Transformers

Transformers Reimplementation

Mario Chacón Falcón

24/05/2024

1.. Introducción

El objetivo de este trabajo es realizar una implementación desde cero del modelo introducido en el conocido artículo [2], los transformers. La memoria constará de dos partes principales. En primer lugar, una sección en la que se repasan los distintos módulos del transformer y se explica la implementación de cada uno de ellos. Entre estos destaca la implementación de la atención multicabeza, el *positional encoding* y del *decoder*. Tras esto, se aplicarán las clases definidas para construir y entrenar 2 transformers, uno para análisis de sentimientos, del dataset IMBD y uno para la predicción de audio, el dataset SHD.

2.. Transformer: Arquitectura del modelo e implementación

Desde su publicación, los transformers han ido tomando cada vez más protagonismo, siendo hoy en día los modelos más usados para procesamiento de lenguaje. Esto es gracias a su habilidad para entender las diferencias semánticas (desambiguar) en función del orden y la relación entre las palabras. Como es usual en tareas de transducción, el transformer consta de dos módulos, el codificador (*encoder*) y el decodificador(*decoder*). A grandes rasgos, el codificador recibe la secuencia de entrada (donde cada x_i representa un token), $(x_1, \dots, x_n)$, a una representación latente, $(z_1, \dots, z_n)$, que será luego traducida por el decodificador para devolver la salida (token a token). En la figura 1 se muestra la arquitectura de un módulo transformer. Introduzcamos cada parte del modelo junto con su código.

2.1. Prepocesado

Los modelos de *Machine Learning* (ML) reciben como datos de entrada vectores numéricos. Para que un modelo de ML resuelva tareas de lenguaje natural, es necesario trabajar con una representación vectorial de las palabras (*embedding*). Estas representaciones vienen ya preentrenadas en las librerías de python, y solo hay que indicar

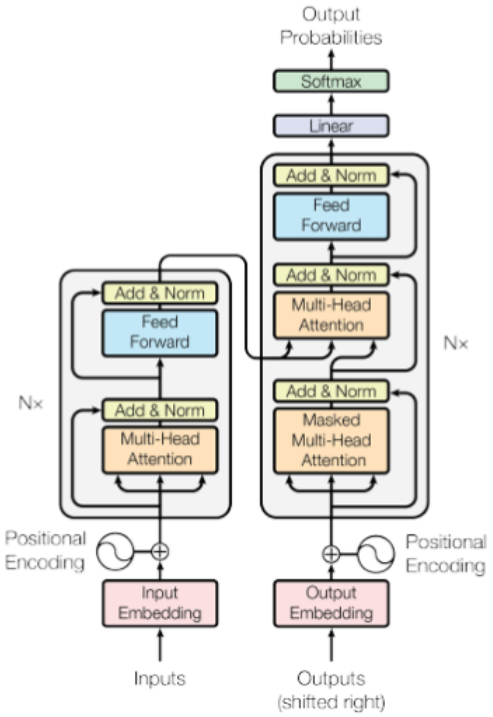

Figura 1: Arquitectura de un transformer [2]

el tamaño del vocabulario (el número de tokens que queremos codificar) y el tamño de representación de nuestros vectores (*embeddingdim*). La capa de embedding en Pytorch se introduce con el siguiente comando:

```
import torch.nn as nn
embeding = nn.Embedding(num_embeddings=vocab_size,
                        embedding_dim=embed_dim)
```

Además de vectorizar los tokens, al recibirlos todos en paralelo, es necesario especificar de alguna forma el orden de los mismos, ya que el orden puede afectar al significado. De esta forma, se introduce el *Positional Encoding*, que, dada una longitud de secuencia máxima, l, y el tamaño del *embedding*, d, generan una l vectores de dimensión d que se sumarán a las representaciones para distinguir las posiciones. En concreto, en este trabajo, se define el positional encoding original del artículo [2], aquel dado por senos en la posición par y por cosenos en la posición impar. Concretamente, si d denota la dimensión del modelo (divisible por 2), t la posición que se trata de codificar y $p_t \in \mathbb{R}^d$ su codificación, entonces:

$$p_t^{(i)} = f(t)^{(i)} = \begin{cases} sin(w_k t) \text{ si } i = 2k \\ cos(w_k t) \text{ si } i = 2k+1 \end{cases}$$

Con $k \in \mathbb{N}$ y donde $w_k = \frac{1}{10000^{2k/d}}$. Esto en código python para la librería PyTorch se

define como la siguiente clase:

```
    class PositionalEncoding(nn.Module):
    def __init__(self,embed_dim,max_len):
        super(PositionalEncoding,self).__init__()
        posenc = torch.zeros(max_len,embed_dim)
        #unsqueeze to be broadcastable when multiplying with w_k
        #dim((pos*w_k))=max_len x model_dim
        position = torch.arange(0,max_len,dtype=torch.float).unsqueeze(1)
        w_k = torch.arange(0,embed_dim,2).float()*(1/(10000**(1/embed_dim
    )))

        posenc[:,0::2] = torch.sin(position*w_k) #los pares son el seno
        posenc[:,1::2] = torch.cos(position*w_k) #los impares son el
    coseno

        #Para que no se tenga en cuenta como parametro entrenable
        self.register_buffer('posenc',posenc.unsqueeze(0))

    def forward(self,x):
        #Se usa x.size() porque las entradas no tienen que ser todas de
    max_len.
        return x + self.posenc[:,:x.size(1)]
```

2.2. Codificador

Una vez ya se han preprocesado los datos se entra al modelo transformer. En primer lugar, se estudiará la capa del codificador. Esta capa consiste de una capa de atención (cuyo papel es desambiguar el significado y la importancia de cada uno de los tokens respecto al resto) seguido de normalizaciones y capas totalmente conectadas. Es interesante destacar que el mecanismo de atención consiste solo en operaciones lineales y que, sin la red totalmente conectada, no tendríamos suficiente expresividad en el modelo. Repasemos entonces en qué consiste el mecanismo de atención, el de atención multicabeza y cómo se paraleliza en la computación. El código python asociado a esto viene dado por:

```
    class MultiHeadAttention(nn.Module):
    def __init__(self,d_k,embed_dim,num_heads,d_v=None):
        super(MultiHeadAttention,self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.d_k = d_k
        self.key_query_dimension = d_k*num_heads
        if d_v is not None:
            self.d_v = d_v
            self.value_dimension = d_v*num_heads
        else:
            self.value_dimension=self.key_query_dimension
            self._d_v = d_k


```

```
16        self.W_q = nn.Linear(embed_dim,self.key_query_dimension) # Query
    matrix of all heads
17        self.W_k = nn.Linear(embed_dim,self.key_query_dimension) # Key
    matrix of all heads
18        self.W_v = nn.Linear(embed_dim,self.value_dimension) # Value
    matrix  of all heads
19        self.W_o = nn.Linear(self.value_dimension,embed_dim) # Matrix to
    resize multihead output
20    def scaled_dot_product_attention(self,Q,K,V,mask=None):
21        # Primero calculamos la atencion de una palabra a otra:
22        att_scores = torch.matmul(Q,K.transpose(-2,-1))/math.sqrt(self.
    d_k)
23
24        # Para cuando se haga el Decoder se mete una mascara
25        if mask is not None:
26            #Aquellos valores de att_scores que coincidan con mask == 0 (
    broadcastable) son sustituidos por -1e9.
27            # De esta forma al aplicar softmax, al ser valores muy chicos
     seran casi cero y no se les prestara atencion.
28            att_scores = att_scores.masked_fill(mask==0,-1e9)
29        att_prob = torch.softmax(att_scores,dim=-1)
30
31        #Finalmente se obtienen los vectores de salida de la atencion
    multicabeza al multiplicar por la matriz V
32        output = torch.matmul(att_prob,V)
33        return output
34    def split_heads(self,x):
35        #Para poder hacer multihead attention tenemos que dividir las
    entradas del scaled_dot_attention_product. De esta forma,
36        #se puede paralelizar la computacion (como dice en el paper
    original).
37        b_size,seq_length,_ = x.size()
38        return x.view(b_size,seq_length,self.num_heads,-1).transpose(1,2)
39    def concat_heads(self,x):
40        b_size,_,seq_length,d_k = x.size()
41        return x.transpose(1,2).contiguous().view(b_size,seq_length,self.
    value_dimension)
42    def forward(self,Q,K,V,mask=None):
43        #Multiplicacion por las matrices de query,value y key. Luego, las
     salidas son proyectadas para llevar a cabo atencion multicabeza.
44        Q = self.split_heads(self.W_q(Q))
45        K = self.split_heads(self.W_k(K))
46        V = self.split_heads(self.W_v(V))
47        #Atencion multicabeza:
48        att_output = self.scaled_dot_product_attention(Q,K,V,mask)
49        output = self.W_o(self.concat_heads(att_output))
50        return output
```

Estudiémoslo paso a paso:

Mecanismo de atención

Sean $(x_1, \ldots, x_l)$ los vectores de longitud e de entrada que representan cada token. Esto es, una matriz $X \in \mathcal{M}_{l \times e}$. Para cada vector x_i, la capa de atención se encarga de calcular:

- La atención que le presta x_i a x_j para cada $i, j \in \{1, \dots, l\}$. Esto se calcula con las matrices de query y de key, $W_q, W_k \in \mathcal{M}_{e\times e}$. De esta forma, el valor $(x_i W_q)(x_j W_k)^t \in \mathbb{R}$ representará la atención que le presta x_i a x_j.
- La importancia de la palabra x_i. Esto se calcula con la matriz de value, $W_v \in \mathcal{M}_{e\times e}$.
- El valor total de atención de cada vector, computado como:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^t}{\sqrt{d_k}}\right)V \in \mathcal{M}_{l\times e}$$

donde $Q = XW_q, K = XW_k, V = XW_v$.

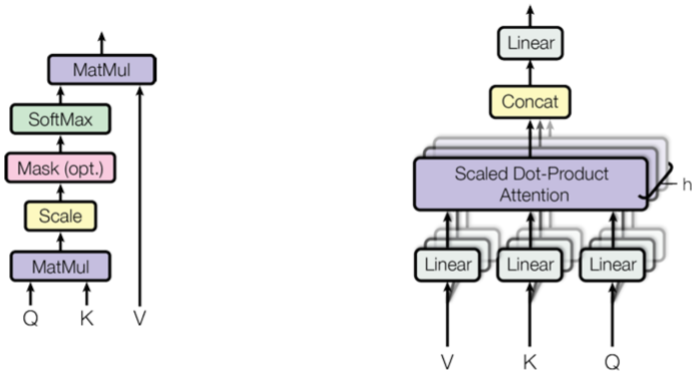

Figura 2: mecanismo de atención (izquierda) y mecanismo de atención multicabeza (derecha), [2]

Mecanismo de atención multicabeza

El mecanismo de atención multicabeza consiste en separar el mecanismo de atención en varias partes de forma que se puedan realizar los cálculos de manera paralela (siendo cada uno una cabeza). Sean $W_q, W_k \in \mathcal{M}_{e\times hd_k}$ y $W_v \in \mathcal{M}_{e\times hd_v}$ las matrices de query, key y value, donde d_k, d_v son hiperparámetros del modelo y h es el número de cabezas. El primer paso del mecanismo de atención multicabeza consiste en el cálculo de las matrices Q, K, V, tal y como se hace en el mecanismo de atención. Tras esto, las matrices Q, K son proyectadas (divididas en h trozos de dimensión $e \times d_k$ ó $e \times d_v$) para calcular h atenciones de forma paralela. Tras haber calculado la atención en cada una de las cabezas, estas son concatenadas y transformadas linealmente por la matriz $W_o \in \mathcal{M}_{hd_v\times e}$, que nos devuelve el tamaño de la salida de la atención a la dimensión $l \times e$, pudiendo así concatenar capas una tras otra. Concretamente, si denotamos por q_i, k_i, v_j a las columnas de Q, K, V, con $1 \le i \le hd_k, 1 \le j \le hd_v$, se tiene que:

$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \dots, \text{head}_h)W_o$$

donde $\text{head}_i = \text{Attention}((q_{((i-1)d_k)+1}, \dots, q_{id_k}), (k_{((i-1)d_k)+1}, \dots, k_{id_k}), (v_{((i-1)d_k)+1}, \dots, v_{id_k}))$

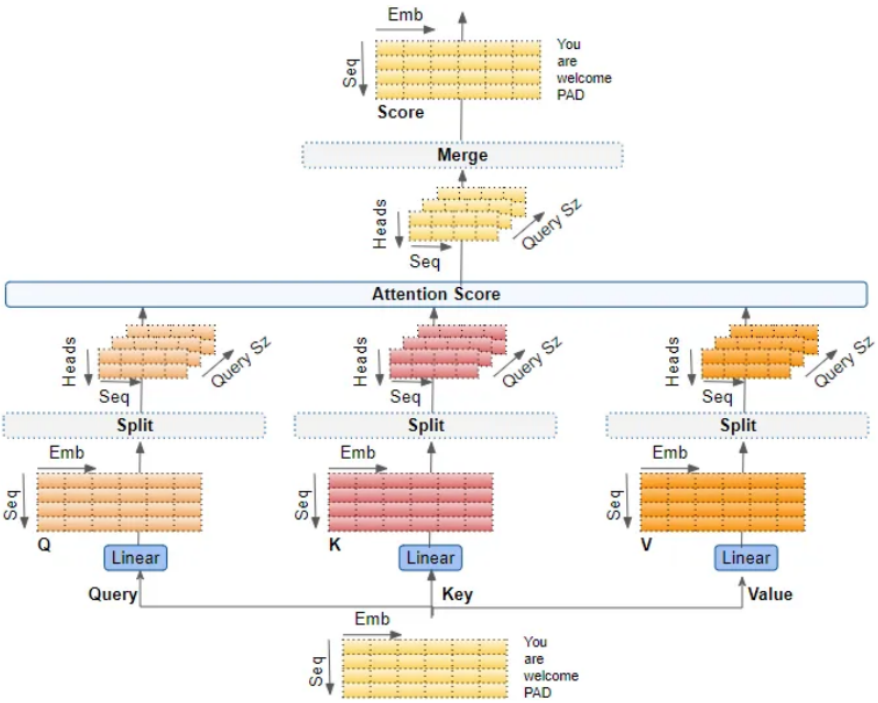

Figura 3: Atención multicabeza paso a paso.

En la figura 3 se muestra un esquema de la atención multicabeza para el caso $d_k * h = e, d_k = d_v$.

Nótese que tomando $d_k = d_v$ con $d_k * h = e$ estamos en las condiciones que se plantean en [2], por lo el código es un planteamiento más general. Se ha optado por esta implementación porque es la que viene por defecto en Tensorflow. La sección del cálculo de la máscara en la capa de atención será explicado en la sección del decodificador.

Capas totalmente conectadas

Tras la capa de atención se incorpora una red totalmente conectada que mantiene la dimensión del embedding y rompe la linealidad de la capa del codificador. La capa totalmente conectada viene dada por:

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

Donde, $W_1 \in \mathcal{M}_{e \times d_f}, W_2 \in \mathcal{M}_{d_f \times e}$, con d_f un hiperparámetro del modelo. El código correspondiente es:

```
    class PositionWiseFeedForward(nn.Module):
    def __init__(self,embed_dim,d_ff):
        super(PositionWiseFeedForward,self).__init__()
        self.ff1 = nn.Linear(embed_dim,d_ff)
        self.ff2 = nn.Linear(d_ff,embed_dim)
        self.relu = nn.ReLU()
```

```
7
8     def forward(self,x):
9         x = self.relu(self.ff1(x))
10        return self.ff2(x)
```

Bloque del codificador

Por último, solo queda juntar todas las capas ya formadas para construir el bloque de codificador, añadiendo las capas residuales, la normalización y el dropout originales del artículo:

```
1     class EncoderLayer(nn.Module):
2     def __init__(self,d_k,embed_dim,num_heads,d_ff,dropout=0.1,d_v=None):
3         super(EncoderLayer,self).__init__()
4         self.self_att = MultiHeadAttention(embed_dim=embed_dim,d_k=d_k,
    d_v=d_v,num_heads=num_heads)
5         self.feed_forward = PositionWiseFeedForward(embed_dim,d_ff)
6         self.norm1 = nn.LayerNorm(embed_dim)
7         self.norm2 = nn.LayerNorm(embed_dim)
8         self.dropout = nn.Dropout(dropout)
9
10    def forward(self,x,mask=None):
11        att_output = self.self_att(x,x,x,mask)
12        x = self.norm1(x+self.dropout(att_output)) #Residual connection +
     normalization
13        ff_out = self.feed_forward(x)
14        x = self.norm2(x+self.dropout(ff_out))
15        return x
```

2.3. Decodificador

En esta sección se creará la clase decodificador, que será casi análoga al codificador, exceptuando el uso de las máscaras para ocultar información. La idea detrás del decodificador es, dada una salida del codificador, z, y, conocida su transducción, y, transucir z de forma que prediga la secuencia de salida. Esta transducción vendrá dada token a token, es decir, que la traducción del siguiente token de la secuencia solo puede depender de los tokens ya predichos y del vector z de codificación. Supongamos que computamos una capa de atención tal y como lo hacíamos en el codificador con la etiqueta de salida. Entonces, la posición (i, j) de la matriz QK^t contendrá la atención que le debe de prestar el token número i al token j. Ahora bien, nótese que si $j > i$ entonces el valor final de la capa de atención asociado a i se verá afectado por un token j que en teoría todavía no se ha predicho. Para lidiar con esto, hemos de modificar la capa de atención de forma que dado un token i, este solo preste atención a aquellos token j que cumplan que $j \leq i$.

Para conseguir esto basta observar que, por definición de la función softmax, se tiene que $\lim_{x\to\infty} \mathrm{softmax}(x) = 0$. De esta forma, como nos interesa que la matriz de atención QK^t tenga su triangular superior toda a 0 (aquellos donde $(qk)_{ij}$ donde $j > i$), basta poner valores muy negativos antes de aplicar la función softmax. Además de la máscara para la atención, también se puede aplicar otra máscara para no tener cuenta

el padding. Este consiste en dado una secuencia de longitud $l_1 < l$, donde l denota la longitud máxima, se aplica el mismo razonamiento que antes con la softmax para poner a cero aquellas columnas j que cumplan que $j > l_1$, pues estarán representando a aquellos tokens que son relleno. Numéricamente, para una tokenización con longitud máxima 5, sea QK^t la siguiente matriz de atención asociada a una y de transducción de longitud 3:

$$\begin{pmatrix} w_{11} & w_{12} & w_{13} & w_{14} & w_{15} \\ w_{21} & w_{22} & w_{23} & w_{24} & w_{25} \\ w_{31} & w_{32} & w_{33} & w_{34} & w_{35} \\ w_{41} & w_{42} & w_{43} & w_{44} & w_{45} \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{pmatrix} \xrightarrow{masking} \begin{pmatrix} w_{11} & -1e9 & -1e9 & -1e9 & -1e9 \\ w_{21} & w_{22} & -1e9 & -1e9 & -1e9 \\ w_{31} & w_{32} & w_{33} & -1e9 & -1e9 \\ w_{41} & w_{42} & w_{43} & w_{44} & -1e9 \\ w_{51} & w_{52} & w_{53} & w_{54} & w_{55} \end{pmatrix} \xrightarrow{paddingmasking}$$

$$\begin{pmatrix} w_{11} & -1e9 & -1e9 & -1e9 & -1e9 \\ w_{21} & w_{22} & -1e9 & -1e9 & -1e9 \\ w_{31} & w_{32} & w_{33} & -1e9 & -1e9 \\ w_{41} & w_{42} & w_{43} & -1e9 & -1e9 \\ w_{51} & w_{52} & w_{53} & -1e9 & -1e9 \end{pmatrix} \xrightarrow{\text{softmax}} \begin{pmatrix} w_{11} & 0 & 0 & 0 & 0 \\ w_{21} & w_{22} & 0 & 0 & 0 \\ w_{31} & w_{32} & w_{33} & 0 & 0 \\ w_{41} & w_{42} & w_{43} & 0 & 0 \\ w_{51} & w_{52} & w_{53} & 0 & 0 \end{pmatrix}$$

Una vez explicado esto, la clase del decodificador es análoga a la clase definida para el codificador:

```
class DecoderLayer(nn.Module):
    def __init__(self,d_k,embed_dim,num_heads,d_ff,dropout=0.1,d_v=None):
        super(DecoderLayer,self).__init__()
        self.self_att = MultiHeadAttention(embed_dim=embed_dim,d_k=d_k,
    num_heads=num_heads,d_v=d_v)
        self.self_enc_att = MultiHeadAttention(embed_dim=embed_dim,d_k=
    d_k,num_heads=num_heads,d_v=d_v)
        self.feed_forward = PositionWiseFeedForward(embed_dim=embed_dim,
    d_ff=d_ff)
        self.norm1 = nn.LayerNorm(embed_dim)
        self.norm2 = nn.LayerNorm(embed_dim)
        self.norm3 = nn.LayerNorm(embed_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self,x,enc_output,tgt_mask,input_mask=None):
        att_output = self.self_att(x,x,x,tgt_mask)
        x = self.norm1(x+self.dropout(att_output))
        enc_att_output = self.self_enc_att(x,enc_output,enc_output,
    input_mask)
        x = self.norm2(x+self.dropout(enc_att_output))
        ff_out = self.feed_forward(x)
        x = self.norm3(x+self.dropout(ff_out))
        return x
```

Nótese que en el primer bloque de atención se computa la atención de la salida respecto a sí misma (con su máscara correspondiente) y en la segunda capa de atención se utiliza la salida del codificador para desambiguar el significado de la salida de la capa de atención anterior, pudiendo así traducir sin observar a palabras futuras.

3.. Experimentación

En esta sección se cubren los tres experimentos hechos para probar el modelo de transformer creado. Debido a que el trabajo se centra en el desarrollo del código del transformer desde cero, simplemente se han creado tres modelos base, sin hacer ninguna búsqueda de hiperparámetros ni optimización. Se plantean tres modelos, uno para resolver el dataset de IMDb, el dataset SHD (Spiking Heidelnberg Digits) y un dataset de traducción de inglés a español.

3.1. IMDb Dataset

El objetivo de este primer experimento es comparar el rendimiento de un modelo creado con las clases definidas por nosotros y aquel definido con keras en clase. Por lo tanto, se usarán los mismos parámetros y la misma arquitectura que en clase (cambiando solo el positional encoding). En concreto, se toma un tamaño de vocabulario de 20000 palabras y se toman secuencias de longitud 200 como máximo. El dataset se carga con el DataLoader de PyTorch como sigue:

```
    from torch.utils.data import TensorDataset,DataLoader
batch_size = 32

# create tensor datasets
trainset = TensorDataset(torch.from_numpy(x_train),
                         torch.from_numpy(y_train))
validset = TensorDataset(torch.from_numpy(x_val),
                         torch.from_numpy(y_val))
testset = TensorDataset(torch.from_numpy(x_test),
                         torch.from_numpy(y_test))

# create dataloaders
trainloader = DataLoader(trainset, shuffle=True, batch_size=batch_size)
valloader = DataLoader(validset, shuffle=True, batch_size=batch_size)
testloader = DataLoader(testset, shuffle=True, batch_size=batch_size)
```

La arquitectura usada consiste en un bloque de un codificador formado por 2 cabezas de atención, dimensión de embeding, dimensión de query y parámetro del perceptrón multicapa 32. Tras esto, convertimos el tensor de salida de dimensión $200x32$ en un tensor de dimensión 32 con una capa de pooling global. Para terminar la clasificación se incluyen dos capas totalmente conectadas de tamaños $32 - 20 - 2$, con funciones de activación relu y softmax respectivamente.

```
    class TransformerModel1(nn.Module):
    def __init__(self,embed_dim,num_heads,d_k,ff_dim,max_len,vocab_size,
    d_v=None):
        super(TransformerModel1,self).__init__()
        self.embed_dim = embed_dim
        self.max_len = max_len
        self.transformerBlock = EncoderLayer(embed_dim=embed_dim,
    num_heads=num_heads,d_ff=ff_dim,d_k=d_k,d_v=d_v)
```

```
        self.posenc = PositionalEncoding(embed_dim=embed_dim,max_len=
    max_len)
        self.embeding = nn.Embedding(num_embeddings=vocab_size,
    embedding_dim=embed_dim)
        self.fc1 = nn.Linear(embed_dim,20)
        self.dropout = nn.Dropout(0.1)
        self.fc2 = nn.Linear(20,2)
    def forward(self,x):
        x = self.posenc(self.embeding(x))
        x = self.transformerBlock(x)
        x = F.avg_pool1d(x.transpose(1,2),self.max_len).reshape(x.size(0)
    ,self.embed_dim)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        output = F.softmax(self.fc2(x),dim=-1)
        return output
```

Para el entrenamiento se usa el optimizador Adam con un *learning rate* de $1e-3$ siendo ajustado en función de las épocas en función el coseno, y con función de pérdida la entropía cruzada binaria. Además se ha entrenado por 20 epocas con un *early stopping* en función de la pérdida en el conjunto de validación con paciencia 3. Esto es:

```
    model = TransformerModel1(embed_dim=32,num_heads=2,d_k=32,ff_dim=32,
                              max_len=maxlen,vocab_size=vocab_size)
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3,betas =
    (0.9,0.999))
loss_fn = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max
    =10, eta_min=0)
```

Tras ejecutar el entrenamiento (1.4 minutos) se consigue una precisión sobre el conjunto de test de un $83\,\%$. En la figura 4 se muestra la gráfica de la pérdida y la precisión durante el entrenamiento.

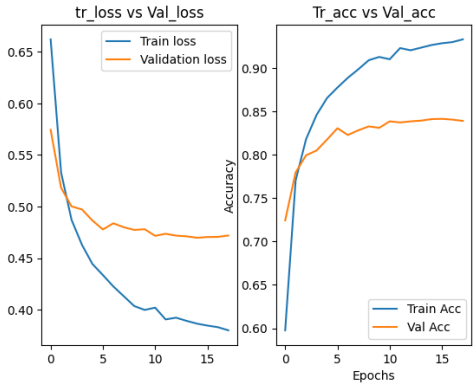

Figura 4: Evolución de función de pérdida y precisión durante el entrenamiento del dataset IMDb

3.2. SHD (Spking Heidelrberg Digits)

El conjunto de datos de SHD es un conjunto de datos neuromórfico cuyo objetivo es clasificar los números del 0 al 9 en inglés y en alemán (20 clases). Los conjuntos de datos neuromórficos tratan de imitar la codificación que hace el ser humano para procesar información. En este caso, el dataset consiste en representaciones binarias muestreadas usando un modelo de coclear artificial, Lausher [1].

El propósito del uso de este dataset es ver cómo una codificación que asemeja aquella hecha por el ser humano es válida para un transformer. En este caso cada token puede interpretarse como un sonido, vectorizado por un vector binario de 700 componentes. La longitud de la secuencia será siempre 100, es decir la misma longitud para todas las secuencias. El conjunto de datos se carga de una librería de conjunto de datos neuromórficos llamada tonic:

```
seed  = 0
train_split = 0.75
val_split = 1-train_split
sensor_size = tonic.datasets.SHD.sensor_size
path = str((os.path.abspath('') + '\\datasets'))
frame_transform = tonic.transforms.ToFrame( sensor_size=sensor_size,
    n_time_bins = 100)

trainset = tonic.datasets.SHD(save_to = path , transform =
    frame_transform, train = True)
testset = tonic.datasets.SHD(save_to = path , transform = frame_transform
    , train = False)


cache_path_train = str(os.path.abspath('') + '\\cachedDatasets\\SHD\\
    train')
cache_path_test = str(os.path.abspath('') + '\\cachedDatasets\\SHD\\test'
    )
cached_trainset = tonic.DiskCachedDataset(trainset,transform = torch.
    from_numpy, cache_path = cache_path_train)
cached_testset = tonic.DiskCachedDataset(testset, transform = torch.
    from_numpy, cache_path = cache_path_test)
(cached_trainset,cached_valset) = torch.utils.data.random_split(
    cached_trainset,[train_split,val_split],
                                                        generator =
    torch.Generator().manual_seed(seed))

trainloader = torch.utils.data.DataLoader(cached_trainset, batch_size =
    batch_size,
                                    collate_fn = tonic.collation.
    PadTensors(batch_first = True), shuffle = True)
valloader = torch.utils.data.DataLoader(cached_valset, batch_size =
    batch_size,
                                    collate_fn = tonic.collation.
    PadTensors(batch_first = True), shuffle = True)
testloader = torch.utils.data.DataLoader(cached_testset, batch_size =
    batch_size,
                                    collate_fn = tonic.collation.
```

```
    PadTensors(batch_first = True))
```

Para clasificar este conjunto de datos se construye un modelo análogo al anterior, con la diferencia de que en este caso no se añade el embedding porque se asume ya hecho y solo se le aplica el *positional encoding*. Tras esto, toda la arquitectura es idéntica, cambiando unicamente las capas del perceptrón multicapa, que pasan a tener dimensiones $700 - 100 - 20$:

```
    class TransformerModel2(nn.Module):
    def __init__(self,embed_dim,num_heads,d_k,ff_dim,max_len,d_v=None):
        super(TransformerModel2,self).__init__()
        self.embed_dim = embed_dim
        self.max_len = max_len
        self.transformerBlock = EncoderLayer(embed_dim=embed_dim,
    num_heads=num_heads,d_ff=ff_dim,d_k=d_k,d_v=d_v)
        self.posenc = PositionalEncoding(embed_dim=embed_dim,max_len=
    max_len)
        #self.embeding = nn.Embedding(num_embeddings=vocab_size,
    embedding_dim=embed_dim)
        self.fc1 = nn.Linear(embed_dim,100)
        self.dropout = nn.Dropout(0.1)
        self.fc2 = nn.Linear(100,20)
    def forward(self,x):
        x = self.posenc(x)
        x = self.transformerBlock(x)
        x = F.avg_pool1d(x.transpose(1,2),self.max_len).reshape(x.size(0)
    ,self.embed_dim)
        x = self.dropout(x)
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        output = F.softmax(self.fc2(x),dim=-1)
        return output
```

Se instancia el modelo y se usan los mismos parámetros de optimización que en el conjunto de datos anterior.

```
model = TransformerModel2(embed_dim=700,num_heads=2,d_k=700,ff_dim=700,
    max_len=100)
optimizer = torch.optim.Adam(model.parameters(), lr = 1e-3,betas =
    (0.9,0.999))
loss_fn = nn.CrossEntropyLoss()
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max
    =10, eta_min=0)
```

En este caso se entrena por 50 épocas con *early stopping* en función de la pérdida en el conjunto de validación con paciencia 8. El entrenamiento duró en total 32 minutos y se obtuvo un rendimiento del $67\,\%$ en el conjunto de test. En la figura se muestra una gráfica de la evolución de la pérdida y precisión en entrenamiento y validación a lo largo del entrenamiento.

A pesar de que se observa por la pendiente de las curvas de precisión y pérdida que el modelo todavía podía presentar mejora, no se ha continuado con el entrenamiento, ya que el objetivo del trabajo no era optimizar el rendimiento en este conjunto y podría haber tomado más tiempo.

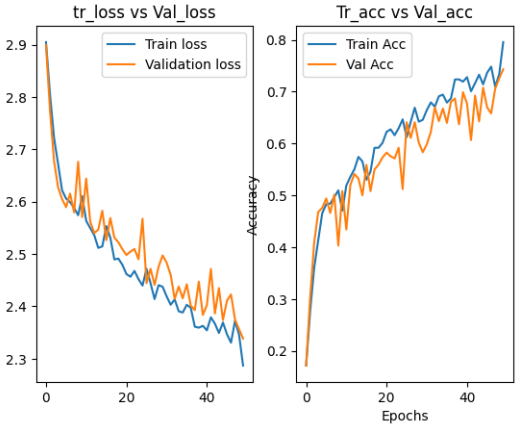

Figura 5: Evolución de función de pérdida y precisión durante el entrenamiento del dataset SHD

4.. Conclusión

En el trabajo se presenta un análisis detallado de la arquitectura de los transformers junto con una implementación a más bajo nivel que PyTorch. Además de esto, se ha aplicado la arquitectura construida a casos particulares, lo cual ha provocado una comprensión en más detalle de los modelos.

Referencias

[1] Benjamin Cramer et al. “The Heidelberg Spiking Data Sets for the Systematic Evaluation of Spiking Neural Networks”. En: *IEEE Transactions on Neural Networks and Learning Systems* 33.7 (2022), págs. 2744-2757. DOI: 10.1109/TNNLS.2020.3044364.

[2] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].