El Transformer: "Attention Is All You Need"

El artículo "Attention Is All You Need", que introduce la arquitectura Transformer, es uno de los trabajos más influyentes en aprendizaje profundo y procesamiento del lenguaje natural (NLP). Demostró que, mediante mecanismos de atención y sin recurrir a recurrencia, era posible alcanzar resultados de vanguardia en traducción automática. A partir de esta arquitectura se desarrollaron modelos posteriores, como BERT, con un rendimiento sobresaliente en una amplia variedad de tareas de NLP.

A lo largo de este documento se emplea una combinación de figuras propias, figuras adaptadas del artículo original, ecuaciones, tablas y fragmentos de código, con el fin de explicar el Transformer con claridad y rigor. Al finalizar la lectura, el lector debería tener una comprensión sólida de los componentes principales del modelo.

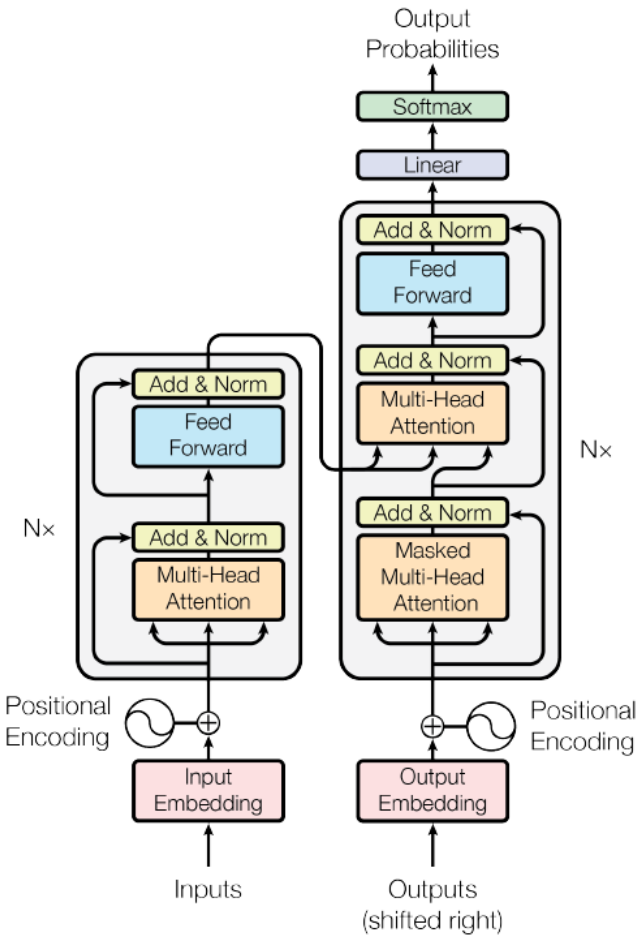


Figura 1. Tomada del artículo original del Transformer.

Artículo original

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention Is All You Need. Advances in Neural Information Processing Systems (NeurIPS), 30, 5998-6008.

Todas las citas y referencias técnicas de este documento se basan en el artículo original del Transformer.

Implementación de referencia en PyTorch

Para la implementación en PyTorch del modelo Transformer se toma como referencia The Annotated Transformer, un cuaderno de IPython que intercala el texto del artículo con código PyTorch funcional. A lo largo de este documento se harán referencias a esa implementación y se explicarán algunos fragmentos en detalle.

Nótese que el orden de exposición de las partes del Transformer en este documento difiere del artículo original y de The Annotated Transformer. Aquí se sigue el flujo de datos del modelo: se parte de una oración en inglés ("I like trees") y se recorre el Transformer hasta obtener su traducción al español ("Me gustan los árboles").

Los hiperparámetros utilizados aquí son los del modelo base Transformer, como se muestra en este extracto de la Tabla 3 del artículo Transformer:

	N	d_{model}	d_{ff}	h	d_k	d_v	P_{drop}	ϵ_{ls}	train steps
base	6	512	2048	8	64	64	0.1	0.1	100K

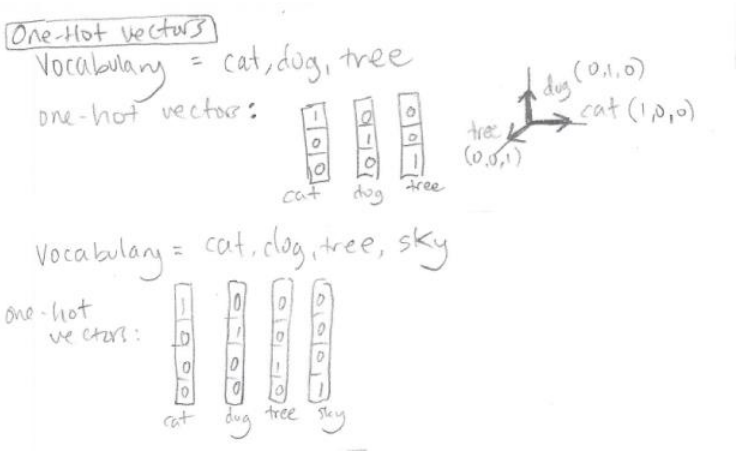
Estos son los mismos hiperparámetros utilizados en el código, en la función "make_model(src_vocab, tgt_vocab, N=6, d_model=512, d_ff=2048, h=8, dropout=0.1)".

Representación de entradas y salidas

En una tarea de traducción inglés-español, la entrada del Transformer es una oración en inglés ("I like trees") y la salida es su traducción al español ("Me gustan los árboles").

Representación de la entrada

Primero, representamos cada palabra de la oración de entrada con un vector one-hot. Un vector one-hot es un vector en el que todos los elementos son cero, excepto uno, que vale uno:



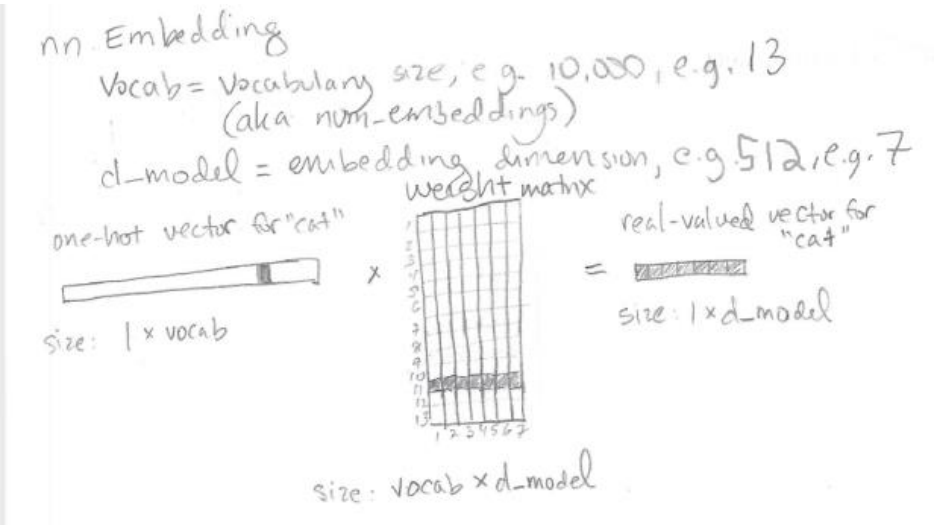
La longitud de cada vector one-hot está determinada por el tamaño del vocabulario. Si queremos representar 10,000 palabras, necesitamos vectores one-hot de longitud 10,000 (de modo que cada palabra tenga una posición única para el "1").

Embeddings de palabras

No es recomendable alimentar el Transformer con vectores one-hot, porque son dispersos, grandes y no capturan similitudes semánticas entre palabras. Por ello se aprende un embedding de palabras: una representación vectorial densa de valores reales que contiene información útil sobre cada palabra. En PyTorch esto se implementa con nn.Embedding; conceptualmente, equivale a multiplicar el vector one-hot por una matriz de pesos aprendible W.

nn.Embedding usa una matriz de pesos W para transformar un vector one-hot en un vector de valores reales. Esa matriz tiene forma (num_embeddings, embedding_dim). num_embeddings es el tamaño del vocabulario (un embedding por palabra) y embedding_dim es la dimensión del vector denso, que puede ser 3, 64, 256, 512, etc. En el artículo de Transformer se usa 512 (el hiperparámetro d_model = 512).

A menudo se describe nn.Embedding como una "tabla de consulta", porque la matriz de pesos puede verse como una tabla que almacena un vector denso para cada palabra del vocabulario:



Hay dos formas comunes de usar la matriz de pesos de nn.Embedding. Una es inicializarla con embeddings preentrenados y mantenerla fija; en ese caso funciona como una tabla de búsqueda. La otra es inicializarla aleatoriamente (o con embeddings preentrenados) y dejarla entrenable. En ese caso, las representaciones se refinan durante el entrenamiento. El Transformer usa inicialización aleatoria y aprende sus propios embeddings.

En The Annotated Transformer, los embeddings de palabras se construyen con la clase "Embeddings", que internamente utiliza nn.Embedding.

```
class Embeddings(nn.Module):
    def __init__(self, d_model, vocab):
        super(Embeddings, self).__init__()
        self.lut = nn.Embedding(vocab, d_model)
        self.d_model = d_model

    def forward(self, x):
        return self.lut(x) * math.sqrt(self.d_model)
```

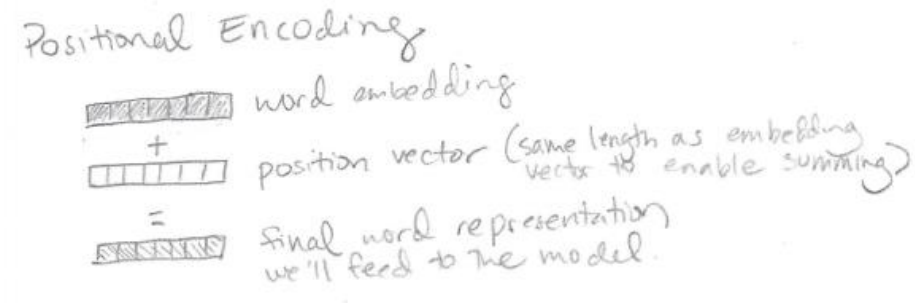
Codificación posicional

Podríamos usar solo embeddings de palabras como representación de entrada. Sin embargo, por sí solos no contienen información sobre el orden de las palabras en la oración:

- "I like trees" y "The trees grew" ambos contienen la palabra "trees"
- La palabra "trees" tiene exactamente el mismo embedding, sin importar si aparece en la segunda o en la tercera posición de una oración.

En una RNN esto no sería problema, porque la oración se procesa secuencialmente, palabra por palabra. En un Transformer, en cambio, todas las palabras se procesan en paralelo: no existe un orden implícito en el cómputo y, por tanto, no hay información posicional inherente.

Para resolverlo, los autores agregan una codificación posicional. Esta codificación permite incorporar el orden de las palabras mediante una secuencia de vectores de valores reales que se suman a los embeddings según la posición de cada token en la oración:



¿Cómo transporta el "vector de posición" la información posicional? Los autores evaluaron dos alternativas para construir los vectores de codificación posicional:

- Opción 1: aprender los vectores de codificación posicional (requiere parámetros entrenables).
- Opción 2: calcular los vectores de codificación posicional con una fórmula (no requiere parámetros entrenables).

Como el rendimiento fue similar en ambas opciones, eligieron la segunda (cálculo), ya que no añade parámetros y además puede generalizar mejor a longitudes de oración no vistas durante el entrenamiento.

Esta es la fórmula que se usa para calcular la codificación posicional:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

En esta ecuación,

- pos es la posición de una palabra en la oración (por ejemplo, "2" para la segunda palabra en la oración)
- i indexa la dimensión del embedding, es decir, la posición dentro del vector de codificación posicional. Para un vector de longitud d_model = 512, i recorre las dimensiones del vector.

¿Por qué usar seno y coseno? Como señalan los autores, "cada dimensión de la codificación posicional corresponde a una sinusoide". Eligieron esta formulación porque plantearon la hipótesis de que facilitaría el aprendizaje de relaciones de posición relativa entre tokens.

En Annotated Transformer, la codificación posicional se crea y se agrega a los embeddings de palabras utilizando la clase "PositionalEncoding":

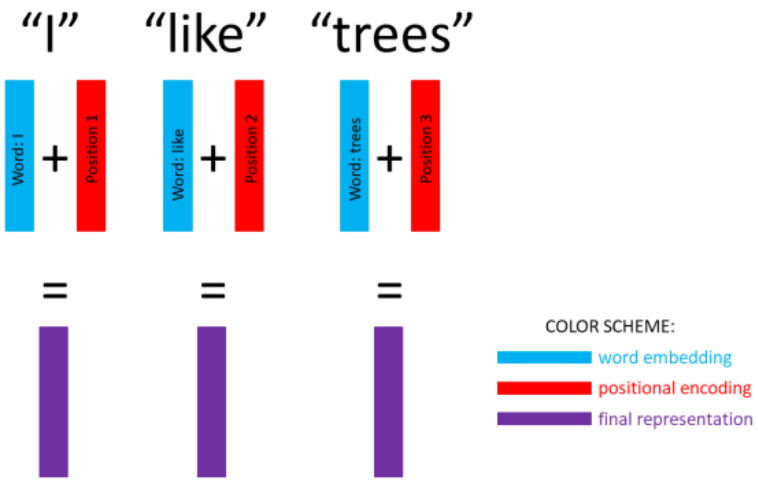
```
class PositionalEncoding(nn.Module):
    "Implement the PE function."
    def __init__(self, d_model, dropout, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Compute the positional encodings once in log space.
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) *
                             -(math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x + Variable(self.pe[:, :x.size(1)],
                         requires_grad=False)
        return self.dropout(x)
```

Resumen de la entrada

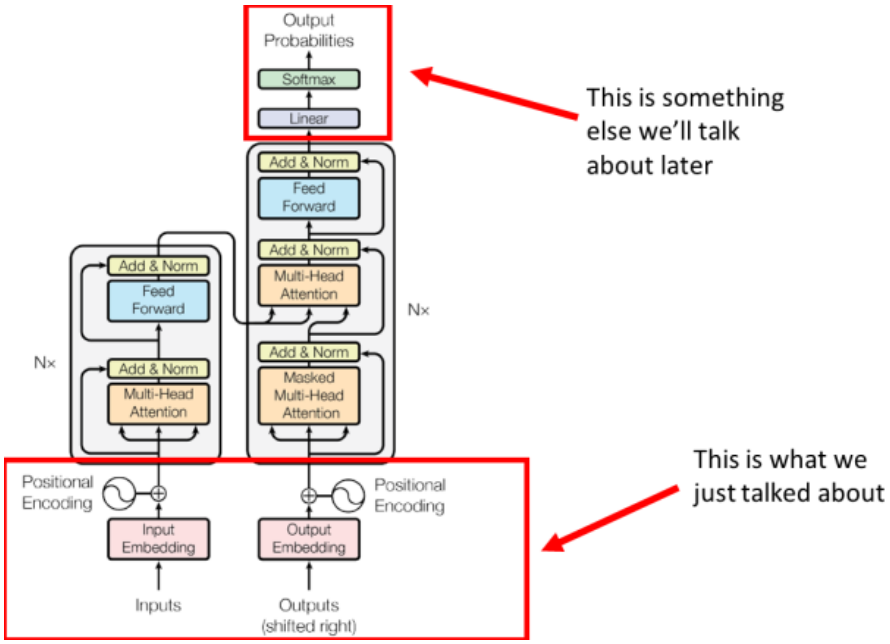
Con esto ya se obtiene la representación de entrada: la oración en inglés "I like trees" convertida en tres vectores (uno por cada palabra):



Cada uno de los vectores en nuestra representación de entrada captura tanto el significado de la palabra como la posición de la palabra en la oración.

Aplicamos exactamente el mismo proceso (embedding de palabras + codificación posicional) para representar la salida, que en este ejemplo es a oración en español "Me gustan los árboles".

Con esto ya cubrimos la parte inferior de la figura 1 del artículo Transformer: cómo se representan las oraciones de entrada y salida antes de pasar al resto del modelo (no debe confundirse con las "output probabilities" que aparecen en la parte superior, que corresponden a otra etapa):



La forma del tensor de entrada (y también del tensor de salida) después del embedding y la codificación posicional es [nbatches, L, 512], donde nbatches es el tamaño del lote, L es la longitud de la secuencia (por ejemplo, L = 3 para "I like trees") y 512 es la dimensión del embedding/codificación posicional. Nótese que los lotes suelen agrupar secuencias de longitudes similares.

El codificador

A continuación, el codificador procesa la oración. El codificador tiene esta estructura:

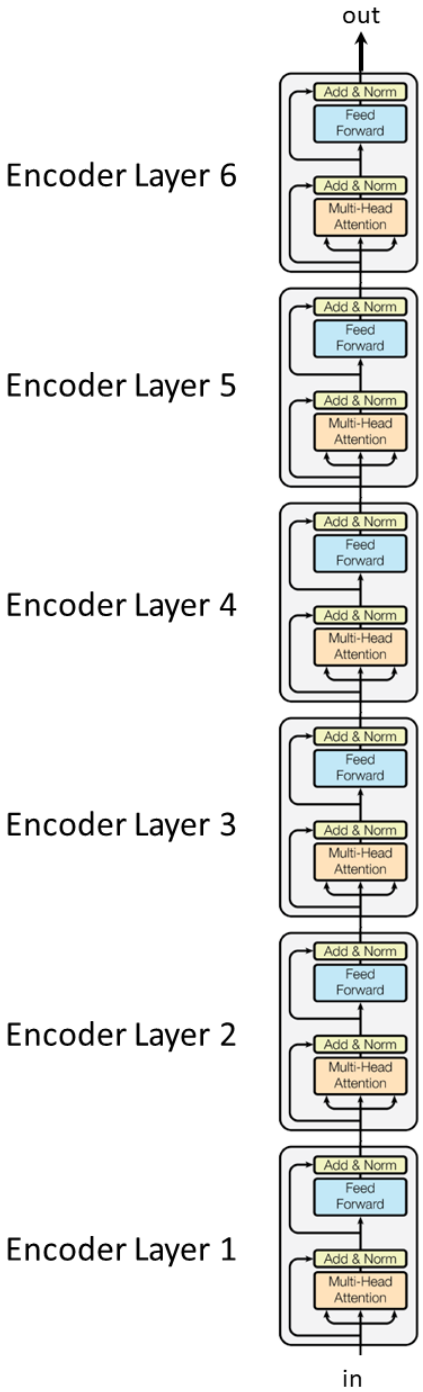


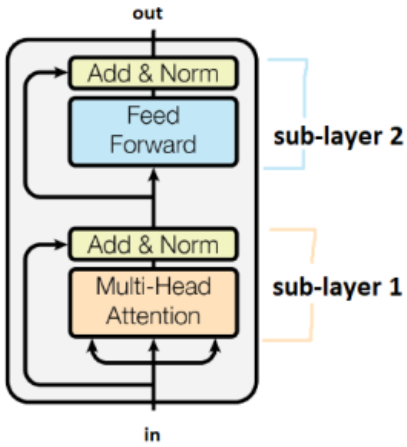
Figura adaptada del artículo original del Transformer desde la figura.

Como puede verse en la figura, el codificador está compuesto por N = 6 capas idénticas, apiladas una sobre otra.

En traducción inglés-español, la entrada del codificador ("in") es la oración en inglés (por ejemplo, "I like trees"), representada como embeddings de palabras más codificaciones posicionales. La salida ("out") es una nueva representación vectorial de esa oración.

Cada una de las seis capas del codificador contiene dos subcapas:

- La primera subcapa es un mecanismo de autoatención de múltiples cabeceras.
- La segunda subcapa es una red feed-forward totalmente conectada, aplicada de forma independiente a cada posición.



A continuación se explica qué hace cada subcapa. Primero, veamos un fragmento del código de The Annotated Transformer que muestra cómo se construye el codificador:

```
class Encoder(nn.Module):
    "Core encoder is a stack of N layers"
    def __init__(self, layer, N):
        super(Encoder, self).__init__()
        self.layers = clones(layer, N)
        self.norm = LayerNorm(layer.size)

    def forward(self, x, mask):
        "Pass the input (and mask) through each layer in turn."
        for layer in self.layers:
            x = layer(x, mask)
        return self.norm(x)


class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
```

- La clase "Encoder" recibe un <layer> y lo apila <N> veces. El <layer> que se apila es una instancia de la clase "EncoderLayer".
- La clase "EncoderLayer" se inicializa con <size>, <self_attn>, <feed_forward> y <dropout>:
    - <size> es d_model, que vale 512 en el modelo base.
    - <self_attn> es una instancia de la clase "MultiHeadedAttention". Corresponde a la subcapa 1.
    - <feed_forward> es una instancia de la clase "PositionwiseFeedForward". Corresponde a la subcapa 2.
    - <dropout> es la tasa de dropout (por ejemplo, 0.1).

Subcapa 1 del codificador: mecanismo de atención de múltiples cabeceras

Comprender la atención de múltiples cabeceras es clave para entender el Transformer. Este mecanismo se usa tanto en el codificador como en el decodificador. Primero se presenta una explicación de alto nivel basada en las ecuaciones del artículo, y luego se revisan los detalles del código.

Resumen conceptual de la atención: claves, consultas y valores

Llamemos x a la entrada del mecanismo de atención. Al inicio del codificador, x es la representación inicial de la oración. En capas posteriores, x es la salida del EncoderLayer anterior; por ejemplo, EncoderLayer3 recibe la salida de EncoderLayer2.

A partir de x se calculan claves, consultas y valores mediante capas lineales distintas:

- clave = lineal_k (x)
- consulta = lineal_q (x)
- valor = linear_v (x)

En una capa del codificador, linear_k, linear_q y linear_v son capas distintas (de 512 a 512) con pesos diferentes que se aprenden por separado. Si compartieran pesos, claves, consultas y valores serían idénticos y no tendría sentido diferenciarlos.

Una vez que tenemos nuestras claves (K), consultas (Q) y valores (V), calculamos la atención de la siguiente manera:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Esta es la ecuación (1) del artículo Transformer. Veamos qué está ocurriendo:

Primero se calcula el producto punto entre consultas y claves. Si procesamos varias consultas y claves a la vez, ese cálculo puede escribirse como una multiplicación de matrices:

$$QK^T$$

Aquí, Q es una matriz (o pila) de consultas q y K es una matriz (o pila) de claves k.

Después de tomar el producto punto, lo dividimos por la raíz cuadrada de d_k:

$$\frac{QK^T}{\sqrt{d_k}}$$

¿Por qué se divide por sqrt(d_k)? Los autores explican que este escalado evita que los productos punto crezcan demasiado cuando aumenta d_k (la longitud de los vectores).

Ejemplo: el producto escalar de [2,2] con [2,2] es 8, mientras que el de [2,2,2,2,2] con [2,2,2,2,2] es 20. No queremos que el valor crezca solo por aumentar la dimensión del vector, así que dividimos por sqrt(d_k). Si los puntajes son demasiado grandes, el softmax entra en regiones con gradientes muy pequeños.

El siguiente paso es aplicar la función softmax, que convierte los puntajes en valores dentro del rango (0, 1):

$$\text{softmax}(\frac{QK^T}{\sqrt{d_k}})$$

¿Qué tenemos hasta aquí? Un conjunto de pesos de atención entre 0 y 1, calculados como softmax(QK^T / sqrt(d_k)).

El último paso es calcular una suma ponderada de los valores V usando los pesos de atención obtenidos:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Descripción detallada de la atención de múltiples cabeceras con código

Esta sección se apoya en el código de https://nlp.seas.harvard.edu/2018/04/03/attention.html porque revisar el código ayuda mucho a entender qué está ocurriendo.

Primero se muestran los fragmentos de código relevantes con algunas anotaciones. No te preocupes por leerlas todas ahora; los puntos clave se explican más abajo. (Según el navegador, quizá necesites abrir la imagen en otra ventana para ver el texto con suficiente tamaño).

En The Annotated Transformer, la atención de múltiples cabeceras se implementa mediante la clase MultiHeadedAttention:

```
class MultiHeadedAttention(nn.Module):
    def __init__(self, h, d_model, dropout=0.1):
        "Take in model size and number of heads."
        super(MultiHeadedAttention, self).__init__()
        assert d_model % h == 0
        # We assume d_v always equals d_k
        self.d_k = d_model // h
        self.h = h
        self.linears = clones(nn.Linear(d_model, d_model), 4)
        self.attn = None
        self.dropout = nn.Dropout(p=dropout)

    def forward(self, query, key, value, mask=None):
        "Implements Figure 2"
        if mask is not None:
            # Same mask applied to all h heads.
            mask = mask.unsqueeze(1)
        nbatches = query.size(0)

        # 1) Do all the linear projections in batch from d_model => h x d_k
        query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
             for l, x in zip(self.linears, (query, key, value))]

        # 2) Apply attention on all the projected vectors in batch.
        x, self.attn = attention(query, key, value, mask=mask,
                                 dropout=self.dropout)

        # 3) "Concat" using a view and apply a final linear.
        x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
        return self.linears[-1](x)
```

Una instancia de esta clase se inicializa con:

- <h> = 8, el número de cabeceras. El modelo base Transformer usa 8 cabeceras.
- <d_model> = 512
- <dropout> = tasa de dropout = 0.1

La dimensión de las claves, d_k, se calcula como d_model/h. Entonces, en este caso, d_k = 512/8 = 64.

Veamos con más detalle la función forward() de MultiHeadedAttention:

```
def forward(self, query, key, value, mask=None):
    "Implements Figure 2"
    if mask is not None:
        # Same mask applied to all h heads.
        mask = mask.unsqueeze(1)
    nbatches = query.size(0)

    # 1) Do all the linear projections in batch from d_model => h x d_k
    query, key, value = [l(x).view(nbatches, -1, self.h, self.d_k).transpose(1, 2)
         for l, x in zip(self.linears, (query, key, value))]

    # 2) Apply attention on all the projected vectors in batch.
    x, self.attn = attention(query, key, value, mask=mask,
                             dropout=self.dropout)

    # 3) "Concat" using a view and apply a final linear.
    x = x.transpose(1, 2).contiguous().view(nbatches, -1, self.h * self.d_k)
    return self.linears[-1](x)
```

La entrada de forward() es query, key, value y mask. Por ahora podemos ignorar mask. ¿De dónde salen query, key y value? En el EncoderLayer salen de la misma x, repetida tres veces (ver resaltado amarillo):

```
class EncoderLayer(nn.Module):
    "Encoder is made up of self-attn and feed forward (defined below)"
    def __init__(self, size, self_attn, feed_forward, dropout):
        super(EncoderLayer, self).__init__()
        self.self_attn = self_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 2)
        self.size = size

    def forward(self, x, mask):
        "Follow Figure 1 (left) for connections."
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, mask))
        return self.sublayer[1](x, self.feed_forward)
|
```

x proviene del EncoderLayer anterior o, si estamos en EncoderLayer1, de la representación inicial de la oración. (En la clase EncoderLayer, self.self_attn es una instancia de MultiHeadedAttention).

Dentro de MultiHeadedAttention, se toman las consultas, claves y valores "antiguos" (derivados de x) y se proyectan para producir nuevas consultas, claves y valores distintos entre sí.

Nótese que la forma de la entrada de "consulta" es [nbatches, L, 512] donde nbatches es el tamaño del lote, L es la longitud de la secuencia y 512 es d_model. Las entradas "clave" y "valor" también tienen forma [nbatches, L, 512].

Paso 1) En la función forward() de MultiHeadedAttention: "realizar en lote todas las proyecciones lineales desde d_model hasta h x d_k".

- Haremos tres proyecciones lineales diferentes en el mismo tensor de forma [nbatches, L, 512] para obtener nuevas consultas, claves y valores, cada uno de forma [nbatches, L, 512]. (La forma no ha cambiado ya que la capa lineal es 512 -> 512).
- Luego, esa salida se reorganiza en 8 cabeceras. Por ejemplo, las consultas pasan de [nbatches, L, 512] a [nbatches, L, 8, 64] mediante view(), donde h = 8 y d_k = 64.
- Finalmente, intercambiaremos las dimensiones 1 y 2 usando la transposición para obtener la forma [nbatches, 8, L, 64].

Paso 2) En el código: "aplicar atención a todos los vectores proyectados en lote".

- La línea específica es x, self.attn = attention(query, key, value, mask=mask, dropout=self.dropout)
- Aquí está la ecuación que estamos implementando usando la función attention ():

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

- Y aquí la implementación de la función attention():

```
def attention(query, key, value, mask=None, dropout=None):
    "Compute 'Scaled Dot Product Attention'"
    d_k = query.size(-1)
    scores = torch.matmul(query, key.transpose(-2, -1))/ math.sqrt(d_k)
    if mask is not None:
        scores = scores.masked_fill(mask == 0, -1e9)
    p_attn = F.softmax(scores, dim = -1)
    if dropout is not None:
        p_attn = dropout(p_attn)
    return torch.matmul(p_attn, value), p_attn
```

Para calcular los puntajes, primero se multiplica la consulta [nbatches, 8, L, 64] por la clave transpuesta [nbatches, 8, 64, L]. Esto corresponde a QK^T en la ecuación. La forma resultante es [nbatches, 8, L, L].

Luego se calculan los pesos de atención p_attn aplicando softmax a los puntajes; si corresponde, también se aplica dropout. p_attn corresponde a softmax(QK^T / sqrt(d_k)) y mantiene la forma [nbatches, 8, L, L].

Finalmente, se multiplica p_attn [nbatches, 8, L, L] por los valores [nbatches, 8, L, 64]. El resultado tiene forma [nbatches, 8, L, 64] y se devuelve junto con los propios pesos de atención p_attn.

Nótese que tanto en la entrada como en la salida de la función de atención aparece la dimensión de 8 cabeceras ([nbatches, 8, L, 64]). Se realiza un cálculo por cada cabecera. Esto constituye la atención de múltiples cabeceras: múltiples subespacios de representación para analizar la misma oración desde perspectivas distintas.

Paso 3) (ya de vuelta en la clase MultiHeadedAttention) consiste en concatenar con view() y luego aplicar una capa lineal final.

- Las líneas específicas del Paso 3 son:

  x = x.transpose (1, 2).contiguous (). view (nbatches, -1, self.h * self.d_k)

  return self.linears [-1] (x)

- x es lo que devuelve la función de atención: nuestra representación de ocho cabeceras [nbatches, 8, L, 64].

- Después se transpone para obtener [nbatches, L, 8, 64] y se reorganiza con view() a [nbatches, L, 8 x 64] = [nbatches, L, 512]. Esa operación equivale a concatenar las 8 cabeceras.
- Por último, se aplica la última capa lineal (self.linears[-1]), que va de 512 a 512. En PyTorch, una capa lineal aplicada a un tensor multidimensional opera sobre la última dimensión, así que la salida sigue teniendo forma [nbatches, L, 512].
- Esa forma [nbatches, L, 512] es justo la que se necesita para alimentar otra capa MultiHeadedAttention. Antes de eso, falta revisar la subcapa 2 del codificador, pero primero conviene mirar la figura 2 del artículo.

Aquí está la figura 2 del artículo Transformer:

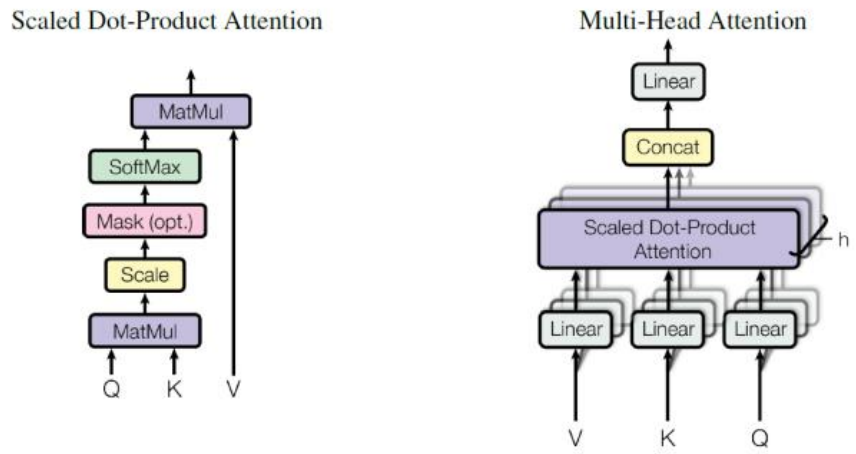


Figura 2. Tomada del artículo original del Transformer.

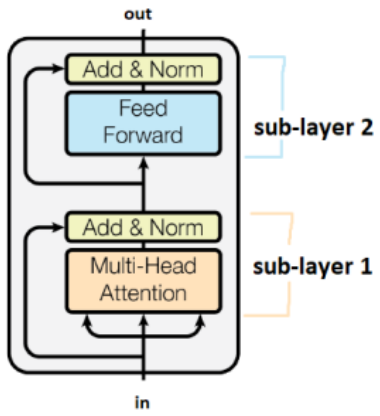
A la izquierda, en "Scaled Dot-Product Attention", se muestra visualmente lo que calcula la función attention(): softmax(QK^T / sqrt(d_k))V. Se llama "scaled" porque divide por sqrt(d_k) y "dot-product" porque QK^T representa productos punto entre consultas y claves.

A la derecha, en "Multi-Head Attention", se muestra lo que hace la clase MultiHeadedAttention. A esta altura ya pueden identificarse claramente las partes de la figura:

- En la parte inferior aparecen V, K y Q originales, que corresponden a la salida x de un EncoderLayer anterior (o a la representación inicial de la oración en EncoderLayer1).
- Luego se aplica una capa lineal (las cajas "Linear") para obtener nuevas versiones proyectadas de V, K y Q.
- Esas versiones proyectadas se envían al bloque de atención escalada con 8 cabeceras (la función attention()).
- Finalmente, se concatena el resultado de las 8 cabeceras y se aplica una última capa lineal para producir la salida de la atención de múltiples cabeceras.

Subcapa 2 del codificador: red feed-forward totalmente conectada y por posición

¡Ya casi hemos terminado de entender un EncoderLayer completo! Recuerda que esta es la estructura básica de un solo EncoderLayer (modificado a partir del artículo Transformer figura 1):



Ya se revisó la subcapa 1 (atención de múltiples cabeceras). A continuación se describe la subcapa 2, la red feed-forward.

La subcapa 2 es más sencilla que la subcapa 1: es una red neuronal feed-forward. Su expresión es:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

En otras palabras, se aplica una capa totalmente conectada con pesos W1 y sesgo b1, luego una no linealidad ReLU (máximo con cero) y después una segunda capa totalmente conectada con pesos W2 y sesgo b2.

A continuación se muestra el fragmento de código correspondiente de The Annotated Transformer:

```
class PositionwiseFeedForward(nn.Module):
    "Implements FFN equation."
    def __init__(self, d_model, d_ff, dropout=0.1):
        super(PositionwiseFeedForward, self).__init__()
        self.w_1 = nn.Linear(d_model, d_ff)
        self.w_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        return self.w_2(self.dropout(F.relu(self.w_1(x))))
```

En resumen, el codificador contiene 6 EncoderLayers, y cada EncoderLayer tiene 2 subcapas: (1) atención de múltiples cabeceras y (2) una red feed-forward.

El decodificador

Ahora que el codificador está claro, el decodificador es más fácil de entender porque su estructura es muy similar. Aquí reaparece la figura 1 con anotaciones adicionales:

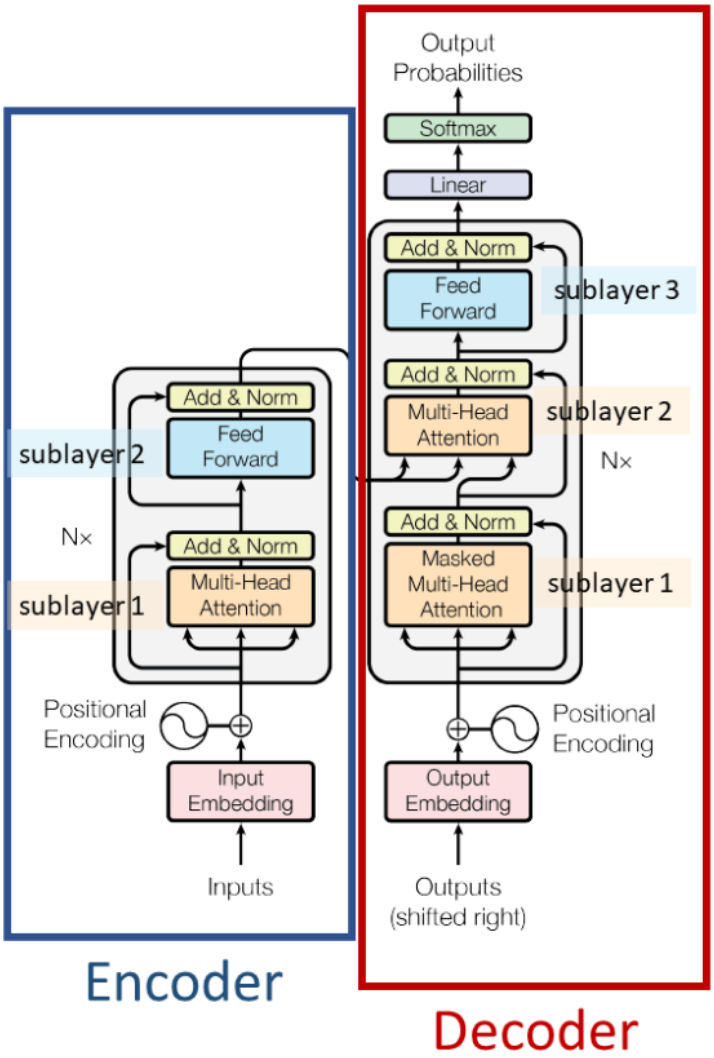


Figura adaptada del artículo original del Transformer.

Hay tres diferencias principales entre el decodificador y el codificador:

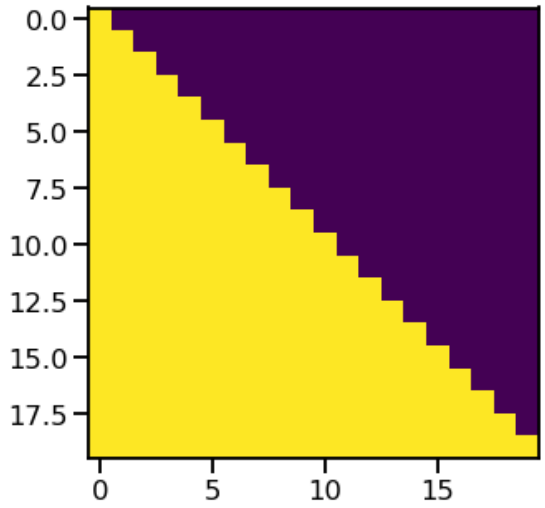
- La subcapa 1 del decodificador usa atención de múltiples cabeceras enmascarada para evitar "ver el futuro".
- El decodificador tiene una subcapa adicional, etiquetada como "subcapa 2" en la figura anterior. Esta subcapa es "codificador-decodificador de atención de múltiples cabeceras".
- Al final del decodificador se aplica una capa lineal y luego la función softmax para producir las probabilidades de salida de la siguiente palabra.

A continuación se describe cada una de estas piezas.

Subcapa 1 del decodificador: atención de múltiples cabeceras enmascarada

El objetivo del enmascaramiento en la atención de múltiples cabeceras es impedir que el decodificador "vea el futuro"; es decir, evitar que utilice información de tokens que aún no debería conocer.

La máscara consta de unos y ceros:



Las líneas de código en la función attention() que usa la máscara están aquí:

```
if mask is not None:
    scores = scores.masked_fill(mask == 0, -1e9)
```

La función masked_fill(mask, value) rellena posiciones del tensor con value donde la máscara lo indica. En este caso se usa para anular (mediante un valor muy negativo antes del softmax) los puntajes que corresponden a palabras futuras.

Como indican los autores, "modificamos la subcapa de autoatención en la pila del decodificador para evitar que una posición atienda a posiciones posteriores". Este enmascaramiento, junto con el desplazamiento de los embeddings de salida, garantiza que la predicción en la posición i dependa solo de posiciones menores que i.

Subcapa 2 del decodificador: atención codificador-decodificador de múltiples cabeceras

A continuación se muestra el código de un DecoderLayer:

```
class DecoderLayer(nn.Module):
    "Decoder is made of self-attn, src-attn, and feed forward (defined below)"
    def __init__(self, size, self_attn, src_attn, feed_forward, dropout):
        super(DecoderLayer, self).__init__()
        self.size = size
        self.self_attn = self_attn
        self.src_attn = src_attn
        self.feed_forward = feed_forward
        self.sublayer = clones(SublayerConnection(size, dropout), 3)

    def forward(self, x, memory, src_mask, tgt_mask):
        "Follow Figure 1 (right) for connections."
        m = memory
        x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask))
        x = self.sublayer[1](x, lambda x: self.src_attn(x, m, m, src_mask))
        return self.sublayer[2](x, self.feed_forward)
```

La línea subrayada en amarillo define la "atención del codificador-decodificador":

x = self.sublayer[1] (x, lambda x: self.src_attn(x, m, m, src_mask))

self.src_attn es una instancia de MultiHeadedAttention. Sus entradas son query = x, key = m, value = m y mask = src_mask. Aquí, x proviene del DecoderLayer anterior, mientras que m ("memory") proviene de la salida del codificador (por ejemplo, EncoderLayer6).

(La línea anterior, x = self.sublayer[0](x, lambda x: self.self_attn(x, x, x, tgt_mask)), define la autoatención de la subcapa 1 del decodificador. Funciona igual que la autoatención del codificador, con el paso adicional de enmascaramiento).

Resumen completo de atención en el transformer

Ya se revisaron los tres tipos de atención del Transformer. A continuación se presenta el resumen de los autores sobre el uso de la atención en el modelo:

3.2.3 Applications of Attention in our Model

The Transformer uses multi-head attention in three different ways:

- In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. This mimics the typical encoder-decoder attention mechanisms in sequence-to-sequence models such as [31, 2, 8].
- The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.
- Similarly, self-attention layers in the decoder allow each position in the decoder to attend to all positions in the decoder up to and including that position. We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. We implement this inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections. See Figure 2.

Salida final del decodificador: capa lineal y Softmax para producir probabilidades de salida

Al final de la pila del decodificador, su salida pasa por una capa lineal seguida de la función softmax para predecir el siguiente token.

El codificador se ejecuta una vez para obtener una representación de la oración de entrada en inglés ("I like trees"). Luego el decodificador se ejecuta varias veces para producir, paso a paso, la traducción al español ("Me gustan los árboles").

La última capa lineal expande la salida del decodificador a un vector cuya longitud es igual al tamaño del vocabulario. Después, el softmax permite escoger (por ejemplo, con decodificación codiciosa) la palabra con mayor probabilidad en el vocabulario de salida.

Después de entrenar la red (es decir, durante inferencia), el proceso típico es el siguiente.

Nota: la salida del codificador se calcula una sola vez y luego se reutiliza en todos los pasos del decodificador.

1. Se alimenta el decodificador con la salida del codificador (que representa "I like trees") y un token especial de inicio de secuencia (por ejemplo, </s>) en la entrada de salida. El decodificador debería predecir "Me".
2. Se vuelve a ejecutar el decodificador con la misma salida del codificador y con la secuencia "</s> Me". Ahora debería predecir "gustan".
3. Se ejecuta con "</s> Me gustan" y debería predecir "los".
4. Se ejecuta con "</s> Me gustan los" y debería predecir "árboles".
5. Se ejecuta con "</s> Me gustan los árboles" y debería predecir el token de fin de secuencia (por ejemplo, </eos>).
6. Como el decodificador ya produjo el token de fin de secuencia, la traducción de la oración termina aquí.

¿Qué ocurre durante el entrenamiento? Al inicio, el decodificador puede producir predicciones incorrectas. Para evitar reutilizar esas predicciones erróneas como entrada del siguiente paso, durante el entrenamiento se emplea teacher forcing (forzado del profesor).

Con teacher forcing, se aprovecha que conocemos la traducción correcta y se alimenta al decodificador con los tokens correctos previos. No obstante, no se le da la palabra actual que debe predecir; por ejemplo, para predecir "árboles" se le puede dar "</s> Me gustan los". Esto se implementa mediante:

- el enmascaramiento (que impide ver palabras futuras; por ejemplo, no se usa "los árboles" cuando se debe predecir "gustan"), y
- un desplazamiento hacia la derecha para que el token "presente" tampoco se use como entrada al momento de predecirlo.

Luego, la pérdida se calcula comparando la distribución de probabilidad producida por el decodificador (por ejemplo, [0.01, 0.01, 0.02, 0.70, 0.20, 0.01, 0.05]) con la distribución objetivo (por ejemplo, un vector one-hot con el "1" en la posición de "árboles").

El enfoque descrito arriba (elegir la palabra de mayor probabilidad en cada paso) se llama decodificación codiciosa. Una alternativa común es beam search, que mantiene varias hipótesis de salida en paralelo.

A continuación se muestra la clase PyTorch Generator utilizada para la capa lineal final y el softmax:

```
class Generator(nn.Module):
    "Define standard linear + softmax generation step."
    def __init__(self, d_model, vocab):
        super(Generator, self).__init__()
        self.proj = nn.Linear(d_model, vocab)

    def forward(self, x):
        return F.log_softmax(self.proj(x), dim=-1)
```

Componentes adicionales

Hasta este punto ya se han revisado los componentes principales del modelo Transformer. A continuación, se mencionan conceptos adicionales que también forman parte de la arquitectura y del entrenamiento:

Dropout: se usa en varios puntos del Transformer. Consiste en desactivar aleatoriamente un subconjunto de neuronas durante el entrenamiento para reducir el sobreajuste.

Conexión residual y normalización de capa: cada subcapa del codificador y del decodificador está rodeada por una conexión residual, seguida de normalización de capa. Para la conexion residual si calculamos una función f(x), una conexión residual produce f(x) + x. Es decir, se suma la entrada original a la salida de la subcapa. La normalización de capa normaliza las activaciones a través de las características de cada ejemplo (a diferencia de la normalización por lotes, que usa estadísticas del lote).

En el siguiente diagrama de un EncoderLayer, las partes relevantes están marcadas en rojo: la flecha y el bloque "Add and Norm", que representan la conexión residual y la normalización de capa:

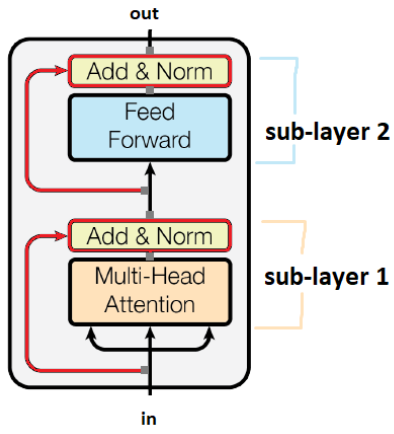


Figura modificada a partir del artículo original del Transformer.

Cita del artículo original: "El resultado de cada subcapa es LayerNorm(x + Sublayer(x)), donde Sublayer(x) es la función implementada por la propia subcapa. Para facilitar estas conexiones residuales, todas las subcapas del modelo, así como las capas de embeddings, producen salidas de dimensión d_model = 512".

En The Annotated Transformer (implementación de referencia en PyTorch), conviene revisar la clase "LayerNorm" y también la clase "SublayerConnection", que aplica LayerNorm, dropout y la conexión residual.

Optimizador Adam: el Transformer se entrena con Adam. Los autores proponen, además, una programación específica de la tasa de aprendizaje: primero aumenta linealmente durante un número de pasos (warmup) y luego disminuye de forma proporcional a la raíz cuadrada inversa del número de paso. Esta estrategia se implementa en la clase "NoamOpt" de The Annotated Transformer.

Suavizado de etiquetas: los autores también aplican label smoothing. Esta técnica toma las etiquetas one-hot y distribuye una pequeña parte de la probabilidad entre clases distintas de la correcta, en lugar de asignar toda la masa a un único "1". En The Annotated Transformer puede revisarse en la clase "LabelSmoothing".

Resumen de la arquitectura

- El Transformer consta de un codificador y un decodificador.
- La oración de entrada (por ejemplo, "I like trees") y la de salida (por ejemplo, "Me gustan los árboles") se representan con embeddings de palabras más codificaciones posicionales.
- El codificador está compuesto por 6 EncoderLayers. El decodificador también está compuesto por 6 DecoderLayers.
- Cada EncoderLayer tiene dos subcapas: autoatención de múltiples cabeceras y una red feed-forward.
- Cada DecoderLayer tiene tres subcapas: autoatención de múltiples cabeceras enmascarada, atención codificador-decodificador de múltiples cabeceras y una red feed-forward.
- Al final del decodificador, se aplica una capa lineal y un softmax para predecir la siguiente palabra.
- El codificador se ejecuta una vez. El decodificador se ejecuta varias veces para producir una palabra en cada paso.