales_textbook.txt'):
    url = 'https://huggingface.co/datasets/goendalf666/sales-textbook_for_
    with open('sales_textbook.txt', 'w') as f:
        f.write(requests.get(url).text)

with open('sales_textbook.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

步骤 1：标记化

我们将使用tiktoken ↗库对数据集进行标记。该库是一个快速且轻量级的标记器，可用于将文本标记为标记。

```
# Using TikToken to tokenize the source text
encoding = tiktoken.get_encoding("cl100k_base")
tokenized_text = encoding.encode(text) # size of tokenized source text is
vocab_size = len(set(tokenized_text)) # size of vocabulary is 3,771
max_token_value = max(tokenized_text)

print(f"Tokenized text size: {len(tokenized_text)}")
print(f"Vocabulary size: {vocab_size}")
print(f"The maximum value in the tokenized text is: {max_token_value}")
```

打印输出：

```
Tokenized text size: 77919
Vocabulary size: 3771
The maximum value in the tokenized text is: 100069
```

第 2 步：词嵌入

我们将数据集分为训练集和验证集。训练集将用于训练模型，验证集将用于评估模型的性能。

```
# Split train and validation
split_idx = int(len(tokenized_text) * 0.8)
train_data = tokenized_text[:split_idx]
val_data = tokenized_text[split_idx:]

# Prepare data for training batch
# Prepare data for training batch
data = train_data
idxs = torch.randint(low=0, high=len(data) - context_length, size=(batch_s
x_batch = torch.stack([data[idx:idx + context_length] for idx in idxs])
y_batch = torch.stack([data[idx + 1:idx + context_length + 1] for idx in :
print(x_batch.shape, x_batch.shape)
```

打印输出（训练输入x和y的形状）：

```
torch.Size([4, 16]) torch.Size([4, 16])
```

步骤 3：位置编码

我们将使用一个简单的嵌入层将输入标记转换为向量。

```
# Define Token Embedding look-up table
token_embedding_lookup_table = nn.Embedding(max_token_value, d_model)

# Get X and Y embedding
x = token_embedding_lookup_table(x_batch.data)
y = token_embedding_lookup_table(y_batch.data)
```

现在，我们的输入 x 和 y 都是形状 (batch_size, context_length, d_model)。

```
# Get x and y embedding
x = token_embedding_lookup_table(x_batch.data) # [4, 16, 64] [batch_size,
y = token_embedding_lookup_table(y_batch.data)
```

应用位置嵌入

正如原始论文中所述，我们将使用正弦和余弦来生成位置嵌入表，然后将这些位置信息添加到输入嵌入标记中。

```
# Define Position Encoding look-up table
position_encoding_lookup_table = torch.zeros(context_length, d_model) # ir
position = torch.arange(0, context_length, dtype=torch.float).unsqueeze(1)
# apply the sine & cosine
div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(100(
position_encoding_lookup_table[:, 0::2] = torch.sin(position * div_term)
position_encoding_lookup_table[:, 1::2] = torch.cos(position * div_term)
position_encoding_lookup_table = position_encoding_lookup_table.unsqueeze(

print("Position Encoding Look-up Table: ", position_encoding_lookup_table.
```

打印输出：

```
Position Encoding Look-up Table:  torch.Size([4, 16, 64])
```

然后，将位置编码添加到输入嵌入向量中。

```
# Add positional encoding into the input embedding vector
input_embedding_x = x + position_encoding_lookup_table # [4, 16, 64] [batc
input_embedding_y = y + position_encoding_lookup_table

X = input_embedding_x

x_plot = input_embedding_x[0].detach().cpu().numpy()
print("Final Input Embedding of x: \n", pd.DataFrame(x_plot))
```

*现在，我们得到了**X 的最终输入嵌入**，它是要输入到转换器块的值：*

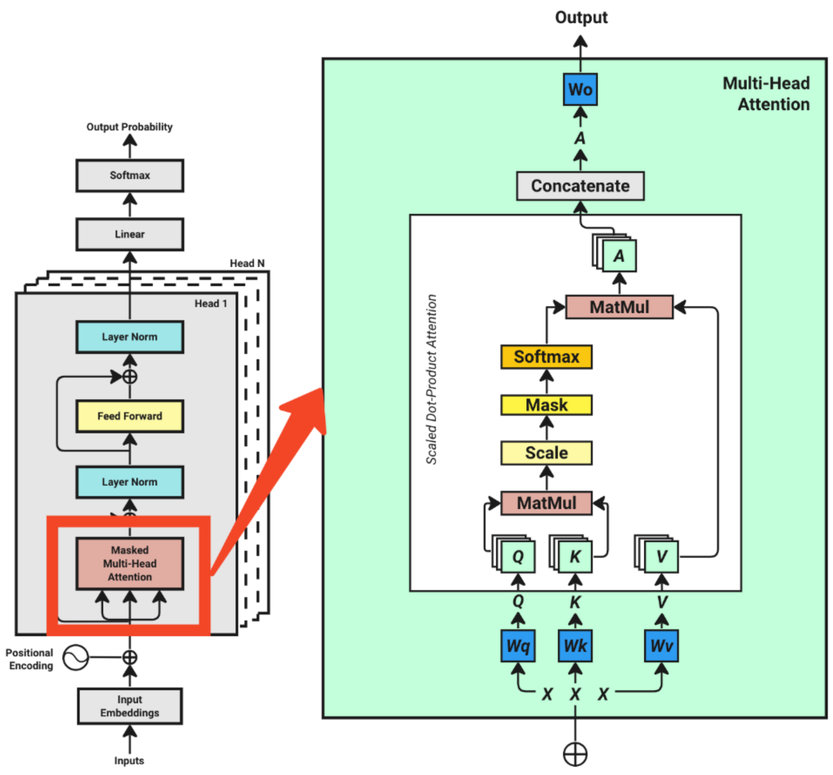
```
Final Input Embedding of x:
           0         1         2         3         4         5         6
0  -1.782388  1.200549 -0.177262  0.278616 -1.322919  0.929397 -0.178307
1   0.434183  2.051380  0.642167  1.294858  0.287493 -0.132648 -0.388530
2   0.180579 -0.714483  0.983105 -0.944209  1.182870 -0.100558  0.807144
3  -0.249654 -3.228277 -0.017824  0.492374  0.992460 -1.281102  0.811163
4   1.533710 -1.794257 -0.115366 -2.216147  0.143978 -2.549789  0.285271
5  -2.187219 -0.290028 -0.914946 -0.614617 -0.033163 -1.060609  2.265111 -
6  -1.015749  1.862600  0.785039  2.425240  0.613279 -1.725359  1.288837 -
7  -0.446721  1.136845  0.336349  1.287424  1.515973  0.814479  0.233362 -
8  -0.190227 -0.968500 -1.648937  2.915030 -3.227971 -0.739308 -0.485671 -
9  -0.224795 -0.326915 -0.362390  1.489488 -0.389251 -0.362224  0.913598 -
10 -0.692015 -1.732475  2.214933 -1.991298 -0.398353  1.266861 -1.057534 -
11  0.713966 -0.232073  2.291222  0.224710 -1.199412  0.022869 -1.532023 -
12 -0.468460  1.830461  1.335220 -1.410995  0.069466  1.672440 -1.680814 -
13  1.436239  0.494849  1.781795  0.060173  0.538164  1.890070 -2.363284
14 -0.113100  0.519679  0.316672  0.299135  3.229518  1.496113 -0.325615
15  0.301521  0.997564 -0.672755 -1.215677  0.949777  0.474997 -0.279164

[16 rows x 64 columns]
```

注意：y 嵌入向量将与我们的 x 具有相同的形状。

步骤 4：变压器块

4.1 多头注意力概述

让我们回顾一下多头注意力图。



现在我们有了输入嵌入**X**，我们可以开始实现多头注意力模块了。实现多头注意力模块需要一系列步骤。让我们一一编写代码。

4.2 准备Q、K、V

```
# Prepare Query, Key, Value for Multi-head Attention

query = key = value = X # [4, 16, 64] [batch_size, context_length, d_model

# Define Query, Key, Value weight matrices
Wq = nn.Linear(d_model, d_model)
Wk = nn.Linear(d_model, d_model)
Wv = nn.Linear(d_model, d_model)

Q = Wq(query) #[4, 16, 64]
Q = Q.view(batch_size, -1, num_heads, d_model // num_heads)  #[4, 16, 4, 1

K = Wk(key) #[4, 16, 64]
K = K.view(batch_size, -1, num_heads, d_model // num_heads)  #[4, 16, 4, 1

V = Wv(value) #[4, 16, 64]
V = V.view(batch_size, -1, num_heads, d_model // num_heads)  #[4, 16, 4, 1
```

然后，我们将 Q、K、V 重塑为 [batch_size、num_heads、context_length、head_size] 以进行进一步计算。

```
# Transpose q,k,v from [batch_size, context_length, num_heads, head_size]
# The reason is that treat each batch with "num_heads" as its first dimens
Q = Q.transpose(1, 2) # [4, 4, 16, 16]
K = K.transpose(1, 2) # [4, 4, 16, 16]
V = V.transpose(1, 2) # [4, 4, 16, 16]
```

4.3 计算QK^T Attention

通过使用 torch.matmul 函数可以非常轻松地完成此操作。

```
# Calculate the attention score betwee Q and K^T
attention_score = torch.matmul(Q, K.transpose(-2, -1))
```

4.4 规模

```
# Then Scale the attention score by the square root of the head size
attention_score = attention_score / math.sqrt(d_model // num_heads)
```

我们实际上可以在一行中重写 4.3 和 4.4：

```
attention_score = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(d_model
print(pd.DataFrame(attention_score[0][0].detach().cpu().numpy()))
```

打印输出 *（一批的一个序列）*：

```
0         1         2         3         4         5         6         7
0   0.105279 -0.365092 -0.339839 -0.650558 -0.464043 -0.531401  0.437939 -
1  -0.302636  0.525435  0.863502 -0.218539  0.600691 -0.413970  0.408111
2   0.201820  0.156336 -0.245585  0.101653  0.228243 -0.565197  0.589193 -
3   0.517998 -0.303064  0.484515 -0.399551 -0.004528 -0.028223 -0.602194
4   0.519200  0.322036  0.328027 -0.031755  0.006269  0.133609 -0.095071 -
5  -0.635913  0.137114  0.083046  0.234778 -0.668992 -0.366838 -0.613126
6  -0.287777 -0.841604 -0.128455 -0.566180  0.079559 -0.530863 -0.082675
7  -0.798519 -0.905525  0.317880 -0.176577  0.751465 -0.564863  1.014724 -
8   0.141614 -0.022169  0.152088 -0.519404 -0.069152 -0.880496 -0.229767 -
9  -0.097493  0.860376  0.073501  0.150553 -0.651579 -0.376676 -0.691368
10 -0.469827  0.302545 -0.015767 -0.175387 -0.049927 -0.706852  0.511237
11 -0.335474 -0.399539 -0.093901 -0.682290  0.312682 -0.310319  0.344753
12 -1.757631 -0.967825 -0.516159 -0.246766 -0.352132 -0.780370 -0.262975 -
13  0.550975  0.313643 -0.074468  0.519995 -0.149188 -0.565922  0.199527 -
14 -0.616347 -0.812240  0.245260  0.167278  0.913596 -0.493119  1.139083 -
15 -0.145391  0.514632 -0.296119 -0.038103 -0.187110 -0.634636  0.509902 -
```

4.5 掩码

```
# Apply Mask to attention scores
attention_score = attention_score.masked_fill(torch.triu(torch.ones(atten
print(pd.DataFrame(attention_score[0][0].detach().cpu().numpy()))
```

打印输出 *（一批的一个序列）*：

```
0         1         2         3         4         5         6         7
0   0.105279      -inf      -inf      -inf      -inf      -inf      -inf
1  -0.302636  0.525435      -inf      -inf      -inf      -inf      -inf
2   0.201820  0.156336 -0.245585      -inf      -inf      -inf      -inf
3   0.517998 -0.303064  0.484515 -0.399551      -inf      -inf      -inf
4   0.519200  0.322036  0.328027 -0.031755  0.006269      -inf      -inf
5  -0.635913  0.137114  0.083046  0.234778 -0.668992 -0.366838      -inf
6  -0.287777 -0.841604 -0.128455 -0.566180  0.079559 -0.530863 -0.082675
7  -0.798519 -0.905525  0.317880 -0.176577  0.751465 -0.564863  1.014724 -
8   0.141614 -0.022169  0.152088 -0.519404 -0.069152 -0.880496 -0.229767 -
9  -0.097493  0.860376  0.073501  0.150553 -0.651579 -0.376676 -0.691368
10 -0.469827  0.302545 -0.015767 -0.175387 -0.049927 -0.706852  0.511237
11 -0.335474 -0.399539 -0.093901 -0.682290  0.312682 -0.310319  0.344753
12 -1.757631 -0.967825 -0.516159 -0.246766 -0.352132 -0.780370 -0.262975 -
13  0.550975  0.313643 -0.074468  0.519995 -0.149188 -0.565922  0.199527 -
14 -0.616347 -0.812240  0.245260  0.167278  0.913596 -0.493119  1.139083 -
15 -0.145391  0.514632 -0.296119 -0.038103 -0.187110 -0.634636  0.509902 -
```

我们可以看到对角线和上三角被-inf 掩盖了。

4.6 Softmax

```
# Softmax the attention score
attention_score = torch.softmax(attention_score, dim=-1)  #[4, 4, 16, 16]
print(pd.DataFrame(attention_score.detach().cpu().numpy()))
```

打印输出 *（一批的一个序列）*：

```
0         1         2         3         4         5         6         7
0   0.062604  0.062472  0.062478  0.062419  0.062452  0.062439  0.062748
1   0.062377  0.062529  0.062643  0.062387  0.062551  0.062364  0.062499
2   0.062565  0.062551  0.062453  0.062535  0.062574  0.062400  0.062717
3   0.062647  0.062427  0.062634  0.062411  0.062486  0.062480  0.062384
4   0.062635  0.062566  0.062567  0.062473  0.062481  0.062512  0.062460
5   0.062371  0.062501  0.062488  0.062527  0.062367  0.062405  0.062373
6   0.062467  0.062379  0.062504  0.062417  0.062562  0.062422  0.062516
7   0.062358  0.062348  0.062566  0.062444  0.062742  0.062384  0.062900
8   0.062632  0.062573  0.062636  0.062449  0.062559  0.062391  0.062513
9   0.062434  0.062727  0.062467  0.062484  0.062360  0.062391  0.062356
10  0.062419  0.062608  0.062511  0.062473  0.062502  0.062385  0.062693
11  0.062463  0.062450  0.062519  0.062403  0.062655  0.062468  0.062669
12  0.062327  0.062405  0.062489  0.062561  0.062530  0.062435  0.062556
13  0.062685  0.062591  0.062482  0.062671  0.062466  0.062395  0.062554
14  0.062372  0.062352  0.062530  0.062509  0.062806  0.062387  0.062958
15  0.062479  0.062693  0.062448  0.062504  0.062470  0.062394  0.062691
```

应用softmax函数后，分数现在介于0和1之间，并且每行的总和为1。

4.7 计算V注意力

最后，我们将注意力分数与**V**相乘以获得多头注意力模块的输出。

```
# Calculate the V attention output
A = torch.matmul(attention_score, V) # [4, 4, 16, 16] [batch_size, num_hea
print(attention_output.shape)
```

打印输出：

```
torch.Size([4, 4, 16, 16])
```

注意：现在的形状是 [4, 4, 16, 16]，即 [batch_size, num_heads, context_length, head_size]。

4.8 连接并输出

回想上一篇文章，↗我们需要连接多头注意力块的输出并将其输入到线性层。

```
A = A.transpose(1, 2) # [4, 16, 4, 16] [batch_size, context_length, num_he
A = A.reshape(batch_size, -1, d_model) # [4, 16, 64] [batch_size, context_
```

注意：现在的形状是 [4, 16, 64]，即 [batch_size, context_length, d_model]。

现在，我们可以应用另一个 [64,64] 线性层**Wo** *（这是训练期间学习到的权重）*并得到多头注意力模块的最终输出：

```
# Define the output weight matrix
Wo = nn.Linear(d_model, d_model)
output = Wo(A) # [4, 16, 64] [batch_size, context_length, d_model]

print(output.shape)
```

如果我们将其打印出来，形状将恢复为 [4, 16, 64]，这与我们的输入嵌入形状相同。

步骤5：残差连接和层归一化

现在我们有了多头注意力模块的输出，我们可以应用残差连接和层规范化。

```
# Add residual connection
output = output + X

# Add Layer Normalization
layer_norm = nn.LayerNorm(d_model)
output = layer_norm(output)
```

步骤 6：前馈网络

```
# Define Feed Forward Network
output = nn.Linear(d_model, d_model * 4)(output)
output = nn.ReLU()(output)
output = torch.dropout(output, p=dropout, train=True)
```

添加最后的残差连接和层归一化：

```
# Add residual connection
output = output + X
# Add Layer Normalization
layer_norm = nn.LayerNorm(d_model)
output = layer_norm(output)
```

步骤7：重复步骤4至6

上面我们完成的只是一个 Transformer 块。实际操作中，我们会将多个 Transformer 块堆叠在一起，形成一个 Transformer 解码器。

我们实际上应该将代码打包成类并使用 PyTorch nn.Module 来构建我们的 Transformer 解码器。但为了演示，我们只保留一个块。

步骤 8：输出概率

应用最后的线性层来获得我们的逻辑：

```
logits = nn.Linear(d_model, max_token_value)(output)
print(pd.DataFrame(logits[0].detach().cpu().numpy()))
```

最后一步是对logits 进行*softmax*以获得每个 token 的概率：

```
# torch.softmax usually used during inference, during training we use torc
# but for illustration purpose, we'll use torch.softmax here
probabilities = torch.softmax(logits, dim=-1)
```

```
0         1         2         3         4         5         6         7
0   0.000007  0.000008  0.000006  0.000005  0.000004  0.000004  0.000009
1   0.000018  0.000016  0.000006  0.000017  0.000005  0.000006  0.000005
2   0.000013  0.000007  0.000008  0.000003  0.000007  0.000009  0.000021
3   0.000005  0.000013  0.000011  0.000004  0.000006  0.000007  0.000012
4   0.000005  0.000010  0.000008  0.000006  0.000017  0.000005  0.000010
5   0.000008  0.000004  0.000007  0.000003  0.000004  0.000011  0.000018
6   0.000005  0.000008  0.000014  0.000004  0.000007  0.000007  0.000012
7   0.000004  0.000008  0.000003  0.000006  0.000005  0.000019  0.000010
8   0.000002  0.000006  0.000005  0.000004  0.000006  0.000010  0.000008
9   0.000006  0.000005  0.000010  0.000008  0.000019  0.000018  0.000012
10  0.000011  0.000005  0.000008  0.000007  0.000017  0.000009  0.000007
11  0.000011  0.000006  0.000004  0.000005  0.000006  0.000012  0.000009
12  0.000005  0.000010  0.000007  0.000009  0.000007  0.000023  0.000011
13  0.000004  0.000016  0.000010  0.000010  0.000026  0.000008  0.000009
14  0.000003  0.000008  0.000019  0.000007  0.000014  0.000004  0.000009
15  0.000005  0.000010  0.000008  0.000005  0.000011  0.000010  0.000009

[16 rows x 100069 columns]
```

请注意，我们在这里得到的是一个形状为 [16, 100069] 的巨大矩阵，它是整个词汇表中每个标记的概率。

完整工作代码

实际上，多个 Transformer 块将堆叠在一起以执行一次解码事务。在训练期间，输出标记将与基本事实标记进行比较以计算损失。然后重复该过程，直至超参数中定义的**max_iters次。**

我的GitHub ↗中有一个完整可用的 Transformer Decoder 代码，您可以查看。

← 背景

什么是 LLM →