TA3 Actividades de Transformer

Índice

1 Introducción

☐ En relación con Transformer, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Es una arquitectura SOTA introducida en al artículo Attention Is All You Need de 2017.
2. Aunque se propuso para traducción automática, pronto se convirtió en la arquitectura de referencia para muchas otras tareas NLP (natural language processing).
3. Su aplicación en tareas de visión ha producido resultados muy competitivos, comparables con los que producen arquitecturas clásicas en visión.
4. Las tres opciones anteriores son correctas.

Solución: La 4

---

2 Arquitectura

☐ Sea un transformer de parámetros `src_vocab_size=3`, `tgt_vocab_size=3`, `embed_dim=2`, `num_layers=1` y `num_heads=1`. El número de parámetros entrenables del encoder es:

1. 24
2. 66
3. 78
4. Ninguno de los anteriores

Solución: La 3

In [ ]:
```
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
```

In [ ]:
```
total_numel = 0
for name, p in model.named_parameters():
    if name[:7] == "encoder" and p.requires_grad:
        total_numel += p.numel()
        print(f"{name}: {p.shape} -> {p.numel()}")
print("Total:", total_numel)
```

```
encoder.layers.0.self_attn.q_proj.weight: torch.Size([2, 2]) -> 4
encoder.layers.0.self_attn.q_proj.bias: torch.Size([2]) -> 2
encoder.layers.0.self_attn.k_proj.weight: torch.Size([2, 2]) -> 4
encoder.layers.0.self_attn.k_proj.bias: torch.Size([2]) -> 2
encoder.layers.0.self_attn.v_proj.weight: torch.Size([2, 2]) -> 4
encoder.layers.0.self_attn.v_proj.bias: torch.Size([2]) -> 2
encoder.layers.0.self_attn.out_proj.weight: torch.Size([2, 2]) -> 4
encoder.layers.0.self_attn.out_proj.bias: torch.Size([2]) -> 2
encoder.layers.0.ff.linear1.weight: torch.Size([8, 2]) -> 16
encoder.layers.0.ff.linear1.bias: torch.Size([8]) -> 8
encoder.layers.0.ff.linear2.weight: torch.Size([2, 8]) -> 16
encoder.layers.0.ff.linear2.bias: torch.Size([2]) -> 2
encoder.layers.0.norm_self_attn.weight: torch.Size([2]) -> 2
encoder.layers.0.norm_self_attn.bias: torch.Size([2]) -> 2
encoder.layers.0.norm_ff.weight: torch.Size([2]) -> 2
encoder.layers.0.norm_ff.bias: torch.Size([2]) -> 2
encoder.norm.weight: torch.Size([2]) -> 2
encoder.norm.bias: torch.Size([2]) -> 2
Total: 78
```

---

3 Traducción automática neuronal

☐ En relación con el entrenamiento de Transformer para traducción automática, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Se lleva a cabo con ejemplos de traducción, esto es, pares de la forma (texto entrada, texto salida). El texto de entrada se tokeniza de acuerdo con un vocabulario de tokens textuales de entrada; análogamente, el de salida se tokeniza según un vocabulario de tokens textuales de salida. Además, se añade un token especial "principio de texto" a la secuencia de salida. Tanto la secuencia de tokens textuales de entrada como la de salida se normalizan en longitud, de acuerdo con una longitud máxima prefijada para ambas, n. En general, las secuencias son de longitud inferior a n, por lo que su normalización supone añadir tokens especiales de padding hasta alcanzar longitud n.
2. La secuencia de n tokens textuales de entrada se transforma en una secuencia de n tokens vectoriales de dimensión prefijada, d_{model}, mediante la aplicación de un embedding de entrada lineal que interpreta cada token textual de entrada como un vector onehot de dimensión igual al vocabulario de tokens textuales de entrada. La secuencia de n tokens textuales de salida se transforma análogamente, con base en un embedding de salida lineal.
3. Transformer aprende a predecir el token vectorial $(i+1)$-ésimo de la salida a partir de la secuencia de n tokens vectoriales de entrada y la subsecuencia de i primeros tokens vectoriales de salida, $i = 1, \ldots, n-1$. Aquí debe tenerse en cuenta el token especial "principio de texto" añadido al inicio de la secuencia de tokens textuales de salida, por lo que la traducción efectiva comienza con la predicción del segundo token vectorial de salida. Cada token vectorial de salida predicho puede interpretarse como un modelo probabístico (categórico) sobre el vocabulario de tokens textuales de salida, tras aplicar la inversa de la transformación lineal del embedding de salida y una softmax.
4. Las tres opciones anteriores son correctas.

Solución: La 4

☐ En relación con la inferencia en Transformer para traducción automática, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Se lleva a cabo a partir del texto de entrada a traducir, cuya tokenización y embedding resulta en una secuencia de n tokens vectoriales de dimensión d_{model}. El texto de salida se inicializa con un token especial "principio de texto", cuyo embedding da lugar a un primer token vectorial de salida, también de de dimensión d_{model}.
2. Transformer predice el token vectorial $(i+1)$-ésimo de la salida a partir de la secuencia de n tokens vectoriales de entrada y la subsecuencia de i primeros tokens vectoriales de salida ya predichos, $i = 1, \ldots, n-1$. La predicción del segundo token vectorial de la salida se basa en el embedding del token "principio de texto" con el que se inicializa la salida. La softmax del unembedding de dicha predicción constituye una distribución de probabilidad sobre el vocabulario de tokens textuales de salida. Con base en esta distribución, se escoge un token textual de salida, por ejemplo el de mayor probabilidad (greedy decoding), y su embedding se añade a la secuencia de tokens vectoriales de salida ya predichos. El proceso de inferencia procede de forma análoga con la predicción del tercer token vectorial de la salida y sucesivos.
3. Greedy decoding es un técnica popular y sencilla de inferencia, pero no la única. De hecho, el proceso de inferencia puede verse como un problema de búsqueda en un espacio de secuencias de tokens textuales de salida, por lo que también se aplican técnicas de búsqueda clásicas como beam search.
4. Las tres opciones anteriores son correctas.

Solución: La 4

---

4 Codificación posicional

Ejercicio: añade codificación posicional a $X = \begin{pmatrix} 0.3191 & -0.8395 \\ 0.3191 & -0.8395 \\ 0.0293 & 0.8776 \end{pmatrix}$

Solución:

$$\mathrm{PE}_{(0,0)} = \sin\left(\frac{0}{10000^{0/2}}\right) = 0 \qquad \mathrm{PE}_{(0,1)} = \cos\left(\frac{0}{10000^{0/2}}\right) = 1$$
$$\mathrm{PE}_{(1,0)} = \sin\left(\frac{1}{10000^{0/2}}\right) = 0.8415 \qquad \mathrm{PE}_{(1,1)} = \cos\left(\frac{1}{10000^{0/2}}\right) = 0.5403$$
$$\mathrm{PE}_{(2,0)} = \sin\left(\frac{2}{10000^{0/2}}\right) = 0.9093 \qquad \mathrm{PE}_{(2,1)} = \cos\left(\frac{2}{10000^{0/2}}\right) = -0.4161$$

$$\mathrm{PE}(X) = \begin{pmatrix} 0.3191 & -0.8395 \\ 0.3191 & -0.8395 \\ 0.0293 & 0.8776 \end{pmatrix} \sqrt{2} + \begin{pmatrix} 0 & 1 \\ 0.8415 & 0.5403 \\ 0.9093 & -0.4161 \end{pmatrix} = \begin{pmatrix} 0.4513 & -0.1872 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$$

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model, create_fixed_positional_encoding
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
```

In [ ]:
```
src = torch.LongTensor([[2, 2, 1]]); model.src_embed.embed(src).data
```

Out[ ]:
```
tensor([[[ 0.3191, -0.8395],
         [ 0.3191, -0.8395],
         [ 0.0293,  0.8776]]])
```

In [ ]:
```
pe = create_fixed_positional_encoding(dim=2, max_len=3); pe
```

Out[ ]:
```
tensor([[[ 0.0000,  1.0000],
         [ 0.8415,  0.5403],
         [ 0.9093, -0.4161]]])
```

In [ ]:
```
model.src_embed(src).data
```

Out[ ]:
```
tensor([[[ 0.4513, -0.1872],
         [ 1.2927, -0.6469],
         [ 0.9508,  0.8249]]])
```

5 Dropout

Ejercicio: aplica dropout a $X = \begin{pmatrix} 0.4513 & -0.1872 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$ en los siguientes casos

1. $p = 0.1,$ asumiendo que no anula ninguna salida
2. $p = 0.5,$ asumiendo que anula las salidas de la primera columna y última fila

Solución:

$$\begin{pmatrix} 0.4513 & -0.1872 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix} \frac{1}{0.9} = \begin{pmatrix} 0.5014 & -0.2080 \\ 1.4364 & -0.7188 \\ 1.0564 & 0.9166 \end{pmatrix} \qquad \begin{pmatrix} 0 & -0.1872 \\ 0 & -0.6469 \\ 0 & 0 \end{pmatrix} \frac{1}{0.5} = \begin{pmatrix} 0 & -0.3744 \\ 0 & -1.2938 \\ 0 & 0 \end{pmatrix}$$

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.1)
src = torch.LongTensor([[2, 2, 1]]); model.src_embed(src).data
```

Out[ ]:
```
tensor([[[ 0.5014, -0.2080],
         [ 1.4364, -0.7188],
         [ 1.0564,  0.9166]]])
```

In [ ]:
```
import torch; import torch.nn as nn; import import_ipynb; import at251; torch.manual_seed(23);
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.5)
src = torch.LongTensor([[2, 2, 1]]); model.src_embed(src).data
```

Out[ ]:
```
tensor([[[ 0.0000, -0.3744],
         [ 0.0000, -1.2938],
         [ 0.0000,  0.0000]]])
```

6 LayerNorm

Ejercicio: aplica $\mathrm{LayerNorm}$ a $X = \begin{pmatrix} 0.4513 & -0.1872 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$

Solución:

$$\begin{bmatrix} \hat{\mu} = 0.1320 & \hat{\sigma} = 0.4515 \\ \hat{\mu} = 0.3229 & \hat{\sigma} = 1.3715 \\ \hat{\mu} = 0.8879 & \hat{\sigma} = 0.0890 \end{bmatrix} \quad \rightarrow \quad \mathrm{LayerNorm}(X) = \begin{pmatrix} 0.7071 & -0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix}$$

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
```

In [ ]:
```
src = torch.LongTensor([[2, 2, 1]]); src_embedded = model.src_embed(src).data; src_embedded.data
```

Out[ ]:
```
tensor([[[ 0.4513, -0.1872],
         [ 1.2927, -0.6469],
         [ 0.9508,  0.8249]]])
```

In [ ]:
```
mean = src_embedded.mean(-1, keepdim=True); print(str(mean).replace('\n', ''))
std = src_embedded.std(-1, keepdim=True); print(str(std).replace('\n', ''))
```

```
tensor([[[0.1320],         [0.3229],         [0.8878]]])
tensor([[[0.4515],         [1.3715],         [0.0890]]])
```

In [ ]:
```
model.encoder.layers[0].norm_self_attn(src_embedded).data
```

Out[ ]:
```
tensor([[[ 0.7071, -0.7071],
         [ 0.7071, -0.7071],
         [ 0.7070, -0.7070]]])
```

7 Atención producto-escalar escalada

Ejercicio: halla $\text{Attention}(Q,K,V)$ con $Q=K=V=\begin{pmatrix}0.7071 & -0.7071\\ 0.7071 & -0.7071\\ 0.7070 & -0.7070\end{pmatrix}$

Solución:

$$\text{AttentionScores}(Q,K)=\begin{pmatrix}0.7071 & -0.7071\\ 0.7071 & -0.7071\\ 0.7070 & -0.7070\end{pmatrix}\begin{pmatrix}0.7071 & 0.7071 & 0.7070\\ -0.7071 & -0.7071 & -0.7070\end{pmatrix}=\mathbf{1}_{3\times 3}$$

$$\text{ScaledAttentionScores}(Q,K)=\frac{1}{\sqrt{2}}\,\mathbf{1}_{3\times 3}$$

$$\text{AttentionWeights}(Q,K)=\frac{1}{3}\,\mathbf{1}_{3\times 3}$$

$$\text{Attention}(Q,K,V)=\frac{1}{3}\,\mathbf{1}_{3\times 3}\begin{pmatrix}0.7071 & -0.7071\\ 0.7071 & -0.7071\\ 0.7070 & -0.7070\end{pmatrix}=\begin{pmatrix}0.7071 & -0.7071\\ 0.7071 & -0.7071\\ 0.7071 & -0.7071\end{pmatrix}$$

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
src = torch.LongTensor([[2, 2, 1]]); x = model.src_embed(src).data
norm_x = model.encoder.layers[0].norm_self_attn(x).data; norm_x
```

Out[ ]:
```
tensor([[[ 0.7071, -0.7071],
         [ 0.7071, -0.7071],
         [ 0.7070, -0.7070]]])
```

In [ ]:
```
query = key = value = norm_x; d_k = query.size(-1)
scores = torch.matmul(query, key.transpose(-2, -1)); scores
```

Out[ ]:
```
tensor([[[1.0000, 1.0000, 0.9999],
         [1.0000, 1.0000, 0.9999],
         [0.9999, 0.9999, 0.9998]]])
```

In [ ]:
```
import math; scaled_scores = scores / math.sqrt(d_k); scaled_scores
```

Out[ ]:
```
tensor([[[0.7071, 0.7071, 0.7070],
         [0.7071, 0.7071, 0.7070],
         [0.7070, 0.7070, 0.7069]]])
```

In [ ]:
```
p_attn = scores.softmax(dim=-1); p_attn
```

Out[ ]:
```
tensor([[[0.3333, 0.3333, 0.3333],
         [0.3333, 0.3333, 0.3333],
         [0.3333, 0.3333, 0.3333]]])
```

In [ ]:
```
self_attn = torch.matmul(p_attn, value); self_attn
```

Out[ ]:
```
tensor([[[ 0.7071, -0.7071],
         [ 0.7071, -0.7071],
         [ 0.7071, -0.7071]]])
```

8 Atención multicabeza

Ejercicio: halla $\mathrm{MultiHead}(Q,K,V)$ con $h=1$, $Q=K=V=\begin{pmatrix}0.7071 & 0.7071\\ 0.7071 & -0.7071\\ 0.7070 & -0.7070\end{pmatrix}$ y

$$W_1^Q=\begin{pmatrix}0.8635 & 0.7223\\ 0.5531 & 0.3659\end{pmatrix}^t \qquad B_1^Q=\mathbf{1}_n(0.6123,-0.2899)$$
$$W_1^K=\begin{pmatrix}-0.0060 & -0.5075\\ -0.0329 & 0.8903\end{pmatrix}^t \qquad B_1^K=\mathbf{1}_n(0.2253,-0.4414)$$
$$W_1^V=\begin{pmatrix}0.4922 & -0.3579\\ -0.5233 & 0.0872\end{pmatrix}^t \qquad B_1^V=\mathbf{1}_n(0.0727,-0.5929)$$
$$W^O=\begin{pmatrix}1.2168 & -0.1905\\ -0.0890 & -0.5564\end{pmatrix}^t \qquad B^O=\mathbf{1}_n(-0.5157,-0.1097)$$

Solución:

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
src = torch.LongTensor([[2, 2, 1]]); x = model.src_embed(src).data
norm_x = model.encoder.layers[0].norm_self_attn(x).data; query = key = value = norm_x; norm_x
```

Out[ ]:
```
tensor([[[ 0.7071, -0.7071],
         [ 0.7071, -0.7071],
         [ 0.7070, -0.7070]]])
```

In [ ]:
```
self_attn = model.encoder.layers[0].self_attn
bsz, seq_len, embed_dim = query.size(); num_heads = 1; head_dim = embed_dim // num_heads
print(f"bsz={bsz}; seq_len={seq_len}; embed_dim={embed_dim}; num_heads={num_heads}; head_dim={head_dim}")
```

```
bsz=1; seq_len=3; embed_dim=2; num_heads=1; head_dim=2
```

In [ ]:
```
print("W_Q: weight", str(self_attn.q_proj.weight.data).replace('\n',''), " bias", self_attn.q_proj.bias.data)
q = self_attn.q_proj(query).view(bsz, -1, num_heads, head_dim).transpose(1, 2); q.data
```

Out[ ]:
```
W_Q: weight tensor([[0.8635, 0.7223],        [0.5531, 0.3659]])  bias tensor([ 0.6123, -0.2899])
tensor([[[[ 0.7122, -0.1575],
          [ 0.7122, -0.1575],
          [ 0.7122, -0.1575]]]])
```

$$QW_1^Q+B_1^Q=\begin{pmatrix}0.7122 & -0.1575\\ 0.7122 & -0.1575\\ 0.7122 & -0.1575\end{pmatrix}$$

In [ ]:
```
print("W_K: weight", str(self_attn.k_proj.weight.data).replace('\n',''), " bias", self_attn.k_proj.bias.data)
k = self_attn.k_proj(key).view(bsz, -1, num_heads, head_dim).transpose(1, 2); k.data
```

Out[ ]:
```
W_K: weight tensor([[-0.0060, -0.5075],        [-0.0329,  0.8903]])  bias tensor([ 0.2253, -0.4414])
tensor([[[[ 0.5799, -1.0942],
          [ 0.5799, -1.0942],
          [ 0.5798, -1.0941]]]])
```

$$KW_1^K+B_1^K=\begin{pmatrix}0.5799 & -1.0942\\ 0.5799 & -1.0942\\ 0.5798 & -1.0941\end{pmatrix}$$

In [ ]:
```
scores = (q @ k.transpose(-2, -1)) * head_dim**-0.5; scores.data
```

Out[ ]:
```
tensor([[[[0.4138, 0.4138, 0.4138],
          [0.4138, 0.4138, 0.4138],
          [0.4138, 0.4138, 0.4138]]]])
```

$$\text{ScaledAttentionScores}(QW_1^Q + B_1^Q, KW_1^K + B_1^K) = 0.4138 \cdot \mathbf{1}_{3\times 3}$$

In [ ]:
```
attn = nn.functional.softmax(scores, dim=-1); attn.data
```

Out[ ]:
```
tensor([[[[0.3333, 0.3333, 0.3333],
          [0.3333, 0.3333, 0.3333],
          [0.3333, 0.3333, 0.3333]]]])
```

$$\text{AttentionWeights}(QW_1^Q + B_1^Q, KW_1^K + B_1^K) = \frac{1}{3} \cdot \mathbf{1}_{3\times 3}$$

In [ ]:
```
print("W_V: weight", str(self_attn.v_proj.weight.data).replace('\n',''), " bias", self_attn.v_proj.bias.data)
v = self_attn.v_proj(value).view(bsz, -1, num_heads, head_dim).transpose(1, 2); v.data
```

Out[ ]:
```
W_V: weight tensor([[ 0.4922, -0.3579],        [-0.5233,  0.0872]])  bias tensor([ 0.0727, -0.5929])
tensor([[[[ 0.6738, -1.0246],
          [ 0.6738, -1.0246],
          [ 0.6737, -1.0246]]]])
```

$$VW_1^V + B_1^V = \begin{pmatrix} 0.6738 & -1.0246 \\ 0.6738 & -1.0246 \\ 0.6737 & -1.0246 \end{pmatrix}$$

In [ ]:
```
values = attn @ v; values = values.transpose(1, 2).reshape(bsz, seq_len, embed_dim); values.data
```

Out[ ]:
```
tensor([[[ 0.6737, -1.0246],
         [ 0.6737, -1.0246],
         [ 0.6737, -1.0246]]])
```

$$\text{head}_1 = \text{Attention}(QW_1^Q + B_1^Q, KW_1^K + B_1^K, VW_1^V + B_1^V) = \begin{pmatrix} 0.6737 & -1.0246 \\ 0.6737 & -1.0246 \\ 0.6737 & -1.0246 \end{pmatrix}$$

In [ ]:
```
print("W_O: weight", str(self_attn.out_proj.weight.data).replace('\n',''),
      " bias", self_attn.out_proj.bias.data)
out = self_attn.out_proj(values).view(bsz, -1, num_heads, head_dim).transpose(1, 2); out.data
```

Out[ ]:
```
W_O: weight tensor([[ 1.2168, -0.1905],        [-0.0890, -0.5564]])  bias tensor([-0.5157, -0.1097])
tensor([[[[0.4993, 0.4004],
          [0.4993, 0.4004],
          [0.4993, 0.4004]]]])
```

$$\text{MultiHead}(Q, K, V) = \text{head}_1 W^O + B^O = \begin{pmatrix} 0.4993 & 0.4004 \\ 0.4993 & 0.4004 \\ 0.4993 & 0.4004 \end{pmatrix}$$

In [ ]:
```
str(self_attn(norm_x, norm_x, norm_x).data).replace('\n','')
```

Out[ ]:
```
'tensor([[[0.4993, 0.4004],         [0.4993, 0.4004],         [0.4993, 0.4004]]])'
```

9 Conexiones residuales

☐ Transformer incluye conexiones residuales en encoder y decoder para facilitar el entrenamiento de modelos muy profundos. Concretamente, el número de capas de conexión residual en una capa encoder es:

1. Una
2. Dos
3. Tres
4. Más de tres

Solución: La 2.

10 Position-wise Feed-Forward Networks

Ejercicio: halla $\mathrm{PWFFN}(X)_{1,:}$ con $X = \begin{pmatrix} 0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix}$ y

$$W_1 = \begin{pmatrix} 0.4008 & 0.1917 \\ -0.4451 & -0.6482 \\ 0.7679 & 0.5881 \\ -0.7363 & -0.6416 \\ 0.2594 & 0.4606 \\ 0.4195 & -0.2898 \\ 0.2920 & 0.0965 \\ -0.0160 & 0.0162 \end{pmatrix}^t$$

$$\boldsymbol{b}_1^t = (0.0312, 0.2093, 0.2466, -0.5398, -0.3994, 0.3540, 0.4932, -0.2173)$$

$$W_2 = \begin{pmatrix} 0.5994 & -0.2837 & -0.2077 & -0.5024 & -0.5487 & 0.7268 & 0.6768 & -0.6624 \\ -0.4707 & 0.2907 & 0.2848 & 0.4173 & 0.4015 & 0.4828 & 0.1108 & 0.1021 \end{pmatrix}^t$$

$$\boldsymbol{b}_2^t = (0.0928, -0.2395)$$

Solución:

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model
model = create_model(src_vocab_size=3, tgt_vocab_size=3, embed_dim=2, num_layers=1, num_heads=1, dropout=0.)
src = torch.LongTensor([[2, 2, 1]]); x = model.src_embed(src).data
norm_x = model.encoder.layers[0].norm_self_attn(x).data
x = x + model.encoder.layers[0].self_attn(norm_x, norm_x, norm_x).data
norm_x = model.encoder.layers[0].norm_ff(x); norm_x.data
```

Out[ ]:
```
tensor([[[ 0.7071, -0.7071],
         [ 0.7071, -0.7071],
         [ 0.7071, -0.7071]]])
```

In [ ]:
```
ff = model.encoder.layers[0].ff
print("linear1 weight", ff.linear1.weight.data)
print("linear1 bias", ff.linear1.bias.data)
print("linear2 weight", ff.linear2.weight.data)
print("linear2 bias", ff.linear2.bias.data)
```

```
linear1 weight tensor([[ 0.4008,  0.1917],
        [-0.4451, -0.6482],
        [ 0.7679,  0.5881],
        [-0.7363, -0.6416],
        [ 0.2594,  0.4606],
        [ 0.4195, -0.2898],
        [ 0.2920,  0.0965],
        [-0.0160,  0.0162]])
linear1 bias tensor([ 0.0312,  0.2093,  0.2466, -0.5398, -0.3994,  0.3540,  0.4932, -0.2173])
linear2 weight tensor([[ 0.5994, -0.2837, -0.2077, -0.5024, -0.5487,  0.7268,  0.6768, -0.6624],
        [-0.4707,  0.2907,  0.2848,  0.4173,  0.4015,  0.4828,  0.1108,  0.1021]])
linear2 bias tensor([ 0.0928, -0.2395])
```

In [ ]:
```
norm_x0 = norm_x[0, 0].data; print(norm_x0)
linear1_norm_x0 = ff.linear1(norm_x0).data; print(linear1_norm_x0)
relu_norm_x0 = nn.functional.relu(linear1_norm_x0); print(relu_norm_x0)
linear2_norm_x0 = ff.linear2(relu_norm_x0).data; print(linear2_norm_x0)
```

```
tensor([ 0.7071, -0.7071])
tensor([ 0.1790,  0.3529,  0.3736, -0.6069, -0.5416,  0.8555,  0.6314, -0.2401])
tensor([0.1790, 0.3529, 0.3736, 0.0000, 0.0000, 0.8555, 0.6314, 0.0000])
tensor([1.0716, 0.3682])
```

$$
\begin{aligned}
\mathrm{PWFFN}(X)_{1,:} &= \mathrm{ReLU}((0.7071, 0.7071)W_1 + \boldsymbol{b}_1^t)W_2 + \boldsymbol{b}_2^t \\
&= \mathrm{ReLU}(0.1790, 0.3529, 0.3736, -0.6069, -0.5416, 0.8555, 0.6314, -0.2401)W_2 + \boldsymbol{b}_2^t \\
&= (0.1790, 0.3529, 0.3736, 0.0000, 0.0000, 0.8555, 0.6314, 0.0000)W_2 + \boldsymbol{b}_2^t \\
&= (1.0716, 0.3682)
\end{aligned}
$$

In [ ]:
```
linear1_norm_x = ff.linear1(norm_x).data; # print(linear1_norm_x)
relu_norm_x = nn.functional.relu(linear1_norm_x); # print(relu_norm_x)
linear2_norm_x = ff.linear2(relu_norm_x).data; print(linear2_norm_x)
```

```
tensor([[[1.0716, 0.3682],
         [1.0716, 0.3682],
         [1.0716, 0.3682]]])
```

In [ ]:
```
ff(norm_x).data
```

Out[ ]:
```
tensor([[[1.0716, 0.3682],
         [1.0716, 0.3682],
         [1.0716, 0.3682]]])
```

11 Encoder

☐ En relación con el encoder (pre-norm) de Transformer, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Dada una secuencia de entrada $X \in \mathbb{R}^{n \times d_{\text{model}}}$, el encoder aplica $N = 6$ capas encoder y termina con una $\mathrm{LayerNorm}$.
2. Cada capa encoder incluye, entre otras, una capa de auto-atención y otra FFN (Position-wise Feed-Forward Network).
3. Cada capa encoder incluye, entre otras, dos capas $\mathrm{LayerNorm}$ y dos Add (conexiones residuales).
4. Las tres opciones anteriores son correctas.

Solución: La 4.

12 Decoder

☐ En relación con el decoder (pre-norm) de Transformer, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Dada una entrada (al decoder) X, salida $Y = Y^{(1)}$ y máscara causal M, el encoder aplica $N = 6$ capas decoder y termina con una $\mathrm{LayerNorm}$.
2. Cada capa decoder incluye, entre otras, una capa de auto-atención, otra de atención cruzada y una FFN (Position-wise Feed-Forward Network).
3. Cada capa decoder incluye, entre otras, tres capas $\mathrm{LayerNorm}$ y tres Add (conexiones residuales).
4. Las tres opciones anteriores son correctas.

Solución: La 4.

13 Transformer

☐ En relación con Transformer, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Su instanciación inicializa embeddings, encoder, decoder y cabeza de `weight` ligado al del embedding de la salida.
2. Transformer implementa $\mathrm{Transformer}(X, Y, M) = \mathrm{Decoder}(\mathrm{Encoder}(X), Y, M)$, donde X es una secuencia de entrada, Y una secuencia de salida, y M su máscara causal asociada.
3. En entrenamiento se minimiza la entropía cruzada de la predicción relativa a la salida con label smoothing. En inferencia se produce la salida auto-regresivamente, esto es, cada vez que se infiere un nuevo token textual de salida, su embedding se añade a la secuencia de tokens vectoriales de salida ya predichos con el fin de predecir un nuevo token textual de salida.
4. Las tres opciones anteriores son correctas.

Solución: La 4.

14 Entrenamiento

☐ Annotated Transformer 2025 incluye algunos refinamientos respecto a la propuesta original de Transformer. En relación con estos refinamientos, indica la opción incorrecta o la última opción si las tres primeras son correctas:

1. Se propone el uso del optimizador AdamW en lugar de Adam.
2. El planificador original depende del tamaño del modelo, lo que dificulta comparar modelos; en lugar del mismo, se emplea un planificador parecido, actualmente popular, cosine Annealing (with Warmup).
3. En la propuesta original, $\mathrm{LayerNorm}$ se aplica tras las conexiones residuales (post-norm). En lugar de ello, se propone aplicarlo primero (pre-norm), esto es, antes de la atención o la FFN (ver arXiv:2002.04745v2 para más detalles).
4. Las tres opciones anteriores son correctas.

Solución: La 4.