T3 Transformer

Índice

1 Introducción

Transformer (vanila): arquitectura SOTA introducida en al artículo Attention Is All You Need de 2017

Vídeos introductorios de 3blue1brown:

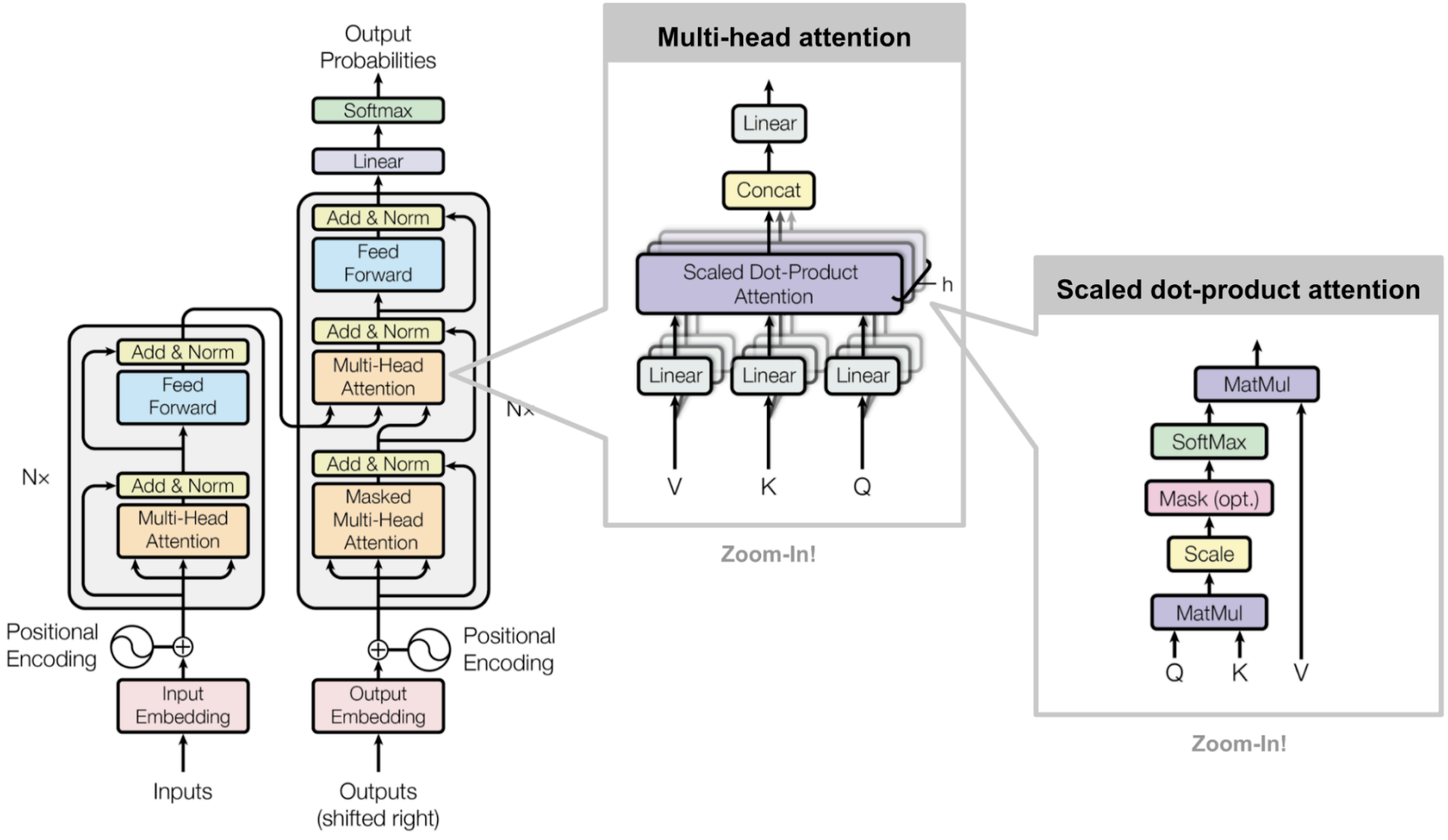
- Transformers, the tech behind LLMs
- Attention in transformers, step-by-step
- How might LLMs store facts

Annotated Transformer 2025 (AT25): implementación diseñada como recurso educativo para introducir Transformer

1. Part 1: Model Architecture
2. Part 2: Preparation for Training
3. Part 3: Toy Training Example
4. Part 4: A Real World Example
5. Part 5: Attention Visualization

Objetivo: introducir la arquitectura Transformer con base en AT25, principalmente la primera parte

2 Arquitectura



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.)
print(str(model)[:1350]+"...")
```

```
Transformer(
  (src_embed): EmbeddingsWithPositionalEncoding(
    (embed): Embedding(3, 2)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (tgt_embed): EmbeddingsWithPositionalEncoding(
    (embed): Embedding(3, 2)
    (dropout): Dropout(p=0.0, inplace=False)
  )
  (encoder): Encoder(
    (layers): ModuleList(
      (0): EncoderLayer(
        (self_attn): MultiHeadAttention(
          (q_proj): Linear(in_features=2, out_features=2, bias=True)
          (k_proj): Linear(in_features=2, out_features=2, bias=True)
          (v_proj): Linear(in_features=2, out_features=2, bias=True)
          (out_proj): Linear(in_features=2, out_features=2, bias=True)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (ff): FeedForward(
          (linear1): Linear(in_features=2, out_features=8, bias=True)
          (linear2): Linear(in_features=8, out_features=2, bias=True)
          (dropout): Dropout(p=0.0, inplace=False)
        )
        (norm_self_attn): LayerNorm()
        (norm_ff): LayerNorm()
        (dropout): Dropout(p=0.0, inplace=False)
      )
    )
    (norm): LayerNorm()
  )
  (decoder): Decoder(
    (layers): ModuleList(
      (0): DecoderLayer(
        (self_attn): MultiHeadAttention(
          (q_proj): Linear(in_features=2, out_features=2, bias=True)
          (k_proj): Linear(in_features=2, out_features=2, b...
```

3 Traducción automática neuronal

Neural machine translation (NMT): aplicación original de Transformer

AT25: AT25 se basa en PyTorch; fijamos la semilla para generación de números aleatorios

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import create_model, create_causal_mask
```

Modelo: modelo aleatorio simple

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

Embedding del texto de entrada: secuencia de vectores de `embed_dim` dimensiones

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

Out[ ]:
```
tensor([[[ 0.0415,  2.2411],
         [ 1.2927, -0.6469],
         [ 0.9508,  0.8249]]])
```

Embedding del texto de salida: secuencia de vectores de `embed_dim` dimensiones

In [ ]:
```
tgt = torch.LongTensor([[0]]); model.tgt_embed(tgt).data
```

Out[ ]:
```
tensor([[[1.3548, 1.4321]]])
```

Proyección lineal final: `generator` transforma la salida del decoder en la predicción probabilística del modelo

In [ ]:
```
model.generator.final_proj.weight = nn.Parameter(torch.randn_like(model.generator.final_proj.weight))
model.generator.final_proj(torch.tensor([[[-0.7071,  0.7071]]])).data
```

Out[ ]:
```
tensor([[[-0.0080, -0.3012, -0.0413]]])
```

Evaluación:

In [ ]:
```
model.eval()
src = torch.LongTensor([[1, 2, 1]]); print(f"Texto de entrada: {src}")
src_mask = torch.ones(1, 1, 3); print(f"Máscara de entrada: {src_mask}\n")
memory = model.encode(src, src_mask)
ys = torch.zeros(1, 1).type_as(src)     # note: ys[0]=0, i.e., ys starts with 0
for i in range(2):
    print(f"Inferencia del siguiente token tras haber predicho {i} tokens:")
    print(f"* Texto de salida ya predicho: {ys}")
    tgt_mask = create_causal_mask(ys.size(1)).type_as(src.data)
    print(f"* Máscara de salida: {str(tgt_mask).replace('\n','')}")
    out = model.decode(ys, memory, src_mask, tgt_mask)
    print(f"* Salida del decoder: {str(out.data).replace('\n','')}")
    prob = model.generator(out[:, -1])     # last token
    print(f"* Logsoftmax del unembedding del último token de la salida: {str(prob.data).replace('\n','')}")
    _, next_word = torch.max(prob, dim=1)
    next_word = next_word.data[0]
    ys = torch.cat([ys, torch.empty(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    print("* Texto de salida con nuevo token inferido:", ys, "\n")
```

```
Texto de entrada: tensor([[1, 2, 1]])
Máscara de entrada: tensor([[[1., 1., 1.]]])

Inferencia del siguiente token tras haber predicho 0 tokens:
* Texto de salida ya predicho: tensor([[0]])
* Máscara de salida: tensor([[[1]]])
* Salida del decoder: tensor([[[-0.7071,  0.7071]]])
* Logsoftmax del unembedding del último token de la salida: tensor([[-0.9981, -1.2913, -1.0314]])
* Texto de salida con nuevo token inferido: tensor([[0, 0]])

Inferencia del siguiente token tras haber predicho 1 tokens:
* Texto de salida ya predicho: tensor([[0, 0]])
* Máscara de salida: tensor([[[1, 0],         [1, 1]]])
* Salida del decoder: tensor([[[-0.7071,  0.7071],         [ 0.7071, -0.7071]]])
* Logsoftmax del unembedding del último token de la salida: tensor([[-1.2162, -0.9231, -1.1830]])
* Texto de salida con nuevo token inferido: tensor([[0, 0, 1]])
```

4 Codificación posicional

Positional encoding: dado un embedding del texto de entrada $X \in \mathbb{R}^{n \times d_{\text{model}}}$, multiplica X por $\sqrt{d_{\text{model}}}$ y añade una secuencia de posiciones absolutas representadas mediante senos y cosenos

$$\text{PE}(X) = X\sqrt{d_{\text{model}}} + \begin{pmatrix} \text{PE}_{(0,0)} & \cdots & \text{PE}_{(0,d_{\text{model}}-1)} \\ \vdots & \vdots & \vdots \\ \text{PE}_{(n-1,0)} & \cdots & \text{PE}_{(n-1,d_{\text{model}}-1)} \end{pmatrix}$$

donde

$$\text{PE}_{(\text{pos},2i)} = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right)$$
$$\text{PE}_{(\text{pos},2i+1)} = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right)$$

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

Solución:

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

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

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

$$\mathrm{PE}(X) = \begin{pmatrix} 0.0293 & 0.8776 \\ 0.3191 & -0.8395 \\ 0.0293 & 0.8776 \end{pmatrix} \sqrt{2} + \begin{pmatrix} 0 & 1 \\ 0.8415 & 0.5403 \\ 0.9093 & -0.4161 \end{pmatrix} = \begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$$

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([[1, 2, 1]]); model.src_embed.embed(src).data
```

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

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

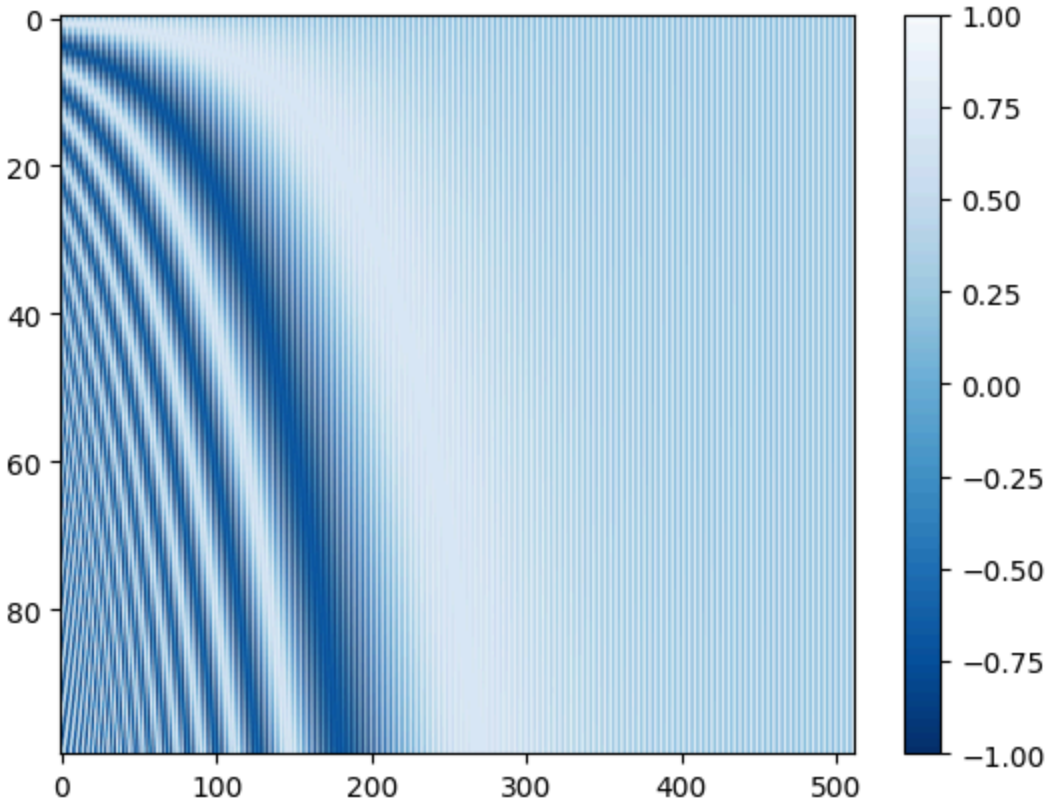
Out[ ]:
```
tensor([[[ 0.0000,  1.0000],
         [ 0.8415,  0.5403],
         [ 0.9093, -0.4161]]])
```

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

Out[ ]:
```
tensor([[[ 0.0415,  2.2411],
         [ 1.2927, -0.6469],
         [ 0.9508,  0.8249]]])
```

Continuidad de la codificación posicional: veamos el caso $n = 100$ y `d_model = 512`

In [ ]:
```
import matplotlib.pyplot as plt
pe = create_fixed_positional_encoding(512, 100)
plt.imshow(*pe, aspect='auto', origin='upper', cmap='Blues_r'); plt.colorbar();
```



5 Dropout

Dropout: capa de regularización de las salidas de otra capa

- Entrenamiento: anula cada salida con una probabilidad dada p; las salidas no anuladas se re-escalan con $\dfrac{1}{1-p}$
- Evaluación: actúa como la función identidad

Fuente: Improving neural networks by preventing co-adaptation of feature detectors

Ejercicio: aplica dropout a $X = \begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$ 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:

$$\begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix} \frac{1}{0.9} = \begin{pmatrix} 0.0461 & 2.4901 \\ 1.4364 & -0.7188 \\ 1.0564 & 0.9166 \end{pmatrix} \qquad \begin{pmatrix} 0 & 2.2411 \\ 0 & -0.6469 \\ 0 & 0 \end{pmatrix} \frac{1}{0.5} = \begin{pmatrix} 0 & 4.4821 \\ 0 & -1.2938 \\ 0 & 0 \end{pmatrix}$$

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([[1, 2, 1]]); model.src_embed(src).data
```

Out[ ]:
```
tensor([[[ 0.0461,  2.4901],
         [ 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([[1, 2, 1]]); model.src_embed(src).data
```

Out[ ]:
```
tensor([[[ 0.0000,  4.4821],
         [ 0.0000, -1.2938],
         [ 0.0000,  0.0000]]])
```

6 LayerNorm

LayerNorm: capa que estandariza cada dato en su última dimensión (si no se indican otras dimensiones)

Ejemplo: $$X = \begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$$

$$\begin{bmatrix} \hat{\mu} = 1.1413 & \hat{\sigma} = 1.5553 \\ \hat{\mu} = 0.3229 & \hat{\sigma} = 1.3715 \\ \hat{\mu} = 0.8879 & \hat{\sigma} = 0.0890 \end{bmatrix} \quad \rightarrow \quad \text{LayerNorm}(X) = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix}$$

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

Out[ ]:
```
tensor([[[ 0.0415,  2.2411],
         [ 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([[[1.1413],         [0.3229],         [0.8878]]])
tensor([[[1.5553],         [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

Queries, Keys y Values: $Q, K \in \mathbb{R}^{n \times d_\text{k}} \quad V \in \mathbb{R}^{n \times d_\text{v}}$

- Cada query $\boldsymbol{q}_i = Q_{i,:}$ es un token de valor a predecir
- Cada par key-value $(\boldsymbol{k}_i, \boldsymbol{v}_i) = (K_{i,:}, V_{i,:})$ es una referencia token-valor
- Asumimos que tokens similares aportan valores informativamente relevantes

Scores de atención: usamos $\boldsymbol{q}_i K^t \in \mathbb{R}^{1 \times n}$ para medir la **atención** que $\boldsymbol{q}_i$ debe prestar a cada key

$$\text{AttentionScores}(Q, K) = QK^t \in \mathbb{R}^{n \times n}$$

Propiedad: si $\boldsymbol{q}_i^t$ y $\boldsymbol{k}_j^t$ son $\mathcal{N}_{d_\text{k}}(\mathbf{0}, \mathbf{I})$, entonces $\boldsymbol{q}_i \boldsymbol{k}_j^t$ sigue una distribución de media nula y varianza d_k

Scores de atención escalados: $\text{ScaledAttentionScores}(Q, K) = \frac{1}{\sqrt{d_\text{k}}} \text{AttentionScores}(Q, K) \in \mathbb{R}^{n \times n}$

Pesos de atención: $\text{AttentionWeights}(Q, K) = \text{Softmax}(\text{ScaledAttentionScores}(Q, K)) \in \mathbb{R}^{n \times n}$

Atención producto-escalar escalada (ver zoom-in 2 en arquitectura): el valor de cada query se obtiene como suma de los valores de referencia ponderados por sus pesos de atención correspondientes

$$\text{Attention}(Q, K, V) = \text{AttentionWeights}(Q, K) V \in \mathbb{R}^{n \times d_\text{v}}$$

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:

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

$$\text{ScaledAttentionScores}(Q,K) = \begin{pmatrix} 0.7071 & -0.7071 & -0.7070 \\ -0.7071 & 0.7071 & 0.7070 \\ -0.7070 & 0.7070 & 0.7069 \end{pmatrix}$$

$$\text{AttentionWeights}(Q,K) = \begin{pmatrix} 0.6728 & 0.1636 & 0.1636 \\ 0.1084 & 0.4458 & 0.4458 \\ 0.1084 & 0.4458 & 0.4458 \end{pmatrix}$$

$$\text{Attention}(Q,K,V) = \begin{pmatrix} 0.6728 & 0.1636 & 0.1636 \\ 0.1084 & 0.4458 & 0.4458 \\ 0.1084 & 0.4458 & 0.4458 \end{pmatrix} \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix} = \begin{pmatrix} -0.2444 & 0.2444 \\ 0.5538 & -0.5538 \\ 0.5538 & -0.5538 \end{pmatrix}$$

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([[1, 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 = scaled_scores.softmax(dim=-1); p_attn
```

Out[ ]:
```
tensor([[[0.6728, 0.1636, 0.1636],
         [0.1084, 0.4458, 0.4458],
         [0.1084, 0.4458, 0.4458]]])
```

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

Out[ ]:
```
tensor([[[-0.2444,  0.2444],
         [ 0.5538, -0.5538],
         [ 0.5538, -0.5538]]])
```

8 Atención multicabeza

Atención multicabeza (zoom-in 1): $W_i^Q, W_i^K \in \mathbb{R}^{d_{\text{model}} \times d_{\text{k}}},\ W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_{\text{v}}},\ W^O \in \mathbb{R}^{hd_{\text{v}} \times d_{\text{model}}}$

$$\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) W^O \in \mathbb{R}^{n \times d_{\text{model}}}$$

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \in \mathbb{R}^{n \times d_{\text{v}}}$$

Atención multicabeza con sesgos: añade un vector de sesgos a cada transformación lineal

Valores originales: $d_{\text{model}} = 512,\ h = 8,\ d_{\text{k}} = d_{\text{v}} = \dfrac{d_{\text{model}}}{h} = 64$

Ejercicio: halla $\text{MultiHead}(Q,K,V)$ 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([[1, 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
```

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

Out[ ]:
```
tensor([[[[ 0.5124, -0.4223],
          [ 0.7122, -0.1575],
          [ 0.7122, -0.1575]]]])
```

$$QW_1^Q + B_1^Q = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix} \begin{pmatrix} 0.8635 & 0.7223 \\ 0.5531 & 0.3659 \end{pmatrix}^t + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (0.6123, -0.2899)$$

$$= \begin{pmatrix} -0.0999 & -0.1324 \\ 0.0999 & 0.1324 \\ 0.0999 & 0.1324 \end{pmatrix} + \begin{pmatrix} 0.6123 & -0.2899 \\ 0.6123 & -0.2899 \\ 0.6123 & -0.2899 \end{pmatrix} = \begin{pmatrix} 0.5124 & -0.4223 \\ 0.7122 & -0.1575 \\ 0.7122 & -0.1575 \end{pmatrix}$$

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([[[[-0.1293,  0.2114],
          [ 0.5799, -1.0942],
          [ 0.5798, -1.0941]]]])
```

$$KW_1^K + B_1^K = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix} \begin{pmatrix} -0.0060 & -0.5075 \\ -0.0329 & 0.8903 \end{pmatrix}^t + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (0.2253, -0.4414)$$

$$= \begin{pmatrix} -0.3546 & 0.6528 \\ 0.3546 & -0.6528 \\ 0.3545 & -0.6527 \end{pmatrix} + \begin{pmatrix} 0.2253 & -0.4414 \\ 0.2253 & -0.4414 \\ 0.2253 & -0.4414 \end{pmatrix} = \begin{pmatrix} -0.1293 & 0.2114 \\ 0.5799 & -1.0942 \\ 0.5798 & -1.0941 \end{pmatrix}$$

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

Out[ ]:
```
tensor([[[[-0.1100,  0.5368,  0.5368],
          [-0.0886,  0.4138,  0.4138],
          [-0.0886,  0.4138,  0.4138]]]])
```

$$\text{ScaledAttentionScores}(QW_1^Q + B_1^Q, KW_1^K + B_1^K)$$

$$= \frac{1}{\sqrt{d_k}}(QW_1^Q + B_1^Q)(KW_1^K + B_1^K)^t$$

$$= \frac{1}{\sqrt{2}} \begin{pmatrix} 0.5124 & -0.4223 \\ 0.7122 & -0.1575 \\ 0.7122 & -0.1575 \end{pmatrix} \begin{pmatrix} -0.1293 & 0.2114 \\ 0.5799 & -1.0942 \\ 0.5798 & -1.0941 \end{pmatrix}^t = \begin{pmatrix} -0.1100 & 0.5368 & 0.5368 \\ -0.0886 & 0.4138 & 0.4138 \\ -0.0886 & 0.4138 & 0.4138 \end{pmatrix}$$

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

Out[ ]:
```
tensor([[[[0.2075, 0.3962, 0.3962],
          [0.2323, 0.3839, 0.3839],
          [0.2323, 0.3839, 0.3839]]]])
```

$$\begin{aligned}
&\text{AttentionWeights}(QW_1^Q + B_1^Q, KW_1^K + B_1^K) \\
&\quad = \text{Softmax}\Big(\text{ScaledAttentionScores}(QW_1^Q + B_1^Q, KW_1^K + B_1^K)\Big) \\
&\quad = \text{Softmax}\begin{pmatrix} -0.1100 & 0.5368 & 0.5368 \\ -0.0886 & 0.4138 & 0.4138 \\ -0.0886 & 0.4138 & 0.4138 \end{pmatrix} = \begin{pmatrix} 0.2075 & 0.3962 & 0.3962 \\ 0.2323 & 0.3839 & 0.3839 \\ 0.2323 & 0.3839 & 0.3839 \end{pmatrix}
\end{aligned}$$

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([[[[-0.5284, -0.1613],
          [ 0.6738, -1.0246],
          [ 0.6737, -1.0246]]]])
```

$$\begin{aligned}
VW_1^V + B_1^V &= \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix} \begin{pmatrix} 0.4922 & -0.3579 \\ -0.5233 & 0.0872 \end{pmatrix}^t + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (0.0727, -0.5929) \\
&= \begin{pmatrix} -0.6011 & 0.4317 \\ 0.6011 & -0.4317 \\ 0.6010 & -0.4316 \end{pmatrix} + \begin{pmatrix} 0.0727 & -0.5929 \\ 0.0727 & -0.5929 \\ 0.0727 & -0.5929 \end{pmatrix} = \begin{pmatrix} -0.5284 & -0.1613 \\ 0.6738 & -1.0246 \\ 0.6737 & -1.0246 \end{pmatrix}
\end{aligned}$$

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

Out[ ]:
```
tensor([[[ 0.4243, -0.8454],
         [ 0.3945, -0.8241],
         [ 0.3945, -0.8241]]])
```

$$
\begin{aligned}
\text{head}_1 &= \text{Attention}(QW_1^Q + B_1^Q, KW_1^K + B_1^K, VW_1^V + B_1^V) \\
&= \text{AttentionWeights}(QW_1^Q + B_1^Q, KW_1^K + B_1^K)(VW_1^V + B_1^V) \\
&= \begin{pmatrix} 0.2075 & 0.3962 & 0.3962 \\ 0.2323 & 0.3839 & 0.3839 \\ 0.2323 & 0.3839 & 0.3839 \end{pmatrix} \begin{pmatrix} -0.5284 & -0.1613 \\ 0.6738 & -1.0246 \\ 0.6737 & -1.0246 \end{pmatrix} = \begin{pmatrix} 0.4243 & -0.8454 \\ 0.3945 & -0.8241 \\ 0.3945 & -0.8241 \end{pmatrix}
\end{aligned}
$$

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([[[[0.1616, 0.3229],
          [0.1214, 0.3137],
          [0.1214, 0.3137]]]])
```

$$
\begin{aligned}
\text{MultiHead}(Q, K, V) &= \text{head}_1\, W^O + B^O \\
&= \begin{pmatrix} 0.4243 & -0.8454 \\ 0.3945 & -0.8241 \\ 0.3945 & -0.8241 \end{pmatrix} \begin{pmatrix} 1.2168 & -0.1905 \\ -0.0890 & -0.5564 \end{pmatrix}^t + \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} (-0.5157, -0.1097) \\
&= \begin{pmatrix} 0.6773 & 0.4327 \\ 0.6371 & 0.4234 \\ 0.6371 & 0.4234 \end{pmatrix} + \begin{pmatrix} -0.5157 & -0.1097 \\ -0.5157 & -0.1097 \\ -0.5157 & -0.1097 \end{pmatrix} = \begin{pmatrix} 0.1616 & 0.3229 \\ 0.1214 & 0.3137 \\ 0.1214 & 0.3137 \end{pmatrix}
\end{aligned}
$$

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

Out[ ]:
```
'tensor([[[0.1616, 0.3229],        [0.1214, 0.3137],        [0.1214, 0.3137]]])'
```

9 Conexiones residuales

Add: capa que residualiza la salida de una capa previa (para facilitar el entrenamiento de modelos muy profundos)

$$\mathrm{Add}(X, \mathrm{Layer}(X)) = X + \mathrm{Layer}(X)$$

Ejemplo: con $X = \begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix}$ y $\mathrm{Layer}(X) = \begin{pmatrix} 0.1616 & 0.3229 \\ 0.1214 & 0.3137 \\ 0.1214 & 0.3137 \end{pmatrix}$

$$\mathrm{Add}(X, \mathrm{Layer}(X)) = \begin{pmatrix} 0.0415 & 2.2411 \\ 1.2927 & -0.6469 \\ 0.9508 & 0.8249 \end{pmatrix} + \begin{pmatrix} 0.1616 & 0.3229 \\ 0.1214 & 0.3137 \\ 0.1214 & 0.3137 \end{pmatrix} = \begin{pmatrix} 0.2031 & 2.5640 \\ 1.4141 & -0.3332 \\ 1.0722 & 1.1386 \end{pmatrix}$$

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([[1, 2, 1]]); x = model.src_embed(src).data; print(x)
norm_x = model.encoder.layers[0].norm_self_attn(x).data
layer_x = model.encoder.layers[0].self_attn(norm_x, norm_x, norm_x).data; print(layer_x)
print(x+layer_x)
```

```
tensor([[[ 0.0415,  2.2411],
         [ 1.2927, -0.6469],
         [ 0.9508,  0.8249]]])
tensor([[[0.1616, 0.3229],
         [0.1214, 0.3137],
         [0.1214, 0.3137]]])
tensor([[[ 0.2031,  2.5640],
         [ 1.4141, -0.3332],
         [ 1.0722,  1.1386]]])
```

10 Position-wise Feed-Forward Networks

Position-wise Feed-Forward Network: MLP $\mathbb{R}^{d_{\text{model}}} \to \mathbb{R}^{d_{\text{model}}}$ con una capa oculta de $d_{\text{ff}} = 4\,d_{\text{model}}$ ReLUs, que se aplica por separado a cada vector de la secuencia de entrada

$$\text{PWFFN}(X) = \text{ReLU}(XW_1 + \mathbf{1}_n \boldsymbol{b}_1^t)W_2 + \mathbf{1}_n \boldsymbol{b}_2^t$$

Ejercicio: halla $\text{PWFFN}(X)_{1,:}$ 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([[1, 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],
         [-0.7070,  0.7070]]])
```

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([-0.1167,  0.0657,  0.1195, -0.4728, -0.2571, -0.1476,  0.3549, -0.1946])
tensor([0.0000, 0.0657, 0.1195, 0.0000, 0.0000, 0.0000, 0.3549, 0.0000])
tensor([ 0.2896, -0.1471])
```

$$
\begin{aligned}
\mathrm{PWFFN}(X)_{1,:} &= \mathrm{ReLU}((-0.7071, 0.7071)W_1 + \boldsymbol{b}_1^t)W_2 + \boldsymbol{b}_2^t \\
&= \mathrm{ReLU}(-0.1167, 0.0657, 0.1195, -0.4728, -0.2571, -0.1476, 0.3549, -0.1946)W_2 + \boldsymbol{b}_2^t \\
&= (0.0000, 0.0657, 0.1195, 0.0000, 0.0000, 0.0000, 0.3549, 0.0000)W_2 + \boldsymbol{b}_2^t \\
&= (0.2896, -0.1470)
\end{aligned}
$$

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([[[ 0.2896, -0.1471],
         [ 1.0716,  0.3682],
         [ 0.2896, -0.1470]]])
```

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

Out[ ]:
```
tensor([[[ 0.2896, -0.1471],
         [ 1.0716,  0.3682],
         [ 0.2896, -0.1470]]])
```

11 Encoder

Auto-atención: $\mathrm{SelfAttention}(X) = \mathrm{MultiHead}(X, X, X)$

Capa encoder: dada una secuencia de entrada $X = X^{(1)} \in \mathbb{R}^{n \times d_{\text{model}}}$, aplica

$$
\begin{aligned}
X^{(2)} &= \mathrm{LayerNorm}(X^{(1)}) \\
X^{(3)} &= \mathrm{SelfAttention}(X^{(2)}) \\
X^{(4)} &= \mathrm{Dropout}(X^{(3)}) \\
X^{(5)} &= \mathrm{Add}(X^{(1)}, X^{(4)}) \\
X^{(6)} &= \mathrm{LayerNorm}(X^{(5)}) \\
X^{(7)} &= \mathrm{PWFFN}(X^{(6)}) \\
X^{(8)} &= \mathrm{Dropout}(X^{(7)}) \\
\mathrm{EncoderLayer}(X) = X^{(9)} &= \mathrm{Add}(X^{(5)}, X^{(8)})
\end{aligned}
$$

Pre-norm vs post-norm: LayerNorm primero (pre-norm; arriba) o tras Add (post-norm; versión original)

Encoder: dada una secuencia de entrada $X = X^{(1)} \in \mathbb{R}^{n \times d_{\text{model}}}$, aplica una pila de $N = 6$ capas encoder idénticas

$$
\begin{aligned}
X^{(2)} &= \mathrm{EncoderLayer}(X^{(1)}) \\
X^{(3)} &= \mathrm{EncoderLayer}(X^{(2)}) \\
&\vdots \\
X^{(N+1)} &= \mathrm{EncoderLayer}(X^{(N)}) \\
\mathrm{Encoder}(X) &= \mathrm{LayerNorm}(X^{(N+1)})
\end{aligned}
$$

Ejemplo: $\mathrm{Encoder}(X)$ de una capa con la entrada y parámetros vistos anteriormente

$$X^{(2)} = \mathrm{LayerNorm}(X^{(1)}) = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix}$$

$$X^{(3)} = \mathrm{SelfAttention}(X^{(2)}) = \begin{pmatrix} 0.1616 & 0.3229 \\ 0.1214 & 0.3137 \\ 0.1214 & 0.3137 \end{pmatrix}$$

$$X^{(5)} = \mathrm{Add}(X^{(1)}, X^{(3)}) = \begin{pmatrix} 0.2031 & 2.5640 \\ 1.4141 & -0.3332 \\ 1.0722 & 1.1386 \end{pmatrix}$$

$$X^{(6)} = \mathrm{LayerNorm}(X^{(5)}) = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7070 & -0.7070 \end{pmatrix}$$

$$X^{(7)} = \mathrm{PWFFN}(X^{(6)}) = \begin{pmatrix} 0.2896 & -0.1471 \\ 1.0716 & 0.3682 \\ 0.2896 & -0.1470 \end{pmatrix}$$

$$X^{(9)} = \mathrm{Add}(X^{(5)}, X^{(7)}) = \begin{pmatrix} 0.4927 & 2.4170 \\ 2.4857 & 0.0350 \\ 1.3618 & 0.9916 \end{pmatrix}$$

$$\mathrm{Encoder}(X) = \mathrm{LayerNorm}(X^{(9)}) = \begin{pmatrix} -0.7071 & 0.7071 \\ 0.7071 & -0.7071 \\ 0.7071 & -0.7071 \end{pmatrix}$$

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([[1, 2, 1]])
x1 = model.src_embed(src).data
x2 = model.encoder.layers[0].norm_self_attn(x1).data
x3 = model.encoder.layers[0].self_attn(x2, x2, x2).data
x5 = x1 + x3
x6 = model.encoder.layers[0].norm_ff(x5)
x7 = model.encoder.layers[0].ff(x6).data; print(x7)
x9 = x5 + x7; print(x9)
print(model.encoder.layers[0](x1, None).data)
print(model.encoder.norm(x9).data)
print(model.encode(src, None).data)
```

```
tensor([[[ 0.2896, -0.1471],
         [ 1.0716,  0.3682],
         [ 0.2896, -0.1470]]])
tensor([[[0.4927, 2.4170],
         [2.4857, 0.0350],
         [1.3618, 0.9916]]])
tensor([[[0.4927, 2.4170],
         [2.4857, 0.0350],
         [1.3618, 0.9916]]])
tensor([[[-0.7071,  0.7071],
         [ 0.7071, -0.7071],
         [ 0.7071, -0.7071]]])
tensor([[[-0.7071,  0.7071],
         [ 0.7071, -0.7071],
         [ 0.7071, -0.7071]]])
```

12 Decoder

Atención cruzada: $\mathrm{CrossAttention}(X, Y) = \mathrm{MultiHead}(Y, X, X)$

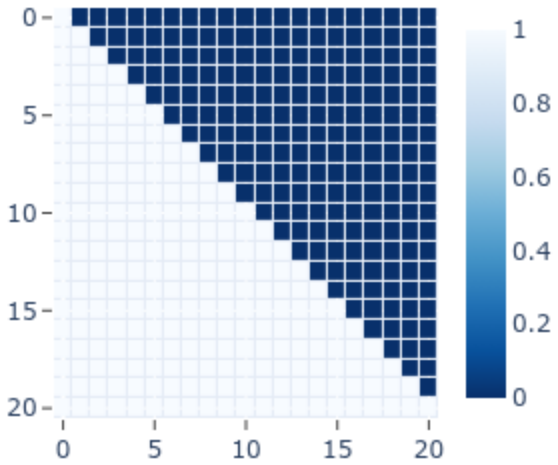
Masking: introducimos una máscara $M \in \{0, 1\}^{n \times n}$ de scores a ignorar (con 0)

$$\mathrm{AttentionWeights}(Q, K, M) = \mathrm{Softmax}\left(\frac{1}{\sqrt{d_k}} QK^t \odot M\right) \in \mathbb{R}^{n \times n}$$

Masking causal: anula pesos de atención a queries de salida no generados aún

In [ ]:
```
def create_causal_mask(size):
    "Mask out subsequent positions to preserve the auto-regressive property."
    attn_shape = (1, size, size); causal_mask = torch.triu(torch.ones(attn_shape), diagonal=1).type(torch.uint8)
    return causal_mask == 0
```

In [ ]:
```
import torch; import plotly.graph_objects as go; n = 21; matrix = create_causal_mask(n).int().numpy()[0]
heat = go.Heatmap(z=matrix, xgap=1.1, ygap=1.1, colorscale='Blues_r', colorbar_thickness=20, colorbar_ticklen=3)
layout = go.Layout(width=300, height=250, yaxis_autorange='reversed', xaxis=dict(ticks='outside'),
    yaxis=dict(ticks='outside'), margin=dict(l=10,r=10,t=10,b=10))
fig=go.Figure(data=[heat], layout=layout); fig.show(renderer="png", width=300, height=250)
```



Capa decoder: dada una entrada (al decoder) X, salida $Y = Y^{(1)}$ y máscara causal M, aplica

$$
\begin{aligned}
Y^{(2)} &= \mathrm{LayerNorm}(Y^{(1)}) & \qquad Y^{(8)} &= \mathrm{Dropout}(Y^{(7)}) \\
Y^{(3)} &= \mathrm{SelfAttention}(Y^{(2)}, M) & \qquad Y^{(9)} &= \mathrm{Add}(Y^{(5)}, Y^{(8)}) \\
Y^{(4)} &= \mathrm{Dropout}(Y^{(3)}) & \qquad Y^{(10)} &= \mathrm{LayerNorm}(Y^{(9)}) \\
Y^{(5)} &= \mathrm{Add}(Y^{(1)}, Y^{(4)}) & \qquad Y^{(11)} &= \mathrm{PWFFN}(Y^{(10)}) \\
Y^{(6)} &= \mathrm{LayerNorm}(Y^{(5)}) & \qquad Y^{(12)} &= \mathrm{Dropout}(Y^{(10)}) \\
Y^{(7)} &= \mathrm{CrossAttention}(X, Y^{(6)}) & \qquad Y^{(13)} &= \mathrm{Add}(Y^{(9)}, Y^{(12)})
\end{aligned}
$$

$$
\mathrm{DecoderLayer}(X, Y, M) = Y^{(13)}
$$

Pre-norm vs post-norm: LayerNorm primero (pre-norm; arriba) o tras Add (post-norm; versión original)

Decoder: dadas una entrada X, salida Y y máscara M, aplica una pila de $N = 6$ capas decoder idénticas

$$
\begin{aligned}
Y^{(2)} &= \mathrm{DecoderLayer}(X, Y^{(1)}, M) \\
Y^{(3)} &= \mathrm{DecoderLayer}(X, Y^{(2)}, M) \\
&\vdots \\
Y^{(N+1)} &= \mathrm{DecoderLayer}(X, Y^{(N)}, M) \\
\mathrm{Decoder}(X, Y, M) &= \mathrm{LayerNorm}(Y^{(N+1)})
\end{aligned}
$$

Ejemplo: $\text{Decoder}(X, Y, M)$ de una capa con la entrada vista anteriormente y salida inicial

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([[1, 2, 1]]); src_mask = torch.ones(1, 1, 3) # None funciona igual (sin padding)
ys = torch.zeros(1, 1).type_as(src); tgt_mask = create_causal_mask(ys.size(1)).type_as(src.data)
print("src =", src, "src_mask =", src_mask, "ys =", ys, "tgt_mask =", tgt_mask)
x1 = model.encode(src, None).data; print("x1 =", str(x1).replace('\n',''))
y1 = model.tgt_embed(ys).data; print("y1 =", y1)
y2 = model.decoder.layers[0].norm_self_attn(y1).data; print("y2 =", y2)
y3 = model.decoder.layers[0].self_attn(y2, y2, y2, tgt_mask).data; print("y3 =", y3)
y4 = model.decoder.layers[0].dropout(y3).data; print("y4 =", y4)
y5 = y1 + y4; print("y5 =", y5)
y6 = model.decoder.layers[0].norm_cross_attn(y5).data; print("y6 =", y6)
y7 = model.decoder.layers[0].cross_attn(y6, x1, x1, None).data; print("y7 =", y7)
y8 = model.decoder.layers[0].dropout(y7).data; print("y8 =", y8)
y9 = y5 + y8; print("y9 =", y9)
y10 = model.decoder.layers[0].norm_ff(y9).data; print("y10 =", y10)
y11 = model.decoder.layers[0].ff(y10).data; print("y11 =", y11)
y12 = model.decoder.layers[0].dropout(y11).data; print("y12 =", y12)
y13 = y9 + y12; print("y13 =", y13)
print(model.decoder.layers[0](y1, x1, None, tgt_mask).data)
print(model.decoder.norm(y13).data, model.decode(ys, x1, None, tgt_mask).data)
```

```
src = tensor([[1, 2, 1]]) src_mask = tensor([[[1., 1., 1.]]]) ys = tensor([[0]]) tgt_mask = tensor([[[1]]])
x1 = tensor([[[-0.7071,  0.7071],         [ 0.7071, -0.7071],         [ 0.7071, -0.7071]]])
y1 = tensor([[[1.3548, 1.4321]]])
y2 = tensor([[[-0.7070,  0.7070]]])
y3 = tensor([[[-0.7602, -0.0424]]])
y4 = tensor([[[-0.7602, -0.0424]]])
y5 = tensor([[[0.5946, 1.3897]]])
y6 = tensor([[[-0.7071,  0.7071]]])
y7 = tensor([[[0.9877, 1.1094]]])
y8 = tensor([[[0.9877, 1.1094]]])
y9 = tensor([[[1.5823, 2.4991]]])
y10 = tensor([[[-0.7071,  0.7071]]])
y11 = tensor([[[0.8719, 1.1273]]])
y12 = tensor([[[0.8719, 1.1273]]])
y13 = tensor([[[2.4542, 3.6264]]])
tensor([[[2.4542, 3.6264]]])
tensor([[[-0.7071,  0.7071]]]) tensor([[[-0.7071,  0.7071]]])
```

13 Transformer

Parámetros:

- `src_vocab_size`: talla del vocabulario de entrada
- `tgt_vocab_size`: talla del vocabulario de salida
- `embed_dim`: dimensión de embedding (tanto de entrada como de salida), $d_{\text{model}} = 512$ en artículo
- `num_layers`: número de capas del encoder y decoder, $N = 6$ en artículo
- `num_heads`: número de cabezas en atención multicabeza, $h = 8$ en artículo
- `dropout=0.1`: probabilidad de dropout, $P_{\text{drop}} = 0.1$ en artículo
- `pre_norm=True`: LayerNorm primero; tras Add en artículo
- `pe_type='fixed'`: positional encoding fijo o entrenable; fijo en artículo, aunque prueban a entrenar

`Generator:` clase auxiliar que implementa la cabeza lineal con una transformación lineal sin sesgo

In [ ]:
```
%%script true
class Generator(nn.Module):
    """
    Define decoder side language model head (`final_proj`, a linear projection) and softmax for generation.
    Note: To tie the weights of final_proj with nn.Embedding, bias term in final_proj is set to be False.
    """
    def __init__(self, embed_dim, vocab_size):
        super().__init__(); self.final_proj = nn.Linear(embed_dim, vocab_size, bias=False) # projection to voc size
    def forward(self, x):
        return F.log_softmax(self.final_proj(x), dim=-1) # softmax and log
```

`Transformer`: subclase de `nn.Module`

- **Instanciación:** inicializa embeddings, encoder, decoder y cabeza de `weight` ligado al del embedding de la salida

In [ ]:
```
%%script true
self.src_embed = EmbeddingsWithPositionalEncoding(src_vocab_size, embed_dim, dropout, pe_type)
self.tgt_embed = EmbeddingsWithPositionalEncoding(tgt_vocab_size, embed_dim, dropout, pe_type)
self.encoder = Encoder(embed_dim, num_layers, num_heads, dropout, pre_norm)
self.decoder = Decoder(embed_dim, num_layers, num_heads, dropout, pre_norm)
self.generator = Generator(embed_dim, tgt_vocab_size)
self.generator.final_proj.weight = self.tgt_embed.embed.weight
```

- **`encode`**: implementa $\mathrm{Encoder}(X)$ tras obtener el embedding de la entrada, X

In [ ]:
```
%%script true
def encode(self, src, src_mask):
    x = self.src_embed(src) # embedding of src
    enc_out = self.encoder(x, src_mask) # encoder output
    return enc_out
```

- **`decode`**: implementa $\mathrm{Decoder}(X, Y, M)$ tras obtener el embedding de la salida, Y

In [ ]:
```
%%script true
def decode(self, tgt, memory, src_mask, tgt_mask):  # the order of args should be consistent across modules
    x = self.tgt_embed(tgt) # embedding of tgt
    dec_out = self.decoder(x, memory, src_mask, tgt_mask) # 'encoder output' serves as K,V in 'decoder x-attention'
    return dec_out
```

- **`forward`**: implementa $\mathrm{Transformer}(X, Y, M) = \mathrm{Decoder}(\mathrm{Encoder}(X), Y, M)$

In [ ]:
```
%%script true
def forward(self, src, tgt, src_mask, tgt_mask):
    memory = self.encode(src, src_mask) # encoder output (memory)
    dec_out = self.decode(tgt, memory, src_mask, tgt_mask) # decoder output
    return dec_out
```

- **`tie_weight`**: liga el weight del embedding de entrada con el de salida (p. ej. si comparten vocs)

In [ ]:
```
%%script true
def tie_weights(self):
    self.src_embed.embed.weight = self.tgt_embed.embed.weight
    print("Source (encoder) and target (decoder) embedding weights are now tied.")
```

- **Atributo `device`**: determina el dispositivo en el que se halla el modelo

In [ ]:
```
%%script true
@property
def device(self) -> torch.device:
    return next(self.parameters()).device
```

Creación e inicialización aleatoria de un modelo: función usada en los ejemplos vistos anteriormente

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at251 import Transformer, create_causal_mask
```

In [ ]:
```
def create_model(src_vocab_size, tgt_vocab_size, embed_dim=512, num_layers=6, num_heads=8, dropout=0.1,
    pre_norm=True, pe_type='fixed', device=None):
    model = Transformer(src_vocab_size, tgt_vocab_size, embed_dim, num_layers, num_heads, dropout,
        pre_norm, pe_type)
    for p in model.parameters():
        if p.dim() > 1:
            nn.init.xavier_uniform_(p)
    if device is not None:      # note: `0` indicates `cuda:0`
        model = model.to(device)
    return model
```

Prueba rápida de inferencia para probar el código:

In [ ]:
```python
def inference_test():
    test_model = create_model(src_vocab_size=11, tgt_vocab_size=11, embed_dim=512, num_layers=2, num_heads=8,
        dropout=0.1, pre_norm=True, pe_type='fixed', device=None)
    # do not tie the weight for a random test:
    test_model.generator.final_proj.weight = nn.Parameter(torch.randn_like(test_model.generator.final_proj.weight))
    test_model.eval()
    src = torch.randint(1, 11, (1, 10)); src_mask = torch.ones(1, 1, 10)
    memory = test_model.encode(src, src_mask); ys = torch.zeros(1, 1).type_as(src) # ys[0]=0
    # model rollout
    for i in range(9):
        tgt_mask = create_causal_mask(ys.size(1)).type_as(src.data)
        out = test_model.decode(ys, memory, src_mask, tgt_mask)
        prob = test_model.generator(out[:, -1])     # last token
        _, next_word = torch.max(prob, dim=1)
        next_word = next_word.data[0]
        ys = torch.cat([ys, torch.empty(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    print(f"{src} -> {ys}")
```

In [ ]:
```python
for _ in range(10):
    inference_test()
```

```
tensor([[ 7,  7,  1,  3,  7, 10,  5,  8,  1,  4]]) -> tensor([[0, 6, 3, 6, 3, 6, 3, 6, 3, 9]])
tensor([[5, 6, 1, 9, 6, 4, 8, 8, 9, 3]]) -> tensor([[0, 1, 7, 7, 7, 7, 7, 7, 7, 7]])
tensor([[ 5,  9,  6,  7,  8,  6,  1,  2,  9, 10]]) -> tensor([[0, 9, 9, 9, 9, 9, 0, 0, 0, 0]])
tensor([[ 9,  7,  2,  1, 10, 10, 10,  9,  3,  2]]) -> tensor([[0, 4, 1, 1, 1, 1, 1, 1, 1, 1]])
tensor([[3, 1, 8, 9, 4, 3, 9, 4, 3, 8]]) -> tensor([[0, 1, 6, 1, 8, 3, 3, 3, 3, 3]])
tensor([[ 9,  6,  6,  2,  9,  1, 10,  9, 10,  5]]) -> tensor([[ 0,  9,  5,  0,  9, 10,  0,  9, 10,  0]])
tensor([[10, 10,  6,  5,  3,  5,  6,  8,  8,  4]]) -> tensor([[0, 9, 8, 4, 9, 8, 8, 8, 9, 8]])
tensor([[4, 2, 5, 4, 9, 6, 8, 2, 3, 5]]) -> tensor([[0, 4, 6, 3, 4, 6, 3, 0, 7, 7]])
tensor([[7, 2, 1, 4, 3, 6, 4, 8, 5, 2]]) -> tensor([[0, 8, 7, 7, 7, 7, 7, 7, 7, 7]])
tensor([[5, 2, 9, 2, 7, 7, 4, 7, 8, 3]]) -> tensor([[ 0,  7,  1,  2,  0, 10, 10, 10, 10, 10]])
```

14 Entrenamiento

Tarea "copia": el modelo debe aprender a copiar la secuencia aleatoria de símbolos dada

Batching y masking: consiste en preparar datos y máscaras asociadas para entrenar el modelo

`data_generator`: generador de datos sintéticos basado en una clase `Batch` para batching y masking

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23)
import import_ipynb; from at2523 import data_generator
```

In [ ]:
```
V = 11; num_batches = 1; batch_size = 1
data_iter = data_generator(V, batch_size, nbatches=num_batches)
for i, batch in enumerate(data_iter):
    print(f"Batch {i}:")
    print(f"src:", str(batch['src']).replace('\n',''))
    print(f"src_mask:", str(batch['src_mask']).replace('\n',''))
    print(f"tgt:", str(batch['tgt']).replace('\n','')) # decoder input
    print(f"tgt_y:", str({batch['tgt_y']}).replace('\n','')) # decoder target
    print(f"tgt_mask:\n", batch['tgt_mask'])
    print(f"num_tokens:", batch['num_tokens'].item())
```

```
Batch 0:
src: tensor([[1, 7, 7, 8, 1, 3, 8, 2, 5, 8]])
src_mask: tensor([[[True, True, True, True, True, True, True, True, True, True]]])
tgt: tensor([[1, 7, 7, 8, 1, 3, 8, 2, 5]])
tgt_y: {tensor([[7, 7, 8, 1, 3, 8, 2, 5, 8]])}
tgt_mask:
 tensor([[[ True, False, False, False, False, False, False, False, False],
         [ True,  True, False, False, False, False, False, False, False],
         [ True,  True,  True, False, False, False, False, False, False],
         [ True,  True,  True,  True, False, False, False, False, False],
         [ True,  True,  True,  True,  True, False, False, False, False],
         [ True,  True,  True,  True,  True,  True, False, False, False],
         [ True,  True,  True,  True,  True,  True,  True, False, False],
         [ True,  True,  True,  True,  True,  True,  True,  True, False],
         [ True,  True,  True,  True,  True,  True,  True,  True,  True]]])
num_tokens: 9
```

Pérdida: entropía cruzada de la predicción $\hat{\boldsymbol{y}}$ relativa a la referencia con label smoothing, $\tilde{\boldsymbol{y}}$

$$\mathbb{H}(\hat{\boldsymbol{y}}, \tilde{\boldsymbol{y}}) = -\sum_c \tilde{y}_c \log \hat{y}_c \qquad \text{donde} \qquad \tilde{\boldsymbol{y}} = (1-\epsilon)\,\boldsymbol{y} + \epsilon\,\frac{1}{C}\mathbf{1}_C \quad \text{con} \quad \epsilon = 0.1$$

Ejemplo: $C = 2$, $\mathcal{C} = \{0, 1\}$, $\hat{\boldsymbol{y}} = (0.2123, 0.7877)^t$, $\boldsymbol{y} = (0.5738, 0.4262)^t$

$$\mathbb{H}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -0.5738 \log 0.2123 - 0.4262 \log 0.7877 = 0.9910$$

$$\tilde{\boldsymbol{y}} = 0.9\,\boldsymbol{y} + 0.1\,(0.5, 0.5)^t = (0.5664, 0.4336)^t$$
$$\mathbb{H}(\hat{\boldsymbol{y}}, \tilde{\boldsymbol{y}}) = -0.5664 \log 0.2123 - 0.4336 \log 0.7877 = 0.9813$$

In [ ]:
```python
import torch; import torch.nn as nn; torch.manual_seed(23); torch.set_printoptions(precision=4)
H = nn.CrossEntropyLoss(label_smoothing=0.0)
y_pred_logits = torch.randn(1, 2); print("y_pred_logits =", y_pred_logits)
y_pred_probs = y_pred_logits.softmax(dim=1); print("y_pred_probs =", y_pred_probs)
y_probs = torch.randn(1, 2).softmax(dim=1); print("y_probs =", y_probs)
print(f"H({y_pred_logits}, {y_probs}, epsilon=0.0) = ", H(y_pred_logits, y_probs))
H = nn.CrossEntropyLoss(label_smoothing=0.1)
print(f"H({y_pred_logits}, {y_probs}, epsilon=0.1) = ", H(y_pred_logits, y_probs))
```

```
y_pred_logits = tensor([[-0.8733,  0.4376]])
y_pred_probs = tensor([[0.2123, 0.7877]])
y_probs = tensor([[0.5738, 0.4262]])
H(tensor([[-0.8733,  0.4376]]), tensor([[0.5738, 0.4262]]), epsilon=0.0) =  tensor(0.9909)
H(tensor([[-0.8733,  0.4376]]), tensor([[0.5738, 0.4262]]), epsilon=0.1) =  tensor(0.9812)
```

Optimizador: adaptive moment estimation (Adam) con $\beta_1 = 0.9,\ \beta_2 = 0.98$ y $\epsilon = 10^{-9}$; aunque actualmente se usa una versión mejorada, Adam with decoupled weight decay (AdamW)

- **Entrada:** lr $\gamma,\ \beta_1, \beta_2 \in [0, 1),\ \theta_0,$ objetivo estocástico a minimizar $f(\theta),$ weight decay $\lambda,\ \epsilon$
- **Inicialización:** primer momento $m_0 \leftarrow 0,$ segundo momento $v_0 \leftarrow 0$
- **Iteración** $i = 1, \ldots:$
  1. $g_i \leftarrow \nabla_\theta f_i(\theta_{i-1})$ (obtiene gradiente con respecto al objetivo estocástico en la iteración i)
  2. $\theta_i \leftarrow \theta_{i-1} - \gamma\,\lambda\,\theta_{i-1}$ (weight decay; $g_i \leftarrow g_i + \lambda\,\theta_{i-1}$ en Adam lo acumula en momentos)
  3. $m_i \leftarrow \beta_1\,m_{i-1} + (1 - \beta_1)\,g_i$ (actualiza estimador sesgado del primer momento)
  4. $v_i \leftarrow \beta_2\,v_{i-1} + (1 - \beta_2)\,g_i^2$ (actualiza estimador sesgado del segundo momento)
  5. $\widehat{m_i} \leftarrow m_i \,/\, \left(1 - \beta_1^i\right)$ (corrige el sesgo del estimador del primer momento)
  6. $\widehat{v_i} \leftarrow v_i \,/\, \left(1 - \beta_2^i\right)$ (corrige el sesgo del estimador del segundo momento)
  7. $\theta_i \leftarrow \theta_{i-1} - \gamma\,\widehat{m_i} \,/\, \left(\sqrt{\widehat{v_i}} + \epsilon\right)$ (actualización de parámetros)

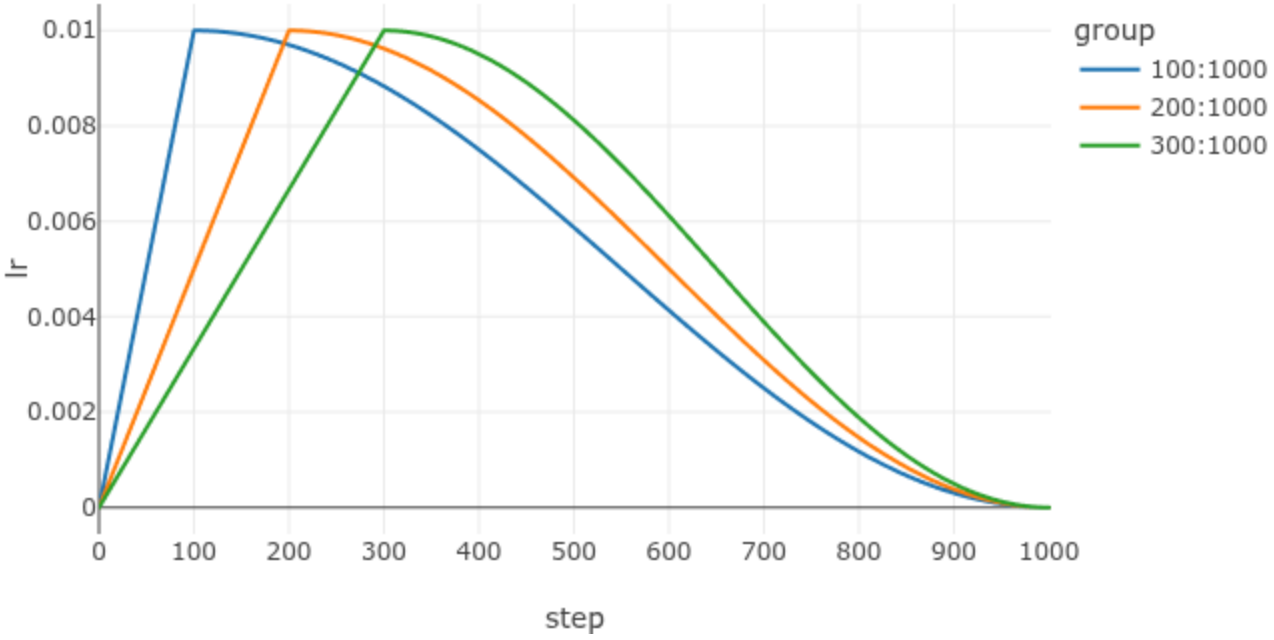
Planificador original: incrementa el learning rate linealmente durante los primeros `warmup_steps` y luego lo decrementa proporcionalmente a la inversa de la raíz de `step_num`

$$lrate = d_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

Incoveniente del scheduler original: su dependencia del tamaño del modelo, lo que dificulta comparar modelos

Planificador actual: cosine Annealing with Warmup primero incrementa el learning rate linealmente, desde cero hasta un valor inicial prefijado, y luego lo decrementa hasta cero con base en la función coseno

In [ ]:
```
import pandas as pd; import torch; from torch.optim.lr_scheduler import LambdaLR; import plotly.express as px
import import_ipynb; from at2523 import cosine_schedule_with_warmup
def get_lr_curve(num_warmup_steps, num_training_steps):
    dummy_model = torch.nn.Linear(1, 1); results = []; base_lr = 0.01; num_steps = 1001
    optimizer = torch.optim.AdamW(dummy_model.parameters(), lr=base_lr, betas=(0.9, 0.98), eps=1e-9)
    lr_scheduler = cosine_schedule_with_warmup(optimizer=optimizer,
        num_warmup_steps=num_warmup_steps, num_training_steps=num_training_steps)
    for step in range(num_steps):
        lr = optimizer.param_groups[0]["lr"]; optimizer.step(); lr_scheduler.step()
        results.append({'step': step, 'lr': lr, 'group': f"{num_warmup_steps}:{num_training_steps}"})
    return pd.DataFrame(results)
def plot_lr_curve(df):
    fig = px.line(df, x="step", y="lr", color="group", template='none')
    xaxis = dict(tickmode='linear', tick0=0, dtick=100, range=[0, 1002]); width = 650; height = 325
    fig.update_layout(width=width, height=height, xaxis=xaxis, margin=dict(l=50,r=50,t=10,b=50))
    fig.show(renderer="png", width=width, height=height)
df1 = get_lr_curve(100, 1000); df2 = get_lr_curve(200, 1000); df3 = get_lr_curve(300, 1000)
plot_lr_curve(pd.concat([df1, df2, df3]))
```



Entrenamiento:

In [ ]:
```
import torch; import torch.nn as nn; torch.manual_seed(23); import import_ipynb
from at251 import create_model, create_causal_mask; from at2523 import data_generator, cosine_schedule_with_warmup
V = 11; model = create_model(V, V, embed_dim=512, num_layers=2, num_heads=1, dropout=0.1)
device = torch.accelerator.current_accelerator().type if torch.accelerator.is_available() else "cpu"
model = model.to(device); model.train()
num_batches = 50; batch_size = 100; num_epochs = 20; num_training_steps = num_epochs * num_batches
warmup_ratio = 0.1; num_warmup_steps = int(num_training_steps * warmup_ratio)
criterion = nn.CrossEntropyLoss(label_smoothing=0.1)
base_lr=0.001; optimizer = torch.optim.AdamW(model.parameters(), lr=base_lr, betas=(0.9, 0.98), eps=1e-9)
scheduler = cosine_schedule_with_warmup(optimizer, num_warmup_steps, num_training_steps)
for epoch in range(num_epochs):
    loss = 0; tokens = 0
    for i, batch in enumerate(data_generator(V, batch_size, nbatches=num_batches)):
        batch = batch.to(device); logits = model(batch.src, batch.tgt, batch.src_mask, batch.tgt_mask)
        y_pred = model.generator(logits)
        loss = criterion(y_pred.reshape(-1, y_pred.shape[-1]), batch.tgt_y.reshape(-1))
        loss.backward(); optimizer.step(); optimizer.zero_grad(set_to_none=True); scheduler.step()
        loss += loss.item(); tokens += batch.num_tokens
    if epoch % 5 == 0 or epoch == num_epochs-1:
        print(f"Epoch {epoch} loss {loss:<.4f} tokens {tokens} avg_loss {loss/tokens:g}")
```

```
Epoch 0 loss 4.7656 tokens 45000 avg_loss 0.000105903
Epoch 5 loss 1.3655 tokens 45000 avg_loss 3.03441e-05
Epoch 10 loss 1.1685 tokens 45000 avg_loss 2.59673e-05
Epoch 15 loss 1.1055 tokens 45000 avg_loss 2.45659e-05
Epoch 19 loss 1.0713 tokens 45000 avg_loss 2.38068e-05
```

In [ ]:
```
model.eval(); src = torch.randint(1, 11, (1, 10)).to(device); src_mask = torch.ones(1, 1, 10).to(device)
memory = model.encode(src, src_mask); ys = torch.zeros(1, 1).type_as(src).to(device)
for i in range(9):
    tgt_mask = create_causal_mask(ys.size(1)).type_as(src.data)
    out = model.decode(ys, memory, src_mask, tgt_mask); prob = model.generator(out[:, -1])
    _, next_word = torch.max(prob, dim=1); next_word = next_word.data[0]
    ys = torch.cat([ys, torch.empty(1, 1).type_as(src.data).fill_(next_word)], dim=1)
print(f"{src}\n{ys}")
```

```
tensor([[10,  1,  5,  3,  3, 10,  3,  8,  7,  6]], device='cuda:0')
tensor([[ 0,  1,  5,  3,  3, 10,  3,  8,  7,  6]], device='cuda:0')
```