Explanation

[1]:
```
import torch
from time import time
```

1 Explanation for line 332 to 335

$$
\begin{aligned}
f &= Index_{row}(F, I^c) \\
Q &= (f + P) \cdot W^Q \\
&= f \cdot W^Q + P \cdot W^Q \\
&= Index_{row}(F, I^c) \cdot W^Q + P \cdot W^Q \\
&= Index_{row}(F \cdot W^Q, I^c) + P \cdot W^Q
\end{aligned}
$$

1.1 Variables

[2]:
```
device = "cuda"

## n is the size of character set.
n = 64

## d_emb denotes the embedding dimension.
d_emb = 512

## F denotes the all embeddings in the character embedding layer.
## F shape: (n, d_emb)
character_embedding_layer = torch.nn.Embedding(n, d_emb, padding_idx=0).
 ↪to(device)
F = character_embedding_layer.weight

def generate_position_embeddings(max_length, d_emb):
    """Generate position embeddings, according to the method mentioned in
"Attention is all you need", from Vaswani et al"""
    d_emb_half = d_emb >> 1
    frequencies = torch.pow(torch.tensor([1e4]), -1 / d_emb_half).
 ↪repeat([d_emb_half])
    frequencies[0] = 1.0
    frequencies = frequencies.cumprod(-1)
```

```
    positions = torch.arange(0, max_length)
    phases = torch.einsum("i, j->ij", positions, frequencies)
    position_embeddings = torch.zeros([max_length, d_emb])
    position_embeddings[:, 0::2] = torch.sin(phases)
    position_embeddings[:, 1::2] = torch.cos(phases)
    return position_embeddings

## l is the max length of the words
l = 32

## P denotes the all position embeddings.
## P shape: (l, d_emb)
P = generate_position_embeddings(l, d_emb).to(device)

## IC is the character index for words.
## Here, we randomly sample 500x32 integers from 0 to 63 as the index,
## which means that there are 500 words, and the maximum length of word is 32␣
 ↪characters.
## In practice, we will generate the index according to the character sequence␣
 ↪of the word,
## and fill 0 to fit the maximum length.
## For example, if maximum length of word is 8, the word "apple" will be encoded␣
 ↪as
## [1, 16, 16, 12, 5, 0, 0, 0]. (a: 1, p: 16, l: 12, e: 5)
## IC shape: [500, 32]
IC = torch.randint(low=0, high=64, size=[500, 32]).to(device)

## WQ, WK, WV is the weight matrix for calculating Q, K, V.
WQ = torch.nn.Linear(in_features=d_emb, out_features=d_emb, bias=False).
 ↪to(device)
WK = torch.nn.Linear(in_features=d_emb, out_features=d_emb, bias=False).
 ↪to(device)
WV = torch.nn.Linear(in_features=d_emb, out_features=d_emb, bias=False).
 ↪to(device)
```

1.2 Index at first, then dot product

$$f = Index_{row}(F, I^c)$$
$$Q = (f + P) \cdot W^Q$$

[3]:
```
## character_embedding_layer(IC) means Index_row(F, I^C)
## f shape: (500, l, d_emb)
f = character_embedding_layer(IC)

## WQ(f + P) means (f + P) \cdot W^Q
```

```
## Q_index_at_first shape: (500, l, d_emb)
Q_index_at_first = WQ(f + P.unsqueeze(0))
```

1.3 Dot product at first, then index

$$
\begin{aligned}
Meta^{Q} &= F \cdot W^{Q} \\
Pos^{Q} &= P \cdot W^{Q} \\
Q &= Index_{row}(Meta^{Q}, I^{c}) + Pos^{Q}
\end{aligned}
$$

[4]:
```
def index(matrix, index, dim):
    shape_head = matrix.shape[:dim]
    shape_tail = list(matrix.shape[dim:])
    shape_tail.pop(0)
    new_shape = [*shape_head, *index.shape, *shape_tail]
    return matrix.index_select(dim, index.reshape(-1)).reshape(new_shape)

## WQ(F) means F \cdot W^Q
## metaQ shape: (n, d_emb)
metaQ = WQ(F)

## WQ(P) means P \cdot W^Q
## posQ shape: (l, d_emb)
posQ = WQ(P)

## Q_dot_product_at_first shape: (500, l, d_emb)
Q_dot_product_at_first = index(metaQ, IC, -2) + posQ.unsqueeze(0)
```

1.4 The mean absolute error(MAE) between two method

The MAE is about 2.2×10^{-7}, mean that the results of the two methods are almost equal.

[5]:
```
mae = torch.mean(torch.abs(Q_index_at_first - Q_dot_product_at_first))
print(mae.item())
```

```
2.195932609083684e-07
```

1.5 Efficiency comparison

[6]:
```
start = time()

for i in range(10000):
    f = character_embedding_layer(IC)
    Q_index_at_first = WQ(f + P.unsqueeze(0))

wall_time_index_at_first = time() - start
```

```
print(f"Index at first costs {wall_time_index_at_first:.2f} seconds.")
```

```
Index at first costs 11.20 seconds.
```

[7]:
```
start = time()

for i in range(10000):
    metaQ = WQ(F)
    posQ = WQ(P)
    Q_dot_product_at_first = index(metaQ, IC, -2) + posQ.unsqueeze(0)

wall_time_dot_product_at_first = time() - start
print(f"Dot product at first costs {wall_time_dot_product_at_first:.2f} seconds.
 ↪")
```

```
Dot product at first costs 4.38 seconds.
```

2 Explanation for index based scale dot-product attention(ISDPA, Page 5, Algorithm 1)

2.1 Scale dot-product attention (SDPA)

$$Attention(Q,K,V) = softmax\left(\frac{QK^T}{\sqrt{d_{emb}}}\right)V$$

2.2 QK^T can be computed by index

$$\begin{aligned} f &= Index_{row}(F, I^c) \\ Q &= (f+P)\cdot W^Q \\ K &= (f+P)\cdot W^K \\ Meta^Q &= F\cdot W^Q \\ Meta^K &= F\cdot W^K \\ Pos^Q &= P\cdot W^Q \\ Pos^K &= P\cdot W^K \end{aligned}$$

$$
\begin{aligned}
QK^T &= ((f+P)\cdot W^Q)((f+P)\cdot W^K)^T \\
&= (f\cdot W^Q + P\cdot W^Q)(f\cdot W^K + P\cdot W^K)^T \\
&= (f\cdot W^Q)(f\cdot W^K)^T + (f\cdot W^Q)(P\cdot W^K)^T + \\
&\quad (P\cdot W^Q)(f\cdot W^K)^T + (P\cdot W^Q)(P\cdot W^K)^T \\
&= (f\cdot W^Q)(f\cdot W^K)^T + (f\cdot W^Q)(P\cdot W^K)^T + \\
&\quad (P\cdot W^Q)(f\cdot W^K)^T + (P\cdot W^Q)(P\cdot W^K)^T \\
&= (Index_{row}(F, I^c)\cdot W^Q)(Index_{row}(F, I^c)\cdot W^K)^T + \\
&\quad (Index_{row}(F, I^c)\cdot W^Q)(P\cdot W^K)^T + \\
&\quad (P\cdot W^Q)(Index_{row}(F, I^c)\cdot W^K)^T + \\
&\quad (P\cdot W^Q)(P\cdot W^K)^T \\
&= (Index_{row}(F\cdot W^Q, I^c))(Index_{row}(F\cdot W^K, I^c))^T + \\
&\quad (Index_{row}(F\cdot W^Q, I^c))(P\cdot W^K)^T + \\
&\quad (P\cdot W^Q)(Index_{row}(F\cdot W^K, I^c))^T + \\
&\quad (P\cdot W^Q)(P\cdot W^K)^T \\
&= Index_{column}(Index_{row}((F\cdot W^Q)(F\cdot W^K)^T, I^c), I^c) + \\
&\quad Index_{row}((F\cdot W^Q)(P\cdot W^K)^T, I^c) + \\
&\quad Index_{column}((P\cdot W^Q)(F\cdot W^K)^T, I^c) + \\
&\quad (P\cdot W^Q)(P\cdot W^K)^T \\
&= Index_{column}(Index_{row}((Meta^Q)(Meta^K)^T, I^c), I^c) + \\
&\quad Index_{row}((Meta^Q)(Pos^K)^T, I^c) + \\
&\quad Index_{column}((Pos^Q)(Meta^K)^T, I^c) + \\
&\quad (Pos^Q)(Pos^K)^T
\end{aligned}
$$

2.2.1 Compute Q, K, V by standard method

[8]:
```
def compute_qkv_by_standard_method(P, IC):
    """P denotes the all position embeddings.
IC is the character index for words."""
    ## character_embedding_layer(IC) means Index_row(F, I^C)
    ## f shape: (500, l, d_emb)
    f = character_embedding_layer(IC)

    ## WQ(f + P) means (f + P) \cdot W^Q
    ## Q shape: (500, l, d_emb)
    Q = WQ(f + P)

    ## WK(f + P) means (f + P) \cdot W^Q
    ## K shape: (500, l, d_emb)
```

```
    K = WK(f + P)

    ## WK(f + P) means (f + P) \cdot W^Q
    ## K shape: (500, l, d_emb)
    V = WV(f + P)

    ## QK^T
    ## QK shape: (500, l, l)
    QK_standard = Q @ K.transpose(-1, -2)
    return V, QK_standard


V, QK_standard = compute_qkv_by_standard_method(P, IC)
```

2.2.2 Compute Q, K, V by index method

The following statements correspond to line 410 to 417 of the manuscript. As an explanation, W_P^Q, W_P^K, W_P^V are equal to W_M^Q, W_M^Q, W_M^V respectively. This can help us to confirm whether computing QK by index method can achive the same result as the standard method. But in fact, for performance considerations, we actually use different weight matrixes to compute posQ, posK, posV.

[9]:
```
def compute_qkv_by_index_method(F, P, IC):
    """F denotes the all embeddings in the character embedding layer.
P denotes the all position embeddings.
IC is the character index for words."""
    ## WQ(F) means F \cdot W_M^Q
    ## metaQ shape: (n, d_emb)
    metaQ = WQ(F)

    ## WK(F) means F \cdot W_M^Q
    ## metaK shape: (n, d_emb)
    metaK = WK(F)

    ## WV(F) means F \cdot W_M^V
    ## metaV shape: (n, d_emb)
    metaV = WV(F)

    ## As an explanation, W_P^Q, W_P^K, W_P^V are equal to W_M^Q, W_M^Q, W_M^V␣
 ↪respectively.
    ## This can help us to confirm whether computing QK by index method can␣
 ↪achive the same result as the standard method.
    ## But in fact, for performance considerations, we actually use different␣
 ↪weight matrixes to compute posQ, posK, posV
    ## WQ(P) means P \cdot W_P^Q
    ## posQ shape: (l, d_emb)
    posQ = WQ(P)
```

```
    ## WK(P) means P \cdot W_P^K
    ## posK shape: (l, d_emb)
    posK = WK(P)

    ## WV(P) means P \cdot W_P^V
    ## posK shape: (l, d_emb)
    posV = WV(P)

    ## Compute V
    V = index(metaV, IC, -2) + posV

    ## Mm = Index_{column}(Index_{row}((Meta^Q)(Meta^K)^T, I^c), I^c)
    ## mm shape: (500, l, l)
    mm = metaQ @ metaK.transpose(-1, -2)
    # mm = mm.unsqueeze(0)
    # mm = torch.cat([mm.index_select(-2, i).index_select(-1, i) for i in IC])
    mm = index(mm, IC, -2)
    column_indexes = torch.einsum(
        "wac, wbc -> wabc",
        torch.stack((IC, torch.ones_like(IC)), -1),
        torch.stack((torch.ones_like(IC), IC), -1))
    mm = mm.gather(-1, column_indexes[:, :, :, 1])

    ## Mp = Index_{row}((Meta^Q)(Pos^K)^T, I^c)
    ## mp shape: (500, l, l)
    mp = metaQ @ posK.transpose(-1, -2)
    mp = index(mp, IC, -2)

    ## Pm = Index_{column}((Pos^Q)(Meta^K)^T, I^c)
    ## pm shape: (500, l, l)
    pm = posQ @ metaK.transpose(-1, -2)
    pm = index(pm, IC, -1).transpose(0, 1)

    ## Pp = (Pos^Q)(Pos^K)^T
    ## pp shape: (1, l, l)
    pp = posQ @ posK.transpose(-1, -2)

    ## QK shape: (500, l, l)
    QK_index = mm + mp + pm + pp
    return V, QK_index


V, QK_index = compute_qkv_by_index_method(F, P, IC)
```

2.2.3 The mean absolute error(MAE) between two method

The MAE is about 6.4×10^{-6}, mean that the results of the two methods are almost equal.

[10]:
```
mae = torch.mean(torch.abs(QK_standard - QK_index))
print(mae.item())
```

```
6.423888862627791e-06
```

2.2.4 Efficiency comparison

[11]:
```
start = time()

for i in range(10000):
    V, QK_standard = compute_qkv_by_standard_method(P, IC)

wall_time_standard = time() - start
print(f"The standard method costs {wall_time_standard:.2f} seconds.")
```

```
The standard method costs 36.69 seconds.
```

[12]:
```
start = time()

for i in range(10000):
    V, QK_index = compute_qkv_by_index_method(F, P, IC)

wall_time_index = time() - start
print(f"The index method costs costs {wall_time_index:.2f} seconds.")
```

```
The index method costs costs 7.28 seconds.
```

2.3 The whole process of ISDPA

[13]:
```
def isdpa(F, P, IC):
    """ISDPA: Index based scale dot-product attention.
F denotes the all embeddings in the character embedding layer.
P denotes the all position embeddings.
IC is the character index for words."""
    ## Compute QK^T
    V, QK_index = compute_qkv_by_index_method(F, P, IC)

    ## The self attention matrix
    score = torch.softmax(QK_index / torch.sqrt(torch.tensor(d_emb).float()), -1)

    ## Calculate the means of Score by columns as mean pooling.
    score = score.mean(-2)

    ## Compute the result
```

```
    result = torch.einsum("wl, wld -> wd", score, V)
    return result
```

[14]:
```
def sdpa(P, IC):
    """SDPA: Scale dot-product attention.
P denotes the all position embeddings.
IC is the character index for words."""
    ## Compute QK^T
    V, QK_standard = compute_qkv_by_standard_method(P, IC)

    ## The self attention matrix
    score = torch.softmax(QK_standard / torch.sqrt(torch.tensor(d_emb).float()),␣
 ↪-1)

    ## Calculate the means of Score by columns as mean pooling.
    score = score.mean(-2)

    ## Compute the result
    result = torch.einsum("wl, wld -> wd", score, V)
    return result
```

2.3.1 The mean absolute error(MAE) between two method

The MAE is about 6.4×10^{-6}, mean that the results of the two methods are almost equal.

[15]:
```
torch.mean(torch.abs(isdpa(F, P, IC) - sdpa(P, IC)))
print(mae.item())
```

```
6.423888862627791e-06
```

2.3.2 Efficiency comparison

[16]:
```
start = time()

for i in range(10000):
    result = sdpa(P, IC)

wall_time_sdpa = time() - start
print(f"The SDPA costs {wall_time_sdpa:.2f} seconds.")
```

```
The SDPA costs 38.63 seconds.
```

[17]:
```
start = time()

for i in range(10000):
    result = isdpa(F, P, IC)
```

```
wall_time_isdpa = time() - start
print(f"The ISDPA costs {wall_time_isdpa:.2f} seconds.")
```

```
The ISDPA costs 8.88 seconds.
```

3 Explanation for "2-D stack for multi-head attentions"(Page 4, line 365-385)

[18]:
```
## Suposed head_1, head_2, ..., head_h have been computed.
## h denotes for the number of heads.
h = 32
heads = [torch.rand([500, d_emb]).to(device) for i in range(h)]
```

3.1 Standard multi-head attention

$$MultiHead(Q,K,V) = Concat(head_1, ..., head_h)W^O$$
$$where\ head_i = SDPA(Q_i, K_i, V_i)$$

[19]:
```
WO_multi_head = torch.nn.Linear(h * d_emb, d_emb, bias=False).to(device)

def multi_head(heads):
    ## After concat, multi_head is a 2-D tensor.
    ## multi_head shape: (500, d_emb * h)
    heads = torch.cat(heads, dim=-1)
    ## multi_head_result shape: (500, d_emb)
    multi_head_result = WO_multi_head(heads)
    return multi_head_result
```

3.2 2-D stack for multi-head attentions

$$MultiHead(Q,K,V) = Tanh(Tanh(Stack(head_1, head_2, \cdots, head_h)W^H)W^O)$$
$$\text{where } head_i = ISDPA(Q_i, K_i, V_i)$$

[20]:
```
## n_hidden: the number of MLP hidden nodes.
n_hidden = 16

## n_output: the number of MLP output nodes
n_output = 1

WH_stack_multi_head = torch.nn.Linear(h, n_hidden, bias=False).to(device)
WO_stack_multi_head = torch.nn.Linear(n_hidden, n_output, bias=False).to(device)

def stack_multi_head(heads):
```

```
    ## After stack, multi_head is a 3-D tensor. For each word, multi_head is a␣
↪2-D tensor.
    ## multi_head shape: (500, d_emb, h)
    heads = torch.stack(heads, dim=-1)
    multi_head_result = torch.tanh(WH_stack_multi_head(heads))
    ## multi_head_result shape: (500, d_emb * n_output)
    multi_head_result = torch.tanh(WO_stack_multi_head(multi_head_result)).
↪reshape([500, d_emb * n_output])
    return multi_head_result
```