Índice del capítulo

3 Modelos de lenguaje y arquitectura GPT **1**
3.1 Formulación de un modelo de lenguaje 1
3.1.1 Tokenización 1
3.1.2 Formulación autorregresiva 5
3.2 Inferencia usando una RNN 6
3.2.1 Vectores de embedding y RNN 6
3.2.2 Entrenamiento 9
3.2.3 Generación 13
3.3 Arquitectura GPT 18
3.3.1 Layer normalization 20
3.3.2 Self attention 23
3.3.3 Red feedforward 31
3.3.4 Bloque Transformer 33
3.3.5 Clase principal 35
3.3.6 Entrenamiento y generación 37

Referencias **43**

Capítulo 3

Modelos de lenguaje y arquitectura GPT

En este capítulo revisaremos las bases de los modelos de lenguaje actuales. Para esto, comenzaremos revisando la formulación autorregresiva de una secuencia de texto y concluiremos con la implementación de un modelo GPT [1] de juguete en español. El foco estará puesto en realizar implementaciones limpias y sencillas, al mismo tiempo que se formula cada elemento de manera precisa, lo cual muchas veces es pasado por alto en la literatura de este tipo de modelos.

Las librerías que se utilizarán son las siguientes:

```python
import torch
from torch import nn, optim
from torch.utils.data import Dataset, DataLoader
from dataclasses import dataclass
import matplotlib.pyplot as plt
import tqdm
import re

DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

3.1 Formulación de un modelo de lenguaje

3.1.1 Tokenización

Una secuencia de tokens es un conjunto ordenado finito, $S = (w_1, ..., w_T)$, donde los elementos $w_1, ..., w_T$ son símbolos (e.g., letras o palabras), denominados **tokens**, que pertenecen a un vocabulario común, $\mathcal{V}_0 = \{a_1, ..., a_K\}$ (e.g., letras del abecedario o palabras de un idioma). Por lo general, el vocabulario base $\mathcal{V}_0$ suele ser extendido a otro vocabulario $\mathcal{V} \supset \mathcal{V}_0$ para incluir algunos **tokens especiales** que son utilizados durante el entrenamiento e inferencia de un modelo generativo para texto. En este capítulo, se considerarán los siguientes tokens especiales:

- **Begin of sentence ($\langle\,$BOS$\,\rangle$):** token especial agregado al comienzo de cada secuencia. Se utilizará como token inicial durante la generación.
- **End of sentence ($\langle\,$EOS$\,\rangle$):** token especial agregado al final de cada secuencia. Se utilizará para saber cuándo detener la generación de texto.
- **Unknown ($\langle\,$UNK$\,\rangle$):** token usado para representar tokens desconocidos (fuera de $\mathcal{V}_0$). Permitirá procesar símbolos desconocidos durante el entrenamiento e inferencia.
- **Padding ($\langle\,$PAD$\,\rangle$):** token agregado al final de algunas secuencias para forzar que tengan un largo predefinido. Permitirá realizar el entrenamiento en batches de secuencias.

Por lo tanto, el vocabulario extendido que se utilizará en este capítulo es el siguiente:

$$\mathcal{V} = \left\{ a_1, ..., a_K, \underbrace{\langle\,\text{BOS}\,\rangle}_{a_{K+1}}, \underbrace{\langle\,\text{EOS}\,\rangle}_{a_{K+2}}, \underbrace{\langle\,\text{UNK}\,\rangle}_{a_{K+3}}, \underbrace{\langle\,\text{PAD}\,\rangle}_{a_{K+4}} \right\}.$$

Además, se denotará como $\mathcal{V}^*$ al conjunto de todas las secuencias finitas de tokens formadas por elementos del vocabulario $\mathcal{V}$, es decir:

$$\begin{aligned} \mathcal{V}^* &:= \{S = (w_1, ..., w_T) : w_1, ..., w_T \in \mathcal{V}, T \in \mathbb{N}\} \\ &= \bigcup_{T \in \mathbb{N}} \mathcal{V}^T, \end{aligned}$$

donde la notación $\mathcal{V}^T := \underbrace{\mathcal{V} \times \cdots \times \mathcal{V}}_{T \text{ veces}}$ representa el conjunto de secuencias de largo T (i.e., $\mathcal{V}^T = \{S = (w_1, ..., w_T) : w_1, ..., w_T \in \mathcal{V}\}$).

La siguiente clase `Tokenizer` define un vocabulario extendido, `vocabulary`, y utiliza el método `encode` para mapear un string a su respectiva secuencia de tokens (más precisamente, a su secuencia de índices de tokens según el orden del vocabulario). De forma inversa, el método `decode` transforma una lista de índices en el string de símbolos:

```
class Tokenizer:

    def __init__(self, corpus_vocabulary: list[str]) -> None:
        self.vocabulary = corpus_vocabulary + ['<BOS>', '<EOS>', '<UNK>',
'<PAD>']
        self.vocab_size = len(self.vocabulary)

        self.token_to_id = {token: id for id, token in
enumerate(self.vocabulary)}
        self.bos_id = self.token_to_id['<BOS>']
        self.eos_id = self.token_to_id['<EOS>']
```

```
        self.unk_id = self.token_to_id['<UNK>']
        self.pad_id = self.token_to_id['<PAD>']

    def encode(self, text: str) -> list[int]:
        seq_tokens = re.findall(r'\d|[^\w\s]|\w+|\s', text)
        seq_ids = [self.token_to_id.get(token, self.unk_id) for token in
seq_tokens]
        return seq_ids

    def decode(self, seq_ids: list[int]) -> str:
        seq_tokens = ''.join(self.vocabulary[i] for i in seq_ids)
        return seq_tokens
```

- Ejemplo de uso.

```
vocabulary = ['a', 'b', 'c', 'd', ' ']
tokenizer = Tokenizer(vocabulary)

print(f'<BOS> ID: {tokenizer.bos_id}.')
print(f'<EOS> ID: {tokenizer.eos_id}.')
print(f'<UNK> ID: {tokenizer.unk_id}.')
print(f'<PAD> ID: {tokenizer.pad_id}.')

token_seq = 'a b c d e'
id_seq = tokenizer.encode(token_seq)
print(f'encode("{token_seq}") = {id_seq}.')
print(f'decode({id_seq}) = "{tokenizer.decode(id_seq)}".')
```

```
<BOS> ID: 5. <EOS> ID: 6. <UNK> ID: 7. <PAD> ID: 8. encode(«a
b c d e») = [0, 4, 1, 4, 2, 4, 3, 4, 7]. decode([0, 4, 1, 4,
2, 4, 3, 4, 7]) = «a b c d <UNK>».
```

Para entrenar un modelo de lenguaje, es necesario contar con un conjunto de entrenamiento $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$ formado por secuencias de tokens definidas sobre un mismo vocabulario $\mathcal{V}$. Además, para poder trabajar con batches de secuencias, asumiremos que todas las secuencias de entrenamiento tienen un mismo largo $T \in \mathbb{N}$:

- Las secuencias de largo mayor a T serán truncadas hasta el largo máximo T.
- Las secuencias de largo menor a T serán extendidas agregando el token especial $\langle \text{PAD} \rangle$ tantas veces como sea necesario para alcanzar el largo T.

En las implementaciones, el parámetro T será representado por la el atributo `seq_length`.

El vocabulario $\mathcal{V}_0$ consistirá en todas las palabras distintas que se encuentran en el corpus contenido en el archivo `data.txt`, el cual consiste en un conjunto de cuentos en español, separados por un salto de línea. En consecuencia, cada secuencia del conjunto de entrenamiento, $S = (w_1, ..., w_T) \in \mathcal{D}$, consistirá en un cuento obtenido desde el corpus (truncado o extendido hasta un largo $T \in \mathbb{N}$), con los tokens especiales agregados donde corresponda. El dataset original fue descargado desde el repositorio karen-pal/borges y se preprocesó para eliminar todos los tildes y caracteres poco frecuentes. La siguiente clase `TextDataset` implementa esta lógica:

```
class TextDataset(Dataset):

    def __init__(self, filename: str, seq_length: int) -> None:

        with open(filename, 'r', encoding='utf-8') as file:
            corpus = file.read()

        corpus_vocabulary = sorted(set(re.findall(r'\d|[^\w\s]|\w+|\s',
corpus)))
        self.tokenizer = Tokenizer(corpus_vocabulary)

        sentences = [sentence.strip() for sentence in corpus.split('\n') if
sentence.strip()]
        self.data = [[self.tokenizer.bos_id] +
self.tokenizer.encode(sentence) + [self.tokenizer.eos_id] for sentence in
sentences]

        self.seq_length = seq_length

    def __len__(self) -> int:
        return len(self.data)

    def __getitem__(self, n: int) -> torch.Tensor:

        seq_ids = self.data[n][:]

        if len(seq_ids) > self.seq_length:
            seq_ids = seq_ids[:self.seq_length]
        else:
            seq_ids += [self.tokenizer.pad_id] * (self.seq_length -
len(seq_ids))

        return torch.tensor(seq_ids)
```

Con esta clase se puede inicializar un dataset para el texto contenido en el archivo `data.txt`:

```
seq_length, batch_size = 256, 32

dataset = TextDataset('data.txt', seq_length)
dataloader = DataLoader(dataset, batch_size, shuffle=True, drop_last=True)

print(f'Tamaño del dataset: {len(dataset)} secuencias.')
print(f'Tamaño del dataloader: {len(dataloader)} batches.')
print(f'Tamaño del vocabulario: {dataset.tokenizer.vocab_size} tokens.')
```

> ```
> Tamaño del dataset: 719 secuencias. Tamaño del dataloader: 22
> batches. Tamaño del vocabulario: 60198 tokens.
> ```

Un ejemplo de secuencia contenida en este dataset es el siguiente (solo se mostrarán los primeros 40 tokens por espacio):

```
seq_ids = dataset[0]
seq_tokens = dataset.tokenizer.decode(seq_ids[:40].tolist())
print(seq_tokens)
```

> ```
> Los enanos tienen una especie de sexto sentido que les permite
> reconocerse a primera vista.
> ```

3.1.2 Formulación autorregresiva

Dada una secuencia de tokens, $S = (w_1, ..., w_T) \in \mathcal{V}^*$, un modelo autorregresivo $p_\theta(S)$ descompone la probabilidad que le asigna a la secuencia S de forma causal, expresando la probabilidad conjunta $p_\theta(S) = p_\theta(w_1, ..., w_T)$ como el producto de las probabilidades de cada token de S dados los tokens que lo anteceden:

$$p_\theta(w_1, ..., w_T) = p_\theta(w_1) \prod_{t=1}^{T-1} p_\theta(w_{t+1} \,|\, w_1, ..., w_t),$$

donde $p_\theta(w_1)$ es la distribución que el modelo le asigna al primer token de S, mientras que $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$ es la distribución del $(t+1)$-ésimo token de S dados los t tokens anteriores. Además, dado que el primer token siempre es $\langle \text{BOS} \rangle \in \mathcal{V}$, se puede fijar $p_\theta(w_1) = p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1)$, por lo que solo es necesario diseñar una red neuronal que

aprenda (los parámetros de) la distribución condicional $p_\theta(w_{t+1} \mid w_1, ..., w_t)$, la cual será utilizada posteriormente para generar nuevas secuencias de tokens. Notar que, a priori, la descomposición causal no realiza ninguna suposición de independencia sobre los tokens que componen la secuencia S (como sí ocurre, por ejemplo, en las cadenas de Markov), aunque sí impone que el orden topológico del DAG asociado a esta red bayesiana sea el orden temporal de las variables.

Por otra parte, dado que cada distribución condicional $p_\theta(w_{t+1} \mid w_1, ..., w_t)$ es una distribución de probabilidades sobre el vocabulario $\mathcal{V}$ (el cuwal es finito), se puede considerar, sin pérdida de generalidad, que $p_\theta(w_{t+1} \mid w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$ es una distribución categórica sobre $\mathcal{V}$ con vector de probabilidades $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$, es decir:

$$p_\theta(w_{t+1} = a_k \mid w_1, ..., w_t) = \lambda_\theta(w_1, ..., w_t)_k \in [0, 1], \quad \text{para todo} \quad k \in \{1, ..., |\mathcal{V}|\}.$$

En consecuencia, para entrenar un modelo de lenguaje autorregresivo solo hace falta entrenar un modelo condicional que aprenda el vector de probabilidad $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$, los cuales son parametrizados mediante una red neuronal $\lambda_\theta : \mathcal{V}^* \to [0, 1]^{|\mathcal{V}|}$, la cual recibe una secuencia de tokens $(w_1, ..., w_t) \in \mathcal{V}^*$ (llamada **contexto**) y entrega una distribución sobre $\mathcal{V}$ asociada al próximo token en el tiempo $t + 1$. Por otro lado, dado que $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$ es un vector de probabilidades, se debe que cumplir que $\sum_{k=1}^{|\mathcal{V}|} \lambda_\theta(w_1, ..., w_t)_k = 1$. Esto último se consigue aplicando la función softmax al final de la red neuronal.

3.2 Inferencia usando una RNN

3.2.1 Vectores de embedding y RNN

Dado que $\mathcal{V}$ es un conjunto abstracto sin ninguna estructura matemática definida, para implementar un modelo de lenguaje utilizando redes neuronales, es necesario obtener una representación continua para cada uno de los tokens del vocabulario discreto $\mathcal{V}$ ya que, de lo contrario, conceptos como derivada o gradiente no estarán bien definidos (al menos en su forma usual). Para esto, es usual asociar cada token $a \in \mathcal{V}$ con un vector $\text{emb}(a) \in \mathbb{R}^D$, denominado **vector de embedding**. Más precisamente, el operador de embedding, $\text{emb} : \mathcal{V} \to \mathbb{R}^D$, mapea el k-ésimo token del vocabulario $\mathcal{V}$ a la k-ésima fila de una matriz $E \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$, denominada **matriz de embedding**, la cual es aprendida conjuntamente con el resto de los parámetros del modelo neuronal durante el entrenamiento. Notar que el operador de embedding asume que el vocabulario $\mathcal{V}$ tiene un orden fijo (i.e., se debería escribir $\mathcal{V} = \left(a_1, ..., a_{|\mathcal{V}|}\right)$ en vez de $\mathcal{V} = \left\{a_1, ..., a_{|\mathcal{V}|}\right\}$, pero se seguirá usando la notación de conjunto), lo cual se cumple en la clase `Tokenizer`, donde cada símbolo tiene asociado un índice, los cuales se almacenan en el atributo `token_to_id`. En PyTorch, se puede definir una capa de embedding utilizando la clase `nn.Embedding`, la cual implementa exactamente

esta lógica. La dimensión de embedding $D \in \mathbb{N}$ será denotada como `embedding_dim`, aunque también se le suele llamar **hidden dim**.

- Ejemplo de uso.

```
vocab_size, embedding_dim = dataset.tokenizer.vocab_size, 384
emb = nn.Embedding(vocab_size, embedding_dim)

x = torch.randint(vocab_size, size=[batch_size, seq_length])
y = emb(x)

assert y.shape == (batch_size, seq_length, embedding_dim)
assert emb.weight.shape == (vocab_size, embedding_dim)
```

Por otro lado, aplicar el operador de embedding al k-ésimo token del vocabulario, $a_k \in \mathcal{V}$, es equivalente al realizar el producto matricial $E^\top y_k$, donde $y_k \in \mathbb{R}^{|\mathcal{V}|}$ es el vector one--hot asociado al token a_k (i.e., es el k-ésimo vector de la base canónica de $\mathbb{R}^{|\mathcal{V}|}$). De esta forma, una capa de embedding puede verse como una capa lineal cuya entrada, $a_k \in \mathcal{V}$, es codificada mediante el vector $y_k \in \mathbb{R}^{|\mathcal{V}|}$.

Con respecto a la elección de la arquitectura neuronal a utilizar, es importante notar que el contexto $(w_1, ..., w_t) \in \mathcal{V}^*$ (entrada a la red neuronal) es de largo variable, por lo que no es posible utilizar una red fully connected estándar para parametrizar la distribución condicional $p_\theta(w_{t+1} \mid w_1, ..., w_t)$ (aunque sí se podría utilizar una red convolucional 1D), por lo que hay que usar otro tipo de arquitectura. Una opción frecuente (al menos hasta antes de la aparición de la arquitectura Transformer) es utilizar una red neuronal recurrente (RNN), la cual procesa el contexto $(w_1, ..., w_t) \in \mathcal{V}^*$ comenzando con un vector inicial $h_0 = 0 \in \mathbb{R}^D$ (o cualquier otro valor), y luego procesa iterativamente cada uno de los tokens del contexto $(w_1, ..., w_t)$, desde el primero hasta el último:

$$h_s = f_\theta(w_s, h_{s-1}), \quad \text{para todo} \quad 1 \le s \le t,$$

donde $f_\theta : \mathcal{V} \times \mathbb{R}^D \to \mathbb{R}^D$ es una red neuronal cuya entrada es un token del vocabulario $\mathcal{V}$ y el vector asociado a la iteración anterior. De este modo, se puede considerar la última iteración, $h_t \in \mathbb{R}^D$, para definir $\lambda_\theta(w_1, ..., w_t) = g_\theta(h_t) \in [0, 1]^{|\mathcal{V}|}$, donde $g_\theta : \mathbb{R}^D \to [0, 1]^{|\mathcal{V}|}$ es una función que transforma el último vector $h_t \in \mathbb{R}^D$ en un vector de probabilidades sobre $\mathcal{V}$ (a los bloques finales que cumplen esta función se les suele llamar **cabezales de lenguaje**).

Un ejemplo simple (que será usado como baseline) es la red de Elman clásica [2], la cual considera

- $f_\theta(w_s, h_{s-1}) = \tanh(A\,\mathrm{emb}(w_s) + B\,h_{s-1} + d)$.
- $g_\theta(h_t) = \mathrm{softmax}(C\,h_t + e)$.

Notar que se abusó de notación en $f_\theta(w_s, h_{s-1})$, ya que la función de activación $\tanh : \mathbb{R} \to [-1, 1]$ es aplicada coordenada a coordenada.

De esta forma, los parámetros entrenables del modelo son la capa de embedding emb (la cual aprende una matriz $E \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$), las matrices $A, B \in \mathcal{M}_{D,D}(\mathbb{R})$, $C \in \mathcal{M}_{|\mathcal{V}|,D}(\mathbb{R})$, y los vectores de sesgo $d \in \mathbb{R}^D$, $e \in \mathbb{R}^{|\mathcal{V}|}$. Para efectos de implementación, no se agregará la capa softmax en la salida del modelo. Es decir, el modelo no retornará directamente un vector de probabilidad, si no que retornará un vector no normalizado ($C\, h_t + e \in \mathbb{R}^{|\mathcal{V}|}$ en este caso) denominado **vector de logits**. Esto permitirá calcular la función de pérdida (log--verosimilitud) de forma más eficiente ya que, como se verá en la sección de entrenamiento, la función logaritmo deshace la normalización realizada por la función softmax.

```
class GenerativeRNN(nn.Module):

    def __init__(self, vocab_size: int, embedding_dim: int) -> None:
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.rnn = nn.RNN(embedding_dim, embedding_dim, batch_first=True)
        self.lm_head = nn.Linear(embedding_dim, vocab_size)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        seq_embedding = self.embedding(x.long())
        rnn_output, _ = self.rnn(seq_embedding)
        logits = self.lm_head(rnn_output)
        return logits
```

- Ejemplo de uso.

```
vocab_size, embedding_dim = dataset.tokenizer.vocab_size, 384
rnn = GenerativeRNN(vocab_size, embedding_dim)

x = torch.randint(vocab_size, size=[batch_size, seq_length])
y = rnn(x)

assert y.shape == (batch_size, seq_length, vocab_size)
```

En este capítulo, se utilizará una red recurrente con $D = 384$:

```
rnn = GenerativeRNN(dataset.tokenizer.vocab_size, embedding_dim=384)

n_params = sum(param.numel() for param in rnn.parameters()) / 1e6
print(f'Cantidad de parámetros: {n_params:.3} millones.')
```

```
Cantidad de parámetros: 46.6 millones.
```

3.2.2 Entrenamiento

Para entrenar una red neuronal $\lambda_\theta : \mathcal{V}^* \to [0,1]^{|\mathcal{V}|}$ que aprenda el vector de probabilidades de la distribución $p_\theta(w_{t+1} \mid w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$, el enfoque de optimización natural es el de máxima verosimilitud, el cual consiste en maximizar la cantidad $\log p_\theta(\mathcal{D})$ definida sobre el conjunto de entrenamiento i.i.d. $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$. Sin embargo, considerando que $\log p_\theta(\mathcal{D}) \in (-\infty, 0]$, es más usual formular este problema de manera equivalente como la minimización de $-\log p_\theta(\mathcal{D}) \geq 0$, es decir, la red neuronal es optimizada optimizando

$$\min_\theta - \sum_{S \in \mathcal{D}} \log p_\theta(S).$$

De acuerdo a la factorización causal que realizan los modelos autorregresivos, para una secuencia $S = (w_1, ..., w_T) \in \mathcal{V}^*$, su log-verosimilitud se puede escribir como la suma de las log-probabilidades que el modelo le asigna a cada token de la secuencia, dados los tokens anteriores como contexto:

$$\begin{aligned} \log p_\theta(S) &= \log\left(p_\theta(w_1) \prod_{t=1}^{T-1} p_\theta(w_{t+1} \mid w_1, ..., w_t) \right) \\ &= \cancel{\log p(w_1)} + \sum_{t=1}^{T-1} \log p_\theta(w_{t+1} \mid w_1, ..., w_t), \end{aligned}$$

donde en la última igualdad, $\log p(w_1) = 0$ ya que $p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1) = 1$ (todas las secuencias en $\mathcal{D}$ comienzan con $w_1 = \langle \text{BOS} \rangle$). Sin embargo, una práctica usual es optimizar la log-verosimilitud normalizada por dos cantidades que influyen directamente en la función de costo $-\log p_\theta(\mathcal{D})$:

- **Por el tamaño del batch $\mathcal{D}$:** esto permite que la cantidad $\log p_\theta(\mathcal{D})$ sea independiente del tamaño del batch, por lo que no hay que ajustar la tasa de aprendizaje según esta cantidad.
- **Por el largo de cada secuencia $S \in \mathcal{D}$:** esto cambia levemente el problema para optimizar la log-verosimilitud promedio por token (en vez de la log-verosimilitud de las secuencias), lo cual es importante ya que $\log p_\theta(S^1) \geq \log p_\theta(S^2)$ si $|S^1| \leq |S^2|$ (i.e., sin normalizar, el modelo le da mayor verosimilitud a secuencias cortas).

En resumen, si se cuenta con un dataset de entrenamiento $\mathcal{D} = \{S^1, ..., S^N\} \subset \mathcal{V}^*$, donde cada secuencia $S^n = \left(w_1^{(n)}, ..., w_{T_n}^{(n)}\right) \in \mathcal{D}$ es de largo $T_n \in \mathbb{N}$, el problema de optimización estándar para el entrenamiento de un LLM es el siguiente:

$$\min_\theta -\frac{1}{N}\sum_{n=1}^{N}\frac{1}{T_n}\sum_{t=1}^{T_n-1}\log p_\theta\left(w_{t+1}^{(n)} \,\middle|\, w_1^{(n)},...,w_t^{(n)}\right).$$

Si bien en esta implementación todos los batches tendrán el mismo tamaño y todas las secuencias tendrán el mismo largo (gracias al truncamiento y padding), en entrenamientos más eficientes no siempre es así. Por ejemplo, para ahorrar computación y memoria durante el entrenamiento, se pueden formar batches de secuencias de largos similares para minimizar el uso de truncamiento y padding. Del mismo modo, los batches también pueden ser de diferente tamaño, sobretodo cuando se indica `drop_last=False` al instanciar el DataLoader, donde el último batch puede ser de un tamaño menor que el resto debido a la falta de más muestras en el dataset.

Por otra parte, es importante notar que, para una secuencia $S = (w_1, ..., w_T) \in \mathcal{D}$, esta función de pérdida no necesita conocer todo el vector de probabilidades $\lambda_\theta(w_1, ..., w_t) \in \Delta^{|\mathcal{V}|}$ asociado a la distribución $p_\theta(w_{t+1} \,|\, w_1, ..., w_t)$, si no que es suficiente conocer su valor en la coordenada correspondiente al token $w_{t+1} \in \mathcal{V}$ correcto. En consecuencia, es suficiente implementar el cálculo del logaritmo y softmax del vector de logits solo para la coordenada que se necesita (y no para toda la salida de la red neuronal), lo cual resulta computacionalmente más eficiente.

La clase `nn.CrossEntropyLoss` de PyTorch permite calcular eficientemente esta función objetivo cuando la red neuronal está implementada para retornar el vector de logits (i.e., cuando no se aplica la función softmax en la salida de la red neuronal). En efecto, si `logits` representa la salida de la red neuronal para un batch de secuencias dado como input y `targets` es un tensor con los índices de los tokens que debería predecir el modelo, el valor obtenido con `nn.CrossEntropyLoss` equivale a la función de costo anterior:

```
batch_size, seq_length, vocab_size = 32, 256, dataset.tokenizer.vocab_size

logits = torch.randn(batch_size, seq_length, vocab_size)
targets = torch.randint(0, vocab_size, size=(batch_size, seq_length))

# Cálculo de la función de pérdida paso a paso:
probs = logits.softmax(dim=-1)
target_probs = probs.gather(dim=-1,
index=targets.unsqueeze(-1)).squeeze(-1)

log_likelihood = target_probs.log().sum(dim=-1)
normalized_log_likelihood = log_likelihood / seq_length
manual_loss = - normalized_log_likelihood.mean()

# Cálculo de la función de pérdida con CrossEntropyLoss:
```

```
loss_fn = nn.CrossEntropyLoss()
logits_reshaped = logits.view(batch_size * seq_length, vocab_size)
targets_reshaped = targets.view(batch_size * seq_length)
direct_loss = loss_fn(logits_reshaped, targets_reshaped)

assert torch.isclose(manual_loss, direct_loss)
```

Un último detalle importante al momento de optimizar este tipo de modelos es con respecto a los tokens de padding, $\langle \text{PAD} \rangle \in \mathcal{V}$. Dado que este token solo se incluye en las secuencias para extenderlas hasta un cierto largo predefinido $T \in \mathbb{N}$, su predicción no debe incluirse en el cálculo de la función de pérdida ya que no forman parte de la secuencia original, ni se busca aprender la dinámica de extender las secuencias con tokens de padding luego de generar el token final, $\langle \text{EOS} \rangle \in \mathcal{V}$. Esta omisión se consigue incluyendo el argumento `ignore_index=pad_id` al instanciar la clase `nn.CrossEntropyLoss`.

Con todo esto, el loop de entrenamiento de un modelo de lenguaje es el siguiente:

```
def train_model(
        model: nn.Module,
        optimizer: optim.Optimizer,
        dataloader: DataLoader,
        epochs: int,
        ckpt_filename: str) -> None:

    model.to(DEVICE)
    model.train()

    pad_id = dataloader.dataset.tokenizer.pad_id
    loss_fn = nn.CrossEntropyLoss(ignore_index=pad_id)

    training = {'losses': [], 'model': None}

    try:
        progressbar = tqdm.trange(epochs)
        for epoch in progressbar:

            for seq_batch in dataloader:

                seq_batch = seq_batch.to(DEVICE)
                x_batch, y_batch = seq_batch[:, :-1], seq_batch[:, 1:]

                logits = model(x_batch)

                batch_size, seq_length, vocab_size = logits.shape
```

```
                logits = logits.reshape(batch_size * seq_length,
vocab_size)
                y_batch = y_batch.reshape(batch_size * seq_length)

                loss = loss_fn(logits, y_batch)

                optimizer.zero_grad()
                loss.backward()
                optimizer.step()

                training['losses'].append(loss.item())
                progressbar.set_postfix(loss=loss.item())

    except KeyboardInterrupt:
        print('Entrenamiento interrumpido.')

    training['model'] = model.state_dict()
    torch.save(training, ckpt_filename)
```

Se entrenará la RNN definida anteriormente durante 32 épocas utilizando AdamW [3], el cual es un optimizador usual para LLMs (aunque hoy en día también se utilizan otras opciones más modernas):

```
rnn_optimizer = optim.AdamW(rnn.parameters())
train_model(rnn, rnn_optimizer, dataloader, epochs=32,
ckpt_filename='rnn_training.pt')
```

El entrenamiento toma aproximadamente 5 minutos en Google Colab (usando una GPU NVIDIA Tesla T4). Una vez concluido el entrenamiento, se puede cargar el modelo y ver la evolución de la función de costo a lo largo del entrenamiento:

```
rnn_training = torch.load('rnn_training.pt', DEVICE, weights_only=True)
rnn.load_state_dict(rnn_training['model'])

plt.figure(figsize=(10, 5))
plt.plot(rnn_training['losses'])
plt.xlabel('Iteración')
plt.ylabel('Entropía cruzada media')
plt.grid(alpha=0.3)
plt.show()
```

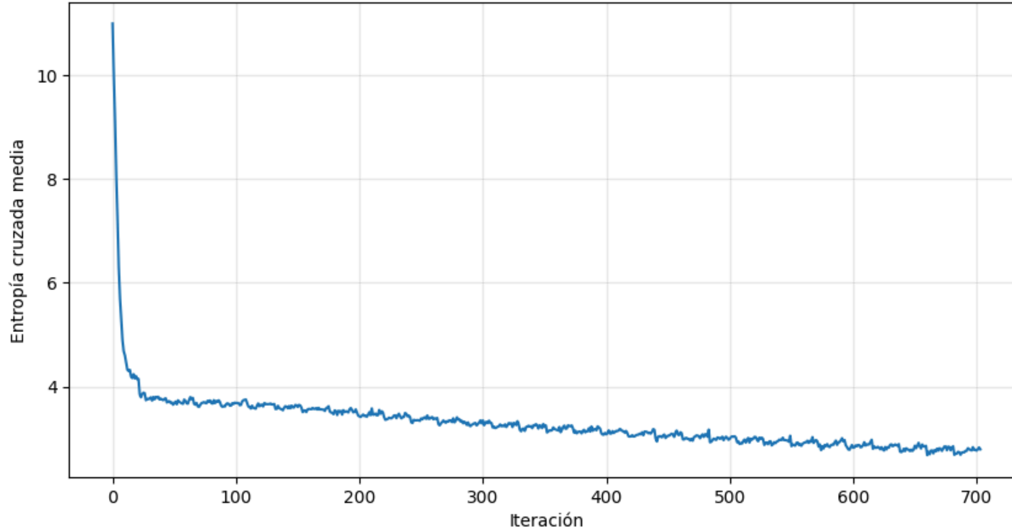

Figura 1: Evolución de la entropía cruzada durante el entrenamiento de la RNN generativa.

Notar que, en un comienzo, la entropía cruzada es cercana a $\log(|\mathcal{V}|) \approx 11$ nats, que es lo máximo que puede valer cuando el modelo asigna probabilidades uniformes a cada token.

3.2.3 Generación

Para la generación de nuevas secuencias de tokens, se puede utilizar el algoritmo de ancestral sampling revisado en la entrada de redes bayesianas. En el caso del enfoque autorregresivo, la generación comienza con el token inicial $w_1 = \langle \text{BOS} \rangle$ (recordar que se fijó $p(w_1) = \delta_{\langle \text{BOS} \rangle}(w_1)$) y luego se debe evaluar iterativamente el modelo neuronal entrenado para predecir el siguiente token, utilizando como contexto la secuencia de tokens que ya han sido generados. La generación de nuevos tokens continúa hasta que se genera el token $\langle \text{EOS} \rangle$ o bien, hasta que se genera una cantidad máxima de tokens. Del mismo modo, se podría realizar generación condicional comenzando con una secuencia de tokens iniciales (llamada **prompt**), $S_{\text{prompt}} = (w_1, ..., w_t)$, lo cual es una tarea usual en aplicaciones como chatbots o agentes.

Dado que el modelo neuronal retorna (los logits de) la distribución $p_\theta(w_{t+1} \mid w_1, ..., w_t) \sim \text{Categorical}(\mathcal{V}; \lambda_\theta(w_1, ..., w_t))$ asociada al siguiente token dado el contexto actual, se debe elegir un criterio para decidir cuál va a ser el próximo token a partir de la distribución entregada por el modelo. Si $S = (w_1, ..., w_t) \in \mathcal{V}^*$ son los tokens de entrada (contexto) y $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$ es el vector de probabilidades (luego de aplicar $\mathsf{softmax}$) que entrega el modelo neuronal para el próximo token (i.e., $p_\theta(w_{t+1} = a_k \mid S) = \lambda_\theta(S)_k$, para todo $k \in \{1, ..., |\mathcal{V}|\}$), una primera idea podría ser elegir el siguiente token $w_{t+1} \in \mathcal{V}$ como el token más probable según el modelo, es decir:

$$w_{t+1} = a_k, \quad \text{donde} \quad k = \underset{k \in \{1, ..., |\mathcal{V}|\}}{\arg\max} \; \lambda_\theta(S)_k$$

Una limitación de este enfoque, denominado **greedy decoding**, es que la elección del token más probable no es necesariamente una decisión óptima ya que, si bien la probabilidad de este token es alta, puede ocurrir que todas las futuras probabilidades sean bajas producto de haber elegido el token más probable en el momento (de ahí viene el adjetivo greedy). Para permitir más variabilidad en la generación, se puede elegir el siguiente token generando una muestra desde la distribución categórica $\text{Categorical}(\mathcal{V}; \lambda_\theta(S))$ (e.g., utilizando `torch.multinomial`) y definir w_{t+1} como la muestra generada. Esto garantiza que el modelo genere siempre secuencias nuevas, aunque reciba la misma secuencia inicial. Sin embargo, dado que $\lambda_\theta(S)_k > 0$ para todo $k \in \{1, ..., |\mathcal{V}|\}$ (ya que este vector se determina usando softmax y $e^l > 0, \forall l \in \mathbb{R}$), el sampling desde la distribución categórica usando $\lambda_\theta(S)$ puede generar, eventualmente, tokens muy poco probables, lo que muchas veces se traduce en secuencias de texto sin sentido debido a la propagación del error de generación. Una solución inmediata a este problema es permitir elegir solamente los $K \geq 1$ tokens más probables, redistribuyendo la masa de los $|\mathcal{V}| - K$ tokens restantes: si $\mathcal{J} = \{k_1, ..., k_K\} \subset \{1, ..., |\mathcal{V}|\}$ son las K coordenadas donde $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$ toma los valores más altos, se puede construir el vector de probabilidad truncado, $\tilde{\lambda}_\theta(S) \in \Delta^{|\mathcal{V}|}$, cuya k-esima coordenada es

$$\tilde{\lambda}_\theta(S)_k = \begin{cases} 0 & \text{si} \;\; k \notin \mathcal{J} \\ \lambda_\theta \frac{(S)_k}{\sum_{i \in \mathcal{J}} \lambda_\theta(S)_i} & \text{si} \;\; k \in \mathcal{J} \end{cases}.$$

Luego, la técnica de **top-K sampling** elige el siguiente token generando una muestra desde la distribución $\text{Categorical}(\mathcal{V}; \tilde{\lambda}_\theta(S))$. Notar que si $K = 1$ se recupera la generación greedy, mientras que si $K = |\mathcal{V}|$ se recupera el sampling categórico usando el vector de probabilidades original, $\lambda_\theta(S) \in \Delta^{|\mathcal{V}|}$. Por otro lado, es importante mencionar que el escalar $K \in \mathbb{N}$ definido aquí no tiene relación con el escalar $K \in \mathbb{N}$ definido al comienzo del capítulo (usado para denotar la cardinalidad del vocabulario base, $\mathcal{V}_0 = \{a_1, ..., a_K\}$), solo es un alcance de nombre.

Otra heurística comúnmente utilizada corresponde a aplicar la idea de temperatura, la cual modifica el vector de probabilidades $\lambda_\theta(S)$ para volverlo más determinista (baja temperatura) o más uniforme (alta temperatura). Más precisamente, si $l \in \mathbb{R}^{|\mathcal{V}|}$ corresponde al vector de logits entregado por la red neuronal (i.e., el vector previo a aplicar softmax) y $T > 0$ es un escalar, denominado **temperatura**, se puede considerar una versión escalada del vector de logits, $\tilde{l} = \frac{l}{T} \in \mathbb{R}^{|\mathcal{V}|}$, y aplicar softmax a esta nueva cantidad para obtener el vector de probabilidades desde el que se realiza el sampling del próximo token (ya sea usando todo el vector o usando top-K sampling). El objetivo de esta ponderación es llevar todos los logits a una escala similar (si $T > 1$) o destacar aún más el logit del token más

probable (si $T < 1$). Notar que el escalar de temperatura $T \in \mathbb{R}_{++}$ definido aquí no tiene relación con el escalar $T \in \mathbb{N}$ usado para definir el largo de las secuencias, solo es un alcance de nombre.

En la siguiente figura se observa cómo cambia el nuevo vector de probabilidades, $\mathrm{softmax}(\tilde{l}) \in \Delta^{|\mathcal{V}|}$, para distintos valores de temperatura, donde $T = 1$ corresponde al vector de probabilidades $\lambda_\theta(S)$ que se obtendría sin aplicar esta técnica:

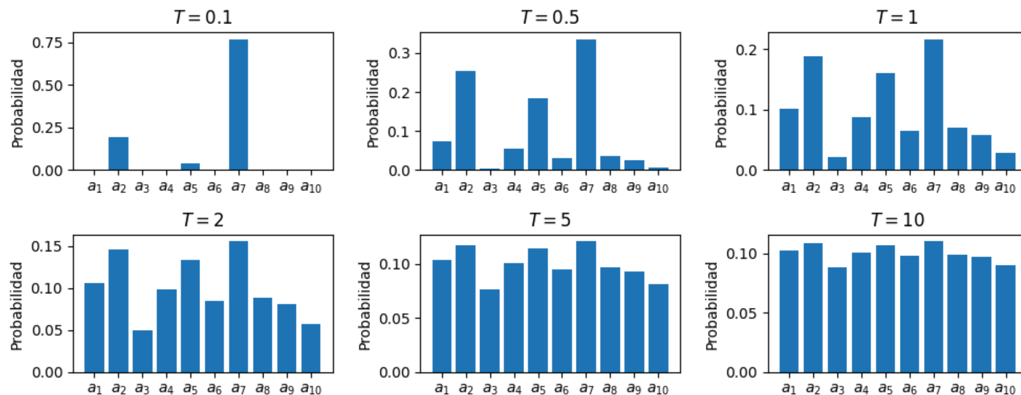

Figura 2: Vector de probabilidades obtenido al aplicar softmax a un vector de logits reescalado por distintos valores de temperatura.

- Código para generar la figura.

```
vocab_size = 10
vocabulary = [f'$a_{{{k}}}$' for k in range(1, vocab_size + 1)]
logits = torch.randn(vocab_size)
temperatures = [0.1, 0.5, 1, 2, 5, 10]

fig, axes = plt.subplots(2, 3, figsize=(10, 4))
axes = axes.flatten()

for i, t in enumerate(temperatures):
    scaled_logits = logits / t
    prob = scaled_logits.softmax(-1)
    axes[i].bar(vocabulary, prob)
    axes[i].set_title(f'$T={t}$')
    axes[i].set_ylabel(f'Probabilidad')

plt.tight_layout()
plt.show()
```

Notar que si $T \to 0$ se recupera la generación greedy, mientras que si $T \to \infty$ el vector de probabilidades se vuelve uniforme (con una probabilidad $\frac{1}{|\mathcal{V}|}$ para cada token). Por lo

tanto, el parámetro de temperatura $T > 0$ permite controlar el trade-off entre versatilidad y confianza en la generación, lo cual es una herramienta muy útil en algunos contextos. Por ejemplo, para la generación de código es usual elegir una baja temperatura, mientras que para la generación de textos narrativos, puede ser preferible utilizar una temperatura más elevada.

Por último, otra heurística usual es penalizar la probabilidad de tokens que ya han sido generados o que estén en la secuencia inicial. Esta técnica se aplica debido a que es común que los modelos de lenguaje basados en redes neuronales entren en ciclos de repetición de un mismo token o secuencias de tokens. Si $c(a_k) \in \mathbb{N}$ es la cantidad de veces que el símbolo $a_k \in \mathcal{V}$ ha aparecido en la secuencia generada hasta el momento, se pueden escalar los logits $l \in \mathbb{R}^{|\mathcal{V}|}$ mediante

$$\tilde{l}_k = \begin{cases} \frac{1}{\gamma^{c(a_k)}} \cdot l_k & \text{si} \quad l_k > 0 \\ \gamma^{c(a_k)} \cdot l_k & \text{si} \quad l_k \leq 0 \end{cases},$$

donde el hiperparámetro $\gamma \geq 1$ controla el nivel de penalización. Más específicamente, si $\gamma > 1$, entonces los logits de tokens repetidos decrecen, reduciendo su probabilidad tras aplicar softmax.

Todas estas técnicas pueden ser incluidas dentro de un mismo loop de generación:

```
def generate_tokens(
        model: nn.Module,
        context: str,
        tokenizer: Tokenizer,
        temperature: float = 0.8,
        top_k: int = 50,
        max_tokens: int = 256,
        repetition_penalty: float = 1.01) -> str:

    model.to(DEVICE)
    model.eval()

    seq_id = [tokenizer.bos_id] + tokenizer.encode(context)
    seq_id = torch.tensor(seq_id, device=DEVICE)

    with torch.no_grad():
        for _ in range(max_tokens):
            logits = model(seq_id.unsqueeze(0))[0, -1, :]

            token_counts = torch.bincount(seq_id, minlength=logits.size(0))
            for token_id, count in enumerate(token_counts):
```

```python
                if count > 0:
                    if logits[token_id] > 0:
                        logits[token_id] /= (repetition_penalty ** count)
                    else:
                        logits[token_id] *= (repetition_penalty ** count)

            if temperature == 0:
                next_token = torch.argmax(logits, dim=-1, keepdim=True)
            else:
                logits = logits / temperature

                top_k_logits, top_k_indices = torch.topk(logits, top_k)
                probs = top_k_logits.softmax(dim=-1)
                next_token = top_k_indices[torch.multinomial(probs,
num_samples=1)]

            seq_id = torch.cat((seq_id, next_token), dim=0)

            if next_token in (tokenizer.eos_id, tokenizer.pad_id):
                break

    return tokenizer.decode(seq_id.tolist())
```

Notar que la técnica de penalización por repetición se aplica antes de aplicar la de temperatura. Por otro lado, la implementación realizada no es muy eficiente: hay cálculos que podrían reutilizarse (p.g. usando programación dinámica), y en la parte de penalización por repetición, el loop con `enumerate(token_counts)` itera sobre todo el vocabulario (se podría hacer más eficiente operando solo sobre los tokens que efectivamente aparecen). En la entrada de LLMs y agentes se revisarán algunas heurísticas más avanzadas para un entrenamiento e inferencia más eficiente.

Usando la función anterior y el modelo neuronal ya entrenado, se puede realizar generación condicional continuando una secuencia de texto dada:

```python
context = 'Habia una vez'

for n in range(10):
    new_tokens = generate_tokens(rnn, context, dataset.tokenizer)
    print(new_tokens)
```

> ```
> Habia una vez por la luz del otro, de la puerta y de los detalles de
> los ojos, y el que no se acerco el caso. Habia una vez que me dijo
> ```

> que habia que en los ojos del mundo, y de la hora. Y es de otro. Habia una vez que como el dia los ojos de la tarde y su vida. Ya son menos. Pero, se sento en que, que lo pocos de los cuatro. Como el brazo, de los dedos, el sol, las cosas. En la ultima hora, habia? Habia una vez una manera de las yerbas que habia sido tan dias, y a la puerta, y el cuarto. ¿Que dio a el espacio de la mañana. Habia una vez un rio para el muro y la luz. No puede la mañana, al instante… ¿Es siempre por la costa. Sus noche, se llamaba a la tierra, sin la pequeña spicula…… Habia una vez que solo como a los dos años, no se ve sus amigos, es la casa de los dos del sol. Todo el mundo, despues, no, las pesadas mujer, ¡Era con una mesa,……. Habia una vez que no se me dijo nunca, de la puerta que en la muerte, como que? ¡No….. Habia una vez ese que no se ha que se encamino a quien estaba en el instante. Las calles y a la ciudad. Habia una vez por la costa del hombre del brazo, aunque no hace que su lado que se le habia veces se trata. Habia una vez su luz del corredor de los hombres para la puerta, y se no de las manos, pues, la calle. Para entre la puerta, a la cabeza, que?

En la generación anterior se observa que, si bien el modelo fue capaz de aprender a separar cada palabra por un espacio, el texto que genera es incoherente. Si bien es posible obtener mejores resultados cambiando algunos hiperparámetros de generación (o usando una RNN más profunda), en la segunda parte de este capítulo se estudiará la arquitectura GPT, la cual define el esqueleto común de prácticamente todos los LLMs modernos.

Otras heurísticas de generación

Pendiente: beam search, nucleus sampling, min-p sampling, penalización por n-gramas, sampling especulativo, contrastive search.

3.3 Arquitectura GPT

A pesar de que las RNNs permiten procesar secuencias de largo variable, tienen limitaciones importantes que no permiten extenderlas fácilmente a escenarios más complejos como los que enfrentan los LLMs de hoy en día. Por ejemplo, omitiendo algunos componentes por simplicidad, una RNN está compuesta esencialmente por iteraciones de la forma $h_s = A\,\mathrm{emb}(w_s) + B\,h_{s-1}$, donde se observa que la matriz $B \in \mathcal{M}_{D,D}(\mathbb{R})$ es multiplicada por sí misma varias veces, dependiendo del largo de la secuencia que se esté procesando. Esta multiplicación reiterada provoca que el gradiente $\frac{\partial L}{\partial B}$ (con L la función de pérdida) crezca de manera irrestricta (**exploding gradient problem**) o bien, decrezca hasta hacerse

cercano a cero (**vanishing gradient problem**), lo cual dependerá del radio espectral de la matriz B. Ambos problemas dificultan el entrenamiento de este tipo de modelos ya que un gradiente irrestricto provoca un entrenamiento inestable (que puede incluso divergir), mientras que un gradiente nulo no permite actualizar los parámetros de la red neuronal por algoritmos de gradiente. Más aún, el tamaño de estos gradientes no es uniforme a lo largo de las capas de la red neuronal, por lo que una tasa de aprendizaje pequeña (e.g., para aminorar la inestabilidad del exploding gradient) puede ser contraproducente para el desvenecimiento de gradiente observado en las primeras capas.

Si bien se han propuesto algunas modificaciones para mitigar estos problemas (e.g., **gradient clipping** para el exploding gradient problem o arquitecturas con compuertas como GRU o LSTM para el vanishing gradient problem), las RNNs siguen teniendo limitaciones intrínsecas debido a su forma recursiva. En efecto, su procesamiento secuencial ralentiza considerablemente el entrenamiento de este tipo de modelos ya que cada secuencia se debe procesar token a token (i.e., no se pueden procesar todos los tokens en paralelo). Por otro lado, toda la información de la secuencia que se está procesando debe poder almacenarse en un vector $h_t \in \mathbb{R}^D$, lo cual es inviable para secuencias largas, provocando que el modelo empiece a olvidar la información de los primeros tokens de la secuencia a medida que se avanza en la recursion (en particular, las RNNs tienen un sesgo hacia los tokens más recientes). La arquitectura Transformer [4], y en particular los modelos tipo GPT, solucionan (al menos en principio) las limitaciones anteriores:

- Las secuencias $S = (w_1, ..., w_T) \in \mathcal{V}$ se procesan en paralelo, es decir, se pueden procesar todos los tokens de todas las secuencias de un batch de entrenamiento en una sola llamada a la red neuronal. En particular, esto permite escalar el entrenamiento en tamaño del dataset y largo de las secuencias. Sin embargo, la generación sigue siendo secuencial (token a token).
- No se produce el problema de vanishing/exploding gradient gracias a que se insertan capas de normalización y conexiones residuales entre las capas de la red neuronal.
- No se olvida información al procesar la secuencia ya que cada token puede poner atención a la información que contienen todos los otros tokens de la secuencia.

En esencia, la arquitectura GPT consiste en mejorar una secuencia de embeddings iniciales, $(x_1, ..., x_T) \subset \mathbb{R}^D$ (asociados a una secuencia de tokens $S = (w_1, ..., w_T) \in \mathcal{V}^*$ dados como entrada), pasándolos a través de una serie de bloques neuronales, llamados bloques Transformer, los cuales buscan enriquecer cada uno de estos embeddings con información adicional obtenida de los tokens vecinos dentro de la secuencia. De este modo, al pasar por el último bloque Transformer, los embeddings obtenidos son lo suficientemente ricos en información como para poder pasarlos por un cabezal de lenguaje simple (similar al usado en la RNN) para realizar la predicción del próximo token. Cada uno de estos

bloques Transformer, que son el elemento principal de la arquitectura GPT, está formado principalmente por dos componentes:

- **Módulo de self-attention:** en este sub-módulo cada token interactúa con los otros tokens de la secuencia para obtener una representación más acorde al contexto.
- **Módulo feedforward:** red fully connected de dos capas usada para mejorar la representación obtenida en el módulo de auto-atención. Este sub-módulo es aplicado de forma independiente a cada token.

Adicionalmente, la arquitectura GPT tiene otros componentes: al comienzo de la arquitectura se debe incluir una capa de embedding inicial (llamada **token embedding**), a la cual se le suma una capa de embedding posicional (llamada **positional encoding**), la cual permitirá inyectar la información acerca de la posición en la que se encuentra cada token. A la salida del último bloque Transformer, se anexa un cabezal de lenguaje similar al usado en la RNN implementada anteriormente. Adicionalmente, cada uno de los componentes que forman la arquitectura GPT incluye sub-módulos menores como capas de normalización (en este caso, layer normalization) y dropout.

En esta sección se explicará e implementará cada uno estos componentes, intentando dar una motivación natural para cada uno de ellos. La siguiente clase `GPTConfig` contiene todos los hiperparámetros que definen la arquitectura GPT (se utilizarán valores pequeños para poder realizar el entrenamiento en Google Colab):

```python
@dataclass
class GPTConfig:
    vocab_size: int = dataset.tokenizer.vocab_size
    context_window: int = 256
    embedding_dim: int = 384
    num_layers: int = 6
    num_heads: int = 6
    head_dim: int = 64
    ff_factor: int = 4
    dropout: float = 0.1

config = GPTConfig()
```

3.3.1 Layer normalization

El primer módulo que se revisará será la capa de normalización, la cual será añadida al comienzo de cada componente de la arquitectura (aunque originalmente se aplicaba al final de cada componente).

Al igual como ocurre en todas las redes neuronales profundas, la aplicación consecutiva de varias capas neuronales suele afectar el entrenamiento del modelo (e.g., por vanishing/exploding gradient). Para mitigar estos problemas, es usual agregar capas de normalización entremedio de los bloques de la red. Por ejemplo, en redes convolucionales es usual utilizar la técnica de batch normalization [5] (o más frecuentemente hoy en día, group normalization [6]), la cual normaliza cada canal de un batch de imágenes para que tenga media nula y varianza unitaria.

Sin embargo, no es obvio cómo aplicar esta técnica en modelos secuenciales ya que, en general, las secuencias de tokens no suelen tener el mismo largo y muchas veces los batches de entrenamiento deben ser pequeños debido a limitaciones de memoria en GPU, lo cual provoca que los estadísticos calculados no sean precisos, perjudicando el entrenamiento.

Por estos motivos, en modelos de lenguaje y en arquitecturas tipo Transformer es usual utilizar otra técnica de normalización más natural para este tipo de modelos, conocida como layer normalization [7], la cual normaliza cada token dentro de una secuencia de forma individual, por lo que su normalización no depende de otras secuencias dentro de un batch de entrenamiento ni de otros tokens de la secuencia que se está procesando. Más precisamente, para un vector de embedding $x \in \mathbb{R}^D$, layer normalization primero calcula su media $\mu \in \mathbb{R}$ y varianza $\sigma^2 \geq 0$ a lo largo de las dimensiones del embedding x:

$$\mu = \frac{1}{D} \sum_{d=1}^{D} x_d \qquad \sigma^2 = \frac{1}{D} \sum_{d=1}^{D} (x_d - \mu)^2.$$

Luego, normaliza el vector de embedding coordenada a coordenada usando los valores de μ y σ. Es decir, la d-ésima coordenada del embedding normalizado, $\tilde{x} \in \mathbb{R}^D$, viene dada por

$$\tilde{x}_d = \frac{x_d - \mu}{\sqrt{\sigma^2 + \epsilon}},$$

donde $\epsilon > 0$ es un escalar fijo y pequeño utilizado para estabilidad numérica. Finalmente, de forma análoga a lo realizado por batch normalization, layer normalization aplica una transformación afín a cada coordenada del embedding normalizado:

$$\text{LayerNorm}(x) \coloneqq \alpha \odot \tilde{x} + \beta,$$

donde $\alpha, \beta \in \mathbb{R}^D$ son parámetros entrenables del módulo LayerNorm y $\odot$ es el producto de Hadamard (i.e., $\alpha \odot \tilde{x} \in \mathbb{R}^D$ con $(\alpha \odot \tilde{x})_d = \alpha_d \, \tilde{x}_d$). La motivación detrás de esta última transformación es poder permitirle al modelo deshacer la normalización anterior. El siguiente diagrama resume este proceso:

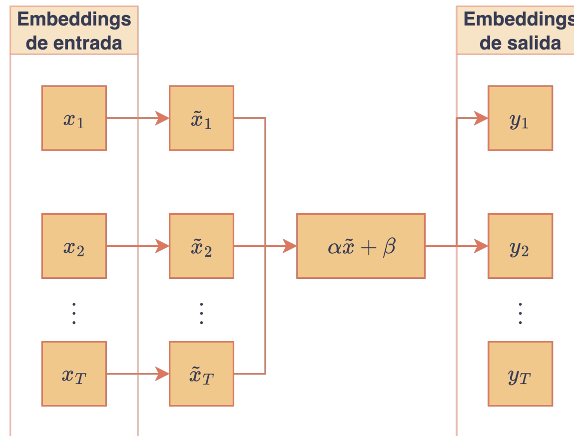

Figura 3: Diagrama del módulo de layer normalization aplicado a un vector de embedding.

La siguiente clase `LayerNorm` implementa esta técnica de normalización:

```
class LayerNorm(nn.Module):

    def __init__(self, emb_dim: int, epsilon: float = 1e-5) -> None:
        super().__init__()

        self.scale = nn.Parameter(torch.ones(emb_dim))
        self.shift = nn.Parameter(torch.zeros(emb_dim))
        self.epsilon = epsilon

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        x_norm = (x - mean) / torch.sqrt(var + self.epsilon)
        return self.scale * x_norm + self.shift
```

- Ejemplo de uso.

```
ln = LayerNorm(config.embedding_dim)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = ln(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
assert y.mean(dim=-1).allclose(torch.zeros(batch_size, seq_length),
atol=1e-5)
assert y.var(dim=-1).allclose(torch.ones(batch_size, seq_length),
atol=1e-2)
```

Notar que los parámetros α y β son inicializados como el vector $1 \in \mathbb{R}^D$ y el vector $0 \in \mathbb{R}^D$ respectivamente, lo cual garantiza que, en un comienzo, estos parámetros no tengan ningún

efecto posterior a la normalización, lo cual estabiliza el entrenamiento. Por otro lado, la transformación afín realizada luego de la normalización, $\alpha \odot \tilde{x} + \beta$, es aplicada sobre todos tokens de la secuencia (y sobre todas las secuencias de un batch de entrenamiento) usando los mismos parámetros $\alpha, \beta \in \mathbb{R}^D$ ya que esta capa de normalización es local y se aplica de forma individual para cada token.

Es importante mencionar que la implementación realizada en la clase `LayerNorm` tiene el mismo comportamiento que la clase `nn.LayerNorm` de PyTorch cuando se utiliza para procesar secuencias, pero no tiene el mismo comportamiento cuando es utilizada para batches imágenes (donde los tensores suelen ser de orden 4). Sin embargo, como se mencionó anteriormente, en imágenes es usual considerar otras técnicas de normalización como `nn.BatchNorm2d` o `nn.GroupNorm`.

3.3.2 Self attention

El componente clave de la arquitectura GPT corresponde al mecanismo de auto-atención (self-attention), el cual está encargado de mejorar los embeddings de una secuencia utilizando la información contextual de cada token. Más precisamente, dada una secuencia de vectores de embeddings, $(x_1, ..., x_T) \subset \mathbb{R}^D$, asociada a una secuencia de tokens $S = (w_1, ..., w_T) \in \mathcal{V}^*$, el objetivo del mecanismo de auto-atención es obtener otra secuencia de embeddings, $(y_1, ..., y_T) \subset \mathbb{R}^P$, donde cada embedding $y_t \in \mathbb{R}^P$ corresponderá a una versión mejorada del respectivo embedding inicial, $x_t \in \mathbb{R}^D$, la cual incorporará información contextual de todos los otros tokens de la secuencia S. Para esto, el mecanismo de auto-atención comienza proyectando cada uno de los embeddings $x_1, ..., x_T \in \mathbb{R}^D$ sobre el espacio $\mathbb{R}^P$ usando 3 proyecciones distintas (en la implementación, el escalar $P \in \mathbb{N}$ se representa por el atributo `head_dim`), es decir, para cada $t \in \{1, ..., T\}$ se definen los vectores

$$q_t = W_Q\, x_t, \qquad k_t = W_K\, x_t, \qquad v_t = W_V\, x_t,$$

donde $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ son las 3 matrices (parámetros entrenables) que realizan las 3 proyecciones lineales a $\mathbb{R}^P$ de los vectores de embedding $x_1, ..., x_T \in \mathbb{R}^D$. De esta forma, cada embedding mejorado, $y_i \in \mathbb{R}^P$, se calculará como una suma ponderada (más precisamente, una combinación convexa) de los vectores $v_1, ..., v_T \in \mathbb{R}^P$, donde los ponderadores (que serán obtenidos a partir de las proyecciones $q_1, ..., q_T \in \mathbb{R}^P$ y $k_1, ..., k_T \in \mathbb{R}^P$) indicarán el grado de importancia que tiene cada token de S para el token $w_i \in S$:

$$y_i = \sum_{j=1}^{T} a\big(q_i, k_j\big)\, v_j.$$

Cada ponderador $a(q_i, k_j) \in [0, 1]$ se interpreta como la **atención** que coloca el token w_i sobre el token w_j, lo cual se determinará utilizando las proyecciones $q_i \in \mathbb{R}^P$ y $k_j \in \mathbb{R}^P$. Por otro lado, notar que para que la combinación lineal que define $y_i \in \mathbb{R}^P$ sea convexa, se debe cumplir que $\sum_{j=1}^{T} a(q_i, k_j) = 1$.

La motivación de realizar 3 proyecciones distintas, q_t, k_t y v_t, para cada embedding $x_t \in \mathbb{R}^D$ de la secuencia de embeddings $(x_1, ..., x_T)$, viene de permitir distintas representaciones que se ajusten al papel que está jugando el respectivo token en cada situación (lógica prestada desde information retrieval):

- El vector $q_t \in \mathbb{R}^P$ (denominado **query**) es usado cuando el embedding $x_t \in \mathbb{R}^D$ es el que busca poner atención sobre otros vectores de embedding de la secuencia.
- El vector $k_t \in \mathbb{R}^P$ (denominado **key**) es usado cuando un token cualquiera de la secuencia busca poner atención sobre el embedding $x_t \in \mathbb{R}^D$.
- El vector $v_t \in \mathbb{R}^P$ (denominado **value**) es usado para representar la información útil del embedding $x_t \in \mathbb{R}^D$, la cual se considerará en la suma que define cada uno de los embeddings finales $(y_1, ..., y_T) \subset \mathbb{R}^P$.

Dado que cada embedding mejorado, $y_t \in \mathbb{R}^D$, incluye información acerca de todos los demás tokens de la secuencia $S = (w_1, ..., w_T)$ (y no solo del token $w_t \in \mathcal{V}$), a estos nuevos embeddings se les suele llamar **embeddings contextuales** ya que permite, por ejemplo, la desambiguación de palabras homógrafas, lo cual no es posible a partir de embeddings obtenidos al mirar los tokens individualmente.

Por otra parte, para satisfacer las restricciones de convexidad, $a(q_i, k_j) \geq 0$ y $\sum_{j=1}^{T} a(q_i, k_j) = 1$, es usual comenzar considerando escalares irrestrictos, $s(q_i, k_j) \in \mathbb{R}$, denominados **scores de atención**, los cuales luego son normalizados usando la función softmax, de manera similar a lo que se realiza para transformar un vector de logits en un vector de probabilidad. Notar que el cálculo de la función softmax debe realizarse a lo largo de la segunda variable de la función de score $s : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$ ya que es esa la coordenada donde se pide que la suma sea 1:

$$a(q_i, k_j) = \text{softmax}(s(q_i, k_1), ..., s(q_i, k_T))_j = \frac{e^{s(q_i, k_j)}}{\sum_{t=1}^{T} e^{s(q_i, k_t)}}.$$

Para la elección de los scores de atención, $s(q_i, k_j) \in \mathbb{R}$, si bien se podría utilizar una red neuronal más pequeña que aprenda buenos valores (como ocurre en la atención de Bahdanau [8]), un enfoque sencillo y que funciona bien corresponde al **dot-product self--attention** (propuesto en la arquitectura original de Transformer [4]), el cual considera

$$s(q_i, k_j) = \frac{\langle q_i, k_j \rangle}{\sqrt{P}},$$

donde el producto punto indica el grado de similitud entre el par query-key (recordar la relación con la similitud coseno: $\cos \angle(x, y) := \frac{\langle x,y \rangle}{\|x\|\|y\|}$), mientras que la división por $\sqrt{P}$ (con P la dimensión de los vectores q_i y k_j) busca mantener la magnitud de los scores controlada, lo cual evita que la función softmax concentre casi toda su masa en una única posición al momento de calcular los ponderadores de atención.

El siguiente diagrama resume el mecanismo de auto-atención para una secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$:

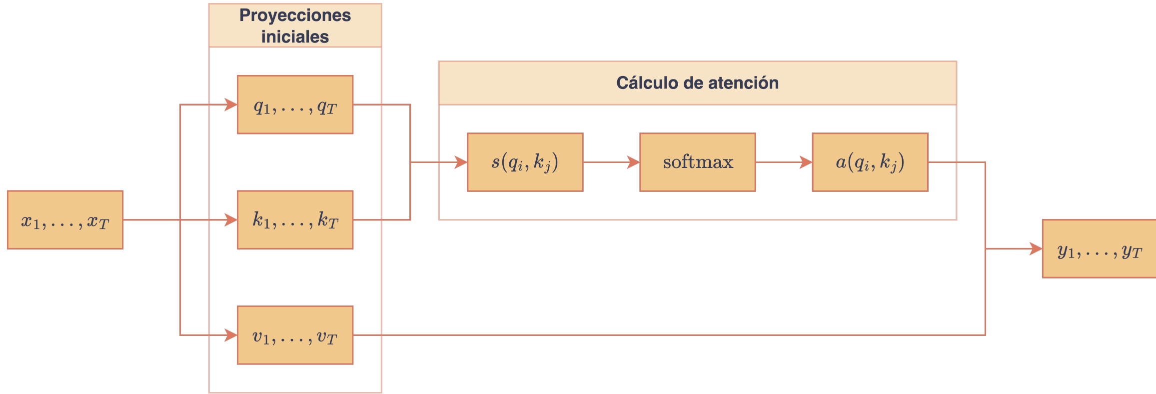

Figura 4: Diagrama del mecanismo de self-attention aplicado a una secuencia de embeddings.

Es importante notar que se pueden calcular todos los embeddings contextuales $y_1, ..., y_T \in \mathbb{R}^P$ en paralelo (hasta el momento, cada $y_t \in \mathbb{R}^P$ se debería computar de forma independiente, iterando sobre $t \in \{1, ..., T\}$). Para esto, se almacenarán las proyecciones realizadas dentro de matrices

$$Q = \begin{pmatrix} q_1^\top \\ \vdots \\ q_T^\top \end{pmatrix}, \quad K = \begin{pmatrix} k_1^\top \\ \vdots \\ k_T^\top \end{pmatrix}, \quad V = \begin{pmatrix} v_1^\top \\ \vdots \\ v_T^\top \end{pmatrix},$$

donde las 3 matrices, las cuales pertenecen a $\mathcal{M}_{T,P}(\mathbb{R})$, almacenan los respectivos embeddings proyectados, uno en cada fila, similar a como funciona una matriz de embedding. Luego, notando que $\langle q_i, k_j \rangle = \langle \text{fila}_i(Q), \text{fila}_j(K) \rangle = (QK^\top)_{ij}$, se pueden obtener los scores de atención mediante $s(q_i, k_j) = \left(\frac{QK^\top}{\sqrt{P}} \right)_{ij}$. De esta forma, si $A \in \mathcal{M}_{T,T}(\mathbb{R})$ es la **matriz de atención**, la cual almacena $A_{ij} = a(q_i, k_j)$, la observación anterior permite escribir $A = \text{softmax}_{\text{por filas}} \left(\frac{QK^\top}{\sqrt{P}} \right)$, donde $\text{softmax}_{\text{por filas}}$ indica que se debe aplicar la función softmax en

cada una de las filas de la **matriz de scores** $S = \frac{QK^\top}{\sqrt{P}} \in \mathcal{M}_{T,T}(\mathbb{R})$ (recordar que cada fila debe sumar 1, no cada columna). De este modo, si $Y \in \mathcal{M}_{T,P}(\mathbb{R})$ es la matriz que almacena los embeddings contextuales calculados por el mecanismo de atención, $(y_1, ..., y_T) \subset \mathbb{R}^P$ (un embedding en cada fila), se observa que $Y_{ij} = \sum_{k=1}^{T} a_{ik} v_{kj} = (AV)_{ij}$, por lo que se puede escribir $Y = \text{Attention}(Q, K, V)$, donde

$$\text{Attention}(Q, K, V) := \underset{\text{por filas}}{\text{softmax}}\left(\frac{QK^\top}{\sqrt{P}}\right)V$$

es el mecanismo de atención actuando sobre los embeddings iniciales $(x_1, ..., x_T) \subset \mathbb{R}^D$, cuyas proyecciones en $\mathbb{R}^P$ se almacenan en las matrices $Q, K, V \in \mathcal{M}_{T,P}(\mathbb{R})$.

Por otro lado, las proyecciones $Q, K, V \in \mathcal{M}_{T,P}(\mathbb{R})$ también pueden ser calculadas mediante un producto matricial: si $X \in \mathcal{M}_{T,D}(\mathbb{R})$ es la matriz que almacena en sus filas los embeddings iniciales, $(x_1, ..., x_T) \subset \mathbb{R}^D$, entonces se tiene que $q_{ij} = \left(W_Q x_i\right)_j = \left\langle \text{fila}_j\left(W_Q\right), x_i \right\rangle = \left\langle \text{fila}_j\left(W_Q\right), \text{fila}_i(X) \right\rangle = \left(XW_Q^\top\right)_{ij}$ (análogo para K y V). Por lo tanto, $Q = XW_Q^\top$, $K = XW_K^\top$ y $V = XW_V^\top$, donde $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ son las matrices de proyección definidas anteriormente.

Masked self-attention

Un detalle importante pasado por alto hasta el momento es la pérdida de causalidad al momento de computar los embeddings contextuales $y_1, ..., y_T \in \mathbb{R}^P$. En efecto, dado que el objetivo central sigue siendo construir un modelo neuronal para predecir la probabilidad del próximo token dados los tokens anteriores, cada embedding contextual y_t solo debería poder tener acceso a la información asociada a los tokens $w_1, ..., w_t \in \mathcal{V}$ y no a la información de los tokens siguientes, $w_{t+1}, ..., w_T \in \mathcal{V}$, por lo que se debe forzar que $a\left(x_i, x_j\right) = 0$ para $j > i$.

Si bien se podría sustituir directamente $A_{ij} = 0$ para $j > i$ en la matriz de atención, esto provocaría que $\sum_{j=1}^{T} a\left(q_i, k_j\right) < 1$, por lo que la combinación lineal ya no sería convexa. Una solución que evita este problema es sustituir en la matriz de scores, $S = \frac{QK^\top}{\sqrt{P}}$, cada entrada donde $j > i$ por un valor muy negativo ($-\infty$ en la implementación). De este modo, como $\lim_{s \to -\infty} e^s = 0$, se tendrá que $a\left(q_i, k_j\right) = \frac{e^{s\left(q_i, k_j\right)}}{\sum_{t=1}^{T} e^{s(q_i, k_t)}} = 0$ para $j > i$, y se seguirá cumpliendo la restricción $\sum_{j=1}^{T} a\left(q_i, k_j\right) = 1$ ya que la función softmax es aplicada luego de esta sustitución.

Esta variante de auto-atención se conoce como **masked self-attention**, y es esencial para preservar la causalidad del lenguaje. En la práctica, este tipo de exclusión se implementa construyendo una matriz auxiliar $M \in \mathcal{M}_{T,T}(\mathbb{R})$, llamada máscara, donde las entradas no

nulas de su i-ésima fila indicarán los tiempos $\{1, ..., T\}$ sobre los que podrá poner atención el embedding contextual $x_i \in \mathbb{R}^D$. En este caso, para mantener la causalidad, $M_{ij} = \begin{cases} 1 \text{ si } j \leq i \\ 0 \text{ si } j > i \end{cases}$ (en particular, $M \in \mathcal{M}_{T,T}(\mathbb{R})$ es una matriz triangular inferior). El siguiente diagrama muestra el mecanismo de auto-atención incluyendo el masking causal:

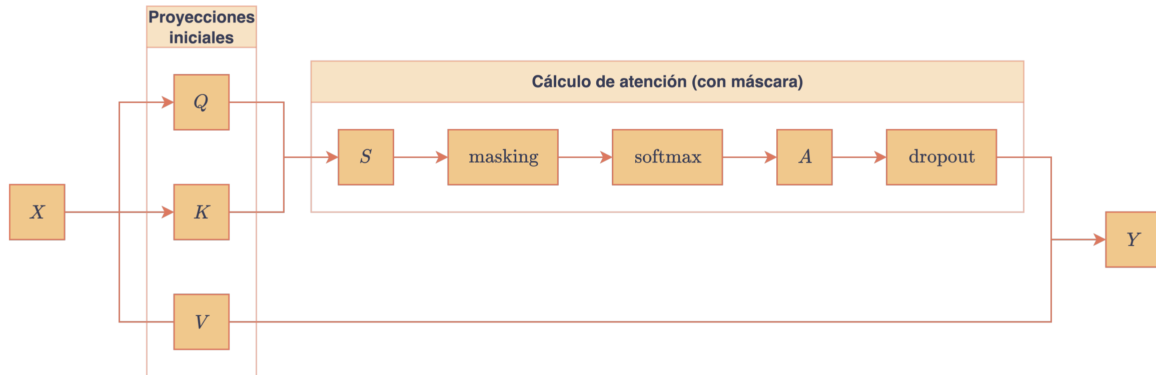

Figura 5: Diagrama del mecanismo de masked self-attention con máscara causal y dropout sobre la matriz de atención.

Notar que se incluyó una capa de dropout [9], la cual es aplicada a la matriz de atención, $A \in \mathcal{M}_{T,T}(\mathbb{R})$. Recordar que para un vector cualquiera, $x \in \mathbb{R}^D$, $\text{Dropout}_p(x) := \frac{1}{1-p} m \odot x$, donde el vector binario $m \in \{0, 1\}^D$ determina sus coordenadas al azar desde $m_d \sim \text{Bernoulli}(1 - p)$. Es decir, $\text{Dropout}_p(x)$ apaga el valor de ciertas coordenadas de x, impidiendo el flujo de información a través de las conexiones neuronales relacionadas con las componentes apagadas. En cambio, el factor de ponderación, $\frac{1}{1-p} \geq 1$ busca compensar el flujo perdido aumentando el valor de las coordenadas que sí quedaron encendidas. En el caso del dropout aplicado a la matriz de atención, el funcionamiento es el mismo: ciertas coordenadas de la matriz $A \in \mathcal{M}_{T,T}(\mathbb{R})$ son sustituidas por 0, mientras que las restantes son ponderadas por $\frac{1}{1-p}$, respetando (en esperanza) el cumplimiento de la restricción $\sum_{j=1}^{T} a(q_i, k_j) = 1$. En el código, el parámetro de dropout $p \in [0, 1]$ se representa por el atributo `config.dropout`, el cual se compartirá de manera global en todas las capas de dropout definidas en el modelo.

Para ejemplificar el uso de dropout, se aplicará dropout (con $p = 0.2$) sobre una matriz cuadrada que simulará ser una matriz de atención (no se aplicará causalidad ni softmax por simplicidad). Notar que los valores que permanecen activos son multiplicados cada uno por $\frac{1}{1-p} = 1.25$:

```
p = 0.2
dropout = nn.Dropout(p)
matrix = torch.ones(5, 5)
matrix = dropout(matrix)
print(matrix)
```

```
tensor([[1.2500, 1.2500, 1.2500, 1.2500, 0.0000], [0.0000, 1.2500,
1.2500, 0.0000, 1.2500], [1.2500, 0.0000, 1.2500, 1.2500, 0.0000],
[1.2500, 1.2500, 0.0000, 1.2500, 1.2500], [0.0000, 1.2500, 1.2500,
1.2500, 1.2500]])
```

La motivación usual de la técnica de dropout es que el modelo aprenda a predecir la respuesta correcta aún cuando no tiene acceso a toda la información. En particular, para la matriz de atención, si $a(q_i, k_j) = 0$, entonces el embedding contextual $y_i \in \mathbb{R}^P$ (asociado al token $w_i \in S$) no podrá obtener información contextual acerca del token $w_j \in S$, independientemente de si $j \leq i$ o no. Dado que las posiciones donde ocurre el dropout se eligen de forma aleatoria en cada secuencia e iteración de entrenamiento, aplicar esta regularización en la matriz de atención fuerza a que el modelo tenga que aprender a utilizar la información de todas las posiciones del contexto para realizar una predicción. Durante la inferencia (i.e., cuando el modelo ya está entrenado y se invocó `model.eval()`), el dropout se deshabilita y todos los tokens del contexto son utilizados para el cálculo de los embeddings contextuales.

La siguiente clase implementa el mecanismo de masked self-attention. Además, aunque no es necesario hacerlo, se guardará la matriz de atención en el atributo `attn_matrix` para poder visualizarla posteriormente cuando el modelo esté en modo inferencia (recordar que el atributo `training` se hereda de la clase `nn.Module`, e indica si el modelo está en modo entrenamiento o en modo inferencia).

```
class SelfAttentionHead(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()

        self.query = nn.Linear(cfig.embedding_dim, cfig.head_dim,
bias=False)
        self.key = nn.Linear(cfig.embedding_dim, cfig.head_dim, bias=False)
        self.value = nn.Linear(cfig.embedding_dim, cfig.head_dim,
bias=False)

        self.dropout = nn.Dropout(cfig.dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:

        Q = self.query(x)
        K = self.key(x)
        V = self.value(x)
```

```python
    batch_size, seq_length, head_dim = Q.shape
    mask = torch.tril(torch.ones(seq_length, seq_length)).to(x.device)

    attn = Q @ K.transpose(-1, -2) / (head_dim ** 0.5)
    attn = attn.masked_fill(mask == 0, float('-inf'))
    attn = self.dropout(attn.softmax(dim=-1))

    if self.training == False:
        self.attn_matrix = attn.detach()

    return attn @ V
```

- Ejemplo de uso.

```python
sa = SelfAttentionHead(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = sa(x)

assert y.shape == (batch_size, seq_length, config.head_dim)
```

Atención multicabezal

El objetivo del mecanismo de auto-atención definido anteriormente es que cada embedding contextual $y_t \in \mathbb{R}^P$ pueda resumir de mejor forma la información contenida en el vector de embedding inicial, $x_t \in \mathbb{R}^D$, haciendo uso del contexto en el que se encuentra el token $w_t \in S$ asociado al embedding x_t (i.e., usando información de los tokens vecinos). Este mecanismo puede repetirse varias veces de forma independiente sobre una misma secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$, donde cada instancia del mecanismo de auto-atención, que se suele llamar **cabezal de atención**, tendrá sus propias matrices de proyección $W_Q, W_K, W_V \in \mathcal{M}_{P,D}(\mathbb{R})$ y, en consecuencia, su propia matriz de atención y embeddings finales, $(y_1, ..., y_T) \subset \mathbb{R}^P$. La motivación de usar varios cabezales de atención es que cada cabezal puede aprender a extraer una información contextual diferente, de forma similar a como actúan los filtros en una red convolucional.

Para una secuencia de embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$ y $H \geq 1$ cabezales de atención independientes actuando sobre esta secuencia, se obtendrán H nuevas secuencias de embeddings, $\left(y_1^{(h)}, ..., y_T^{(h)}\right) \subset \mathbb{R}^P$, con $h \in \{1, ..., H\}$. Con esto, se pueden combinar todos los cabezales para formar una secuencia de embeddings $(\tilde{y}_1, ..., \tilde{y}_T) \subset \mathbb{R}^{P \cdot H}$ de mayor dimensión concatenando los embeddings de todos los cabezales:

$$\tilde{y}_t = \left(y_t^{(1)} \cdots y_t^{(H)}\right) \in \mathbb{R}^{P \cdot H}, \quad \text{para} \;\; t \in \{1, ..., T\}.$$

Además, para poder relacionar los embeddings obtenidos por cada cabezal, cada embedding extendido $\tilde{y}_t$ suele ser pasado por una capa lineal extra, lo que permite, además, recuperar la dimensión de los embeddings originales, $(x_1, ..., x_T) \subset \mathbb{R}^D$. Por lo tanto, los embeddings finales que retorna un bloque de atención multicabezal, $(u_1, ..., u_T) \in \mathbb{R}^D$, son de la forma $u_t = W_0\, \tilde{y}_t \in \mathbb{R}^D$, donde la matriz $W_0 \in \mathcal{M}_{D,P\cdot H}(\mathbb{R})$ proyecta cada embedding concatenado, $\tilde{y}_t \in \mathbb{R}^{P\cdot H}$, de vuelta al espacio $\mathbb{R}^D$. El siguiente diagrama resume esta idea:

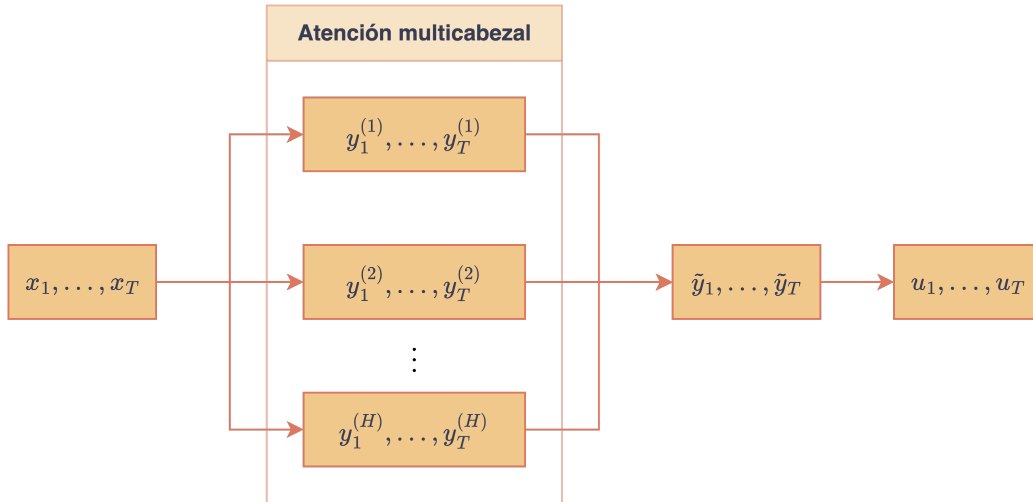

Figura 6: Diagrama del mecanismo de atención multicabezal con concatenación y proyección lineal final.

En la práctica (y en la clase de hiperparámetros `GPTConfig`) es usual considerar como dimensión de cada cabezal a $P = \frac{D}{H}$, donde se asume que $H \mid D$ (i.e., D es divisible por H). La idea de esto es considerar que la dimensión de embedding $D \in \mathbb{N}$ se reparte entre los $H \in \mathbb{N}$ cabezales de atención. En la siguiente implementación, H es representado por el atributo `num_head`:

```
class MultiHeadSelfAttention(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.heads = nn.ModuleList([SelfAttentionHead(cfig) for _ in
range(cfig.num_heads)])
        self.projection = nn.Linear(cfig.num_heads * cfig.head_dim,
cfig.embedding_dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        output = torch.cat([head(x) for head in self.heads], dim=-1)
        output = self.projection(output)
        return output
```

- Ejemplo de uso.

```
mhsa = MultiHeadSelfAttention(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = mhsa(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

Por otra parte, es posible computar los H cabezales de atención en paralelo aplicando el mecanismo de atención sobre matrices Q, K y V más grandes. Aquí no se realizó dicha paralelización por simplicidad en la implementación, pero en la práctica sí se debería realizar para mayor eficiencia.

3.3.3 Red feedforward

El último módulo que compone un bloque Transformer de la arquitectura GPT está enfocado en pasar cada uno de los embeddings obtenidos por la atención multicabezal, $(u_1, ..., u_T) \subset \mathbb{R}^D$, por una misma red fully connected de dos capas. La idea de esta transformación final es darle una mayor capacidad a la red neuronal mediante un post- -procesamiento individual de cada embedding obtenido anteriormente en el mecanismo de atención (cuya función principal era combinar los distintos tokens de la secuencia entre sí). Para cada $u_t \in \mathbb{R}^D$, $t \in \{1, ..., T\}$, el módulo feed forward calcula

$$\mathrm{FF}(u_t) = W_2\, \phi(W_1\, u_t),$$

donde $W_1 \in \mathcal{M}_{J,D}(\mathbb{R})$, $W_2 \in \mathcal{M}_{D,J}(\mathbb{R})$ son los parámetros de la red (con $J \in \mathbb{N}$ la dimensión interna del bloque feed forward) y $\phi : \mathbb{R} \to \mathbb{R}$ es una función de activación que se aplica coordenada a coordenada. El siguiente diagrama resume el uso de la red feed forward para una secuencia $(u_1, ..., u_T) \subset \mathbb{R}^D$ de embeddings, donde se ha denotado por $(r_1, ..., r_T) \subset \mathbb{R}^D$ los vectores de salida:

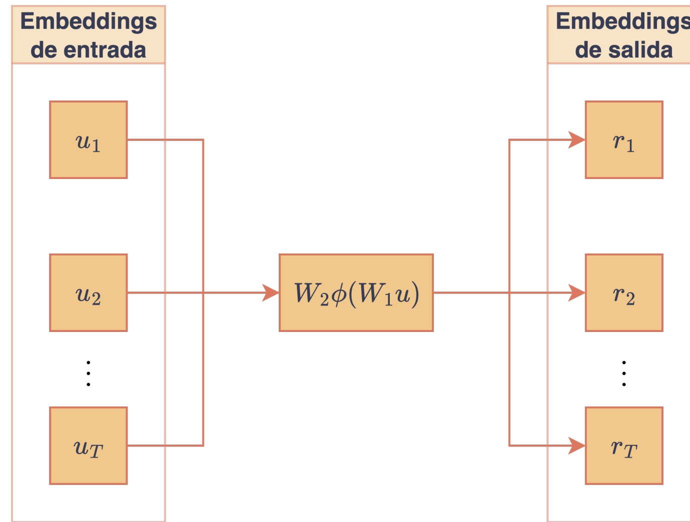

Figura 7: Diagrama de la red feedforward aplicada de forma independiente a cada embedding de la secuencia.

Notar que las dos capas lineales, al igual que todas las otras capas lineales usadas en esta arquitectura, no incluyen un término de sesgo, lo cual se ha vuelto una práctica común en los LLMs y arquitecturas neuronales modernas. Además, es usual considerar $J \gg D$ para darle más capacidad de representación a la red. Más específicamente, J suele ser un múltiplo de D (e.g., $J = 4D$), lo cual será indicado en el hiperparámetro `cfig.ff_factor`. Por otro lado, en la implementación se utilizará GELU [10] como función de activación (i.e., $\phi(x) = x \cdot \Phi(x)$, donde $\Phi : \mathbb{R} \to [0, 1]$ es la distribución acumulada de una gaussiana estándar), lo que es una práctica común para este tipo de modelos, cuya elección, hasta donde tengo entendido, está basada únicamente en resultados empíricos. La siguiente clase `FeedForward` implementa este módulo:

```
class FeedForward(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.fc1 = nn.Linear(cfig.embedding_dim, cfig.ff_factor *
cfig.embedding_dim, bias=False)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(cfig.ff_factor * cfig.embedding_dim,
cfig.embedding_dim, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.fc1(x)
        x = self.activation(x)
```

```
    x = self.fc2(x)
    return x
```

- Ejemplo de uso

```
ff = FeedForward(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = ff(x)

assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

3.3.4 Bloque Transformer

Los 3 módulos definidos anteriormente, `LayerNorm`, `MultiHeadSelfAttention` y `FeedForward`, permiten construir el bloque principal de la arquitectura GPT, llamado **bloque Transformer**, el cual es repetido varias veces en la arquitectura GPT, dependiendo de la profundidad de la arquitectura.

Dada una secuencia de embeddings, $(x_1, ..., x_T) \subset \mathbb{R}^D$, almacenada en una matriz $X \in \mathcal{M}_{T,D}(\mathbb{R})$ (un vector de embedding en cada fila), un bloque Transformer entrega otra secuencia de embeddings mejorados (por el mecanismo de atención y luego la red feed forward) en una matriz de salida del mismo tamaño, $\text{TransformerBlock}(X) \in \mathcal{M}_{T,D}(\mathbb{R})$:

$$\begin{cases} Y = X + \text{Dropout}_p\big(\text{MHSA}\big(\text{LayerNorm}_{\text{por filas}}(X)\big)\big) \\ \text{TransformerBlock}(X) = Y + \text{Dropout}_p\big(\text{FF}_{\text{por filas}}\big(\text{LayerNorm}_{\text{por filas}}(Y)\big)\big) \end{cases}.$$

- La función $\text{LayerNorm}_{\text{por filas}}$ indica que se debe aplicar el módulo `LayerNorm` de forma independiente en cada fila de las matrices X e Y (i.e., se aplica en cada uno de los vectores de embedding).
- La función MHSA corresponde a aplicar el módulo `MultiHeadSelfAttention` a la secuencia de embeddings normalizados contenida en la matriz $\text{LayerNorm}_{\text{por filas}}(X) \in \mathcal{M}_{T,D}(\mathbb{R})$.
- La función $\text{FF}_{\text{por filas}}$ corresponde a aplicar el módulo `FeedForward` a cada uno de los embeddings normalizados contenidos en la matriz $\text{LayerNorm}_{\text{por filas}}(Y) \in \mathcal{M}_{T,D}(\mathbb{R})$.

Notar que, en la práctica, gracias al broadcasting automático realizado por PyTorch, las operaciones por filas se obtienen de forma automática al pasar las matrices de embeddings por los módulos `LayerNorm` y `FeedForward`. Sin embargo, se debe tener presente que dichos módulos actúan siempre de forma independiente sobre cada posición de la secuencia de embeddings.

En resumen, un bloque Transformer no es más que aplicar el mecanismo de atención multicabezal, seguido de una red feed forward. Para ambos módulos, se aplica anteriormente una

capa de normalización (para estabilizar la entrada al respectivo módulo) y posteriormente una capa de dropout (como una técnica de regularización). Además, en ambos módulos se realiza una **conexión residual** (i.e., se suma la entrada a la salida de cada módulo). En el próximo capítulo se realizará una revisión más detallada de estas decisiones de diseño y de sus efectos en la arquitectura.

El siguiente diagrama resume el funcionamiento de un bloque Transformer:

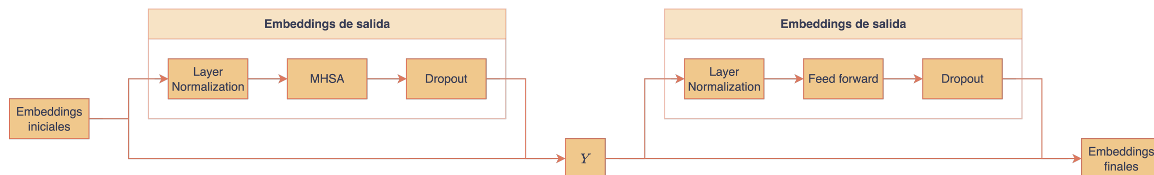

Figura 8: Diagrama de un bloque Transformer con self-attention multicabezal, red feedforward, layer normalization, dropout y conexiones residuales.

Las capas de dropout aplicadas posterior a la atención multicabezal y a la red feedforward tienen el mismo rol que antes: descartar algunas componentes de algunos embeddings de forma aleatoria. Notar que el dropout aplicado en estas capas no descarta tokens completos como ocurre al usar masking en los scores de atención, donde el masking de un parámetro $a(q_i, k_j) \in [0, 1]$ no permite que el i-ésimo token coloque atención sobre el j-ésimo token.

La siguiente clase `TransformerBlock` implementa un bloque Transformer usando los módulos implementados anteriormente:

```
class TransformerBlock(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()
        self.norm1 = LayerNorm(cfig.embedding_dim)
        self.mhsa = MultiHeadSelfAttention(cfig)
        self.norm2 = LayerNorm(cfig.embedding_dim)
        self.ff = FeedForward(cfig)
        self.dropout = nn.Dropout(cfig.dropout)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = x + self.dropout(self.mhsa(self.norm1(x)))
        x = x + self.dropout(self.ff(self.norm2(x)))
        return x
```

- Ejemplo de uso.

```
transformer_blocks = TransformerBlock(config)
x = torch.randn(batch_size, seq_length, config.embedding_dim)
y = transformer_blocks(x)
```

```python
assert y.shape == (batch_size, seq_length, config.embedding_dim)
```

3.3.5 Clase principal

El bloque Transformer definido anteriormente es el componente principal de la arquitectura GPT (y de las arquitecturas tipo Transformer en general). Este bloque es repetido varias veces para aumentar la complejidad del modelo, de forma análoga a como se repiten los bloques ResNet [11] en una red convolucional. Sin embargo, es necesario agregar elementos auxiliares antes y después de la ejecución de los bloques Transformer:

- **Capa de embedding:** todos los módulos implementados hasta el momento (en particular, el bloque Transformer) esperan recibir una secuencia de vectores de embedding $(x_1, ..., x_T) \subset \mathbb{R}^D$ y no directamente una secuencia de tokens $(w_1, ..., w_T) \in \mathcal{V}^*$. Por lo tanto, al igual que en la RNN anterior, se debe anteponer una capa de embedding $\text{emb} : \mathcal{V} \to \mathbb{R}^D$ que transforme esta secuencia de símbolos discretos en vectores que la red neuronal pueda procesar.
- **Positional encoding:** dado que las RNNs procesan los tokens de una secuencia uno a uno, esto le entrega información al modelo acerca del orden de los respectivos tokens dentro de la secuencia. Sin embargo, los bloques Transformer no reciben, al menos de forma explícita, ninguna información acerca de la posición dentro de la secuencia que tienen los embeddings $(x_1, ..., x_T) \subset \mathbb{R}^D$ que va a procesar. Una posible opción para inyectar explícitamente esta información es agregar una segunda capa de embedding (similar a la que codifica cada símbolo del vocabulario $\mathcal{V}$ en un vector de embedding) que codifique cada posición de la secuencia, $(1, 2, ..., T) \subset \mathbb{N}$, en un vector de embedding, $\text{emb}(t) \in \mathbb{R}^D$. De esta forma, el embedding final que entra al primer bloque Transformer consistirá en sumar los embeddings asociados a los tokens de la secuencia (**token embeddings**) con los embeddings asociados a la posición absoluta de cada token dentro de la secuencia (**positional embeddings**). Notar que el uso de una capa de embedding para la posición automáticamente limita el largo máximo de las secuencias (según la cantidad de filas que tenga la matriz de embedding respectiva). Este valor máximo se conoce como **ventana de contexto**.
- **Cabezal de lenguaje:** luego de aplicar consecutivamente los bloques Transformer a las secuencias de embeddings que se van mejorando iterativamente, es necesario agregar un cabezal de lenguaje análogo al utilizado en la RNN, el cual transforma la salida del último bloque Transformer en un vector de probabilidades sobre $\mathcal{V}$, lo cual permitirá utilizar la arquitectura GPT para aprender el vector de probabilidades asociado a la distribución del próximo token dados los tokens de entrada al modelo.

El siguiente diagrama resume la arquitectura GPT completa:

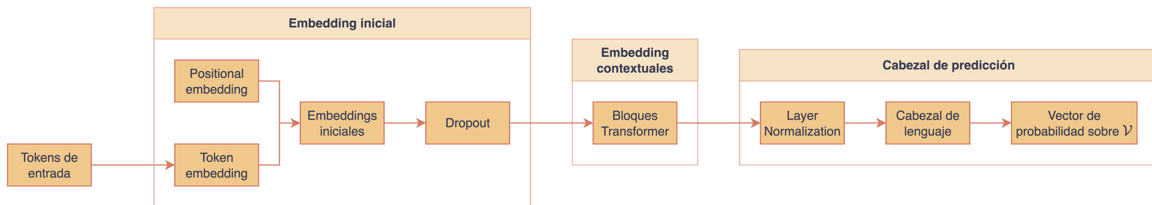

Figura 9: Diagrama de la arquitectura GPT completa [1].

Notar que se agregó una capa de dropout antes de pasar la secuencia de vectores de embedding al primer bloque Transformer. El objetivo de esta capa es, al igual que antes, forzar a que el modelo aprenda a usar todas las posiciones de la secuencia y todas las dimensiones de los vectores de embedding. Adicionalmente, se agregó una capa de normalización previo a la proyección realizada por el cabezal de lenguaje, lo cual permite estabilizar el entrenamiento.

La siguiente clase `GPT` implementa la arquitectura GPT completa utilizando los módulos implementados anteriormente:

```
class GPT(nn.Module):

    def __init__(self, cfig: GPTConfig) -> None:
        super().__init__()

        self.token_embedding = nn.Embedding(cfig.vocab_size,
cfig.embedding_dim)
        self.position_embedding = nn.Embedding(cfig.context_window,
cfig.embedding_dim)
        self.dropout = nn.Dropout(cfig.dropout)
        self.transformer_blocks = nn.Sequential(*[TransformerBlock(cfig)
for _ in range(cfig.num_layers)])
        self.norm_output = LayerNorm(cfig.embedding_dim)
        self.lm_head = nn.Linear(cfig.embedding_dim, cfig.vocab_size,
bias=False)

        self.context_window = cfig.context_window

    def forward(self, x: torch.Tensor) -> torch.Tensor:

        x = x[:, -self.context_window:]
        batch_size, seq_length = x.shape

        token_embedding = self.token_embedding(x.long())
        positional_embedding =
self.position_embedding(torch.arange(seq_length, device=x.device))
```

```python
        output = token_embedding + positional_embedding
        output = self.dropout(output)
        output = self.transformer_blocks(output)
        output = self.norm_output(output)
        output = self.lm_head(output)
        return output
```

- Ejemplo de uso.

```python
gpt = GPT(config)
x = torch.randint(config.vocab_size, size=[batch_size, seq_length])
y = gpt(x)

assert y.shape == (batch_size, seq_length, config.vocab_size)
```

3.3.6 Entrenamiento y generación

Para entrenar y generar nuevas secuencias de texto utilizando este modelo se puede reutilizar sin ningún cambio el código implementado anteriormente para la RNN. Esto muestra que las RNNs y la arquitectura GPT son similares en su objetivo, solo que procesan internamente las secuencias de forma diferente. En este caso, los hiperparámetros de arquitectura que se utilizarán serán los definidos en la clase `GPTConfig`:

```python
gpt = GPT(config)

n_params = sum(param.numel() for param in gpt.parameters()) / 1e6
print(f'Cantidad de parámetros: {n_params:.3} millones.')
```

> Cantidad de parámetros: 57.0 millones.

Para comparar de forma justa las curvas de entrenamiento del modelo GPT con la curva de la RNN, este modelo también se entrenará durante 32 épocas, y utilizando el mismo optimizador y dataloader:

```python
gpt_optimizer = optim.AdamW(gpt.parameters())
train_model(gpt, gpt_optimizer, dataloader, epochs=32,
ckpt_filename='gpt_training.pt')
```

El entrenamiento de este modelo toma aproximadamente 7 minutos en Google Colab (usando, igual que antes, una GPU NVIDIA Tesla T4). Ahora se cargará el modelo y se verán las curvas de entrenamiento de ambos modelos:

```
rnn_training = torch.load('rnn_training.pt', DEVICE, weights_only=True)
gpt_training = torch.load('gpt_training.pt', DEVICE, weights_only=True)
gpt.load_state_dict(gpt_training['model'])

plt.figure(figsize=(10, 5))
plt.plot(rnn_training['losses'], label='RNN')
plt.plot(gpt_training['losses'], label='GPT')
plt.xlabel('Iteración')
plt.ylabel('Entropía cruzada')
plt.grid(alpha=0.3)
plt.legend()
plt.show()
```

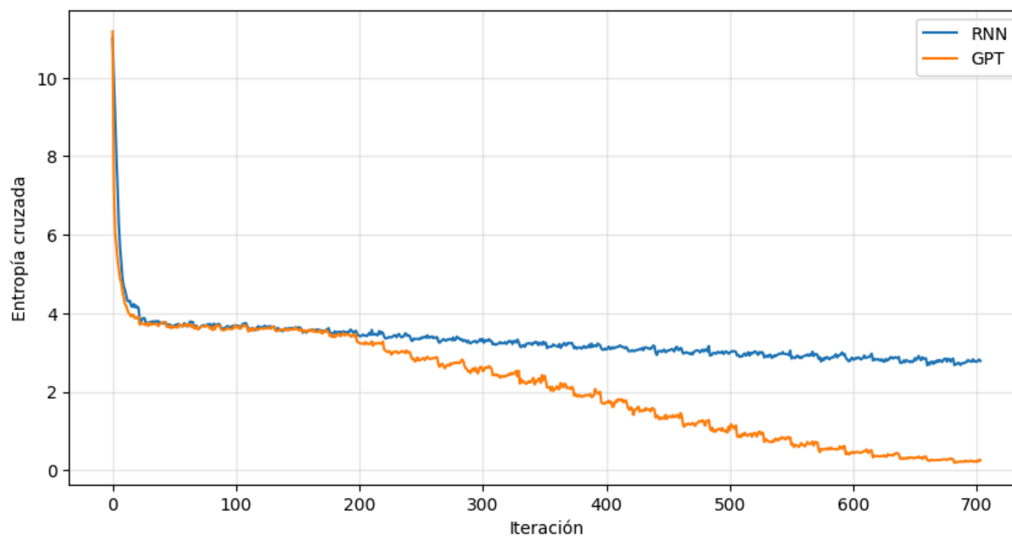

Figura 10: Comparación de las curvas de entropía cruzada durante el entrenamiento de la RNN y la arquitectura GPT.

Se observa que la complejidad de la arquitectura GPT permite obtener una entropía cruzada mucho más baja que con la RNN. Un detalle importante aquí es que al no estar midiendo la función de pérdida en un conjunto de test (i.e., en un conjunto i.i.d con respecto al conjunto de entrenamiento), no es posible ver si el entrenamiento está realizando overfitting o no.

Para la generación, se usará el mismo código de antes (con los mismos parámetros por defecto):

```
context = 'Habia una vez'

for n in range(10):
```

```python
new_tokens = generate_tokens(gpt, context, dataset.tokenizer)
print(new_tokens)
```

```
Habia una vez un matrimonio que tenia un solo hijo. El hombre sembro
la mas hermosa papa en una tierra que estaba lejos de la casa que
habitaban. Habia una vez un matrimonio que tenia un niño hijo.
El hombre sembro la mas hermosa papa de una tierra. En la papa
me miraba porque levantaba la tierra. Una la ciudad habia dibujo
sombra, tenia, pero se excelsa clase de semilla. Habia una vez un
espejo de mano que cuando se quedaba solo y nadie se veia en el
se sentia de lo peor, como que no existia, y quiza tenia razon
pero los otros espejos se burlaban.. Habia una vez un matrimonio
que tenia un solo tenia verano. En esa lo esa hermosa casi por un
publica. En un vuelo un espejo de las casas saludado, Yo me quedaba
edad, habian instalado un persona, cuando. Habia una vez un espejo
de mano que cuando se quedaba solo y nadie se veia en el, razon
por su cual la cual que tenia existia, y quiza tenia razon pero
los otros espejos se burlaban de el, y, se asesorados, noches, el,
en el mismo cajon del, el. Una pierna suelta satisfechos, ajenos a
la, del alma, salir, que, un momento, era niño lo vio. Habia una
vez un niño llamado David N. El niño que niño sembro y y hermosa
y vacias, un mediano. El una muerte en la casa de la casa en la
espalda Habia una vez una viuda de buen pasar, que tenia una hija.
La muchacha era hermosa y la madre queria casarla con un hombre bien
rico. Se presentaron algunos pretendientes, pero hombres honrados,
trabajadores y acomodados, pero, y, los despedia con su musica a
otra parte, Por pelo, riquisimos….. Habia una vez un niño llamado
David N. Me conversaciones El niño Chapachapa, y en la la resortera
despertaba y capataz y admiracion en sus amigos el y vacas y resortera
la escuela, que. Habia una vez un matrimonio que tenia un solo hijo.
El hombre sembro la mas hermosa papa en una tierra que estaba lejos
de la casa. En la casa, Acababa la arboles miraba la sombra, sola.
Pero con casa se le le madre llamaron.. Habia una vez un niño llamado
David N., cuya punteria y habilidad en el manejo de la resortera
despertaba tanta envidia y admiracion, y me admiracion y la casa,
y, puño, el azar madre, firmeza y su excelsa vacias entre, con su
musica, no llamaba…
```

Se observa que el modelo, a diferencia de la RNN anterior, es capaz de generar texto con un poco más de sentido. Sin embargo, resulta ser que gran parte del texto generado corresponde

a fragmentos exactos del dataset usado para entrenar (ver archivo `data.txt`), mostrando que el modelo, si bien tiene la capacidad para aprender a generar secuencias, se sobreajustó fuertemente al conjunto de entrenamiento. Este es un comportamiento esperado en este caso ya que solo se utilizaron un poco más de 700 secuencias para entrenar el modelo. Además, al haber realizado 32 iteraciones sobre el conjunto de entrenamiento, el sobreajuste es casi inevitable (considerar que muchos LLMs realizan por lo general 1 o 2 iteraciones sobre cada dataset de entrenamiento).

Por otro lado, se observa que el tiempo de inferencia es bastante lento. Esto se debe a que el modelo es llamado cada vez que se quiere generar un nuevo token ya que, si bien el entrenamiento es paralelizable, la generación no lo es. Notar, además, que el atributo `attn_matrix` definido en el forward de la clase `SelfAttentionHead` (el cual solo se define cuando el modelo está en modo inferencia) guarda las matrices de atención en cada llamada al modelo, lo que hace aún más lenta la inferencia. En una implementación estándar no es necesario guardar las matrices (y de hecho, no se realiza para evitar el deterioro en la eficiencia), pero aquí se incluyó con fines didácticos.

Visualización de las matrices de atención

En este apartado se visualizarán las matrices de atención `attn_matrix` que se calculan cuando el modelo GPT procesa una secuencia. Si bien esto no es necesario en la práctica, permite ver cómo el modelo está procesando la entrada, entregando propiedades de interpretabilidad a la arquitectura.

La siguiente función `show_attn_matrix` grafica los valores de las matrices de atención de todos los cabezales de atención de un bloque, para una secuencia de entrada dada:

```python
def show_attn_matrix(model: GPT, text: str, block: int) -> None:

    tokens = re.findall(r'\d|[^\w\s]|\w+|\s', text)
    token_ids = dataset.tokenizer.encode(text)
    x = torch.tensor(token_ids, device=DEVICE).unsqueeze(0)

    model.eval()
    _ = model(x)

    heads = model.transformer_blocks[block].mhsa.heads
    attn_matrices = [head.attn_matrix[0] for head in heads]

    fig, axes = plt.subplots(2, len(heads) // 2, figsize=(14, 9))
    fig.suptitle(f'Matrices de atención (block={block})')

    for head, ax in enumerate(axes.flatten()):
```

```
        ax.imshow(attn_matrices[head])
        ax.set_xticks(ticks=range(len(tokens)), labels=tokens)
        ax.set_yticks(ticks=range(len(tokens)), labels=tokens)
        ax.set_title(f'Head {head + 1}')

    plt.tight_layout()
    plt.show()
```

Para el 3º bloque Transformer del modelo entrenado:

```
text = 'Habia una vez, en un pueblo muy lejano'
show_attn_matrix(gpt, text, block=2)
```

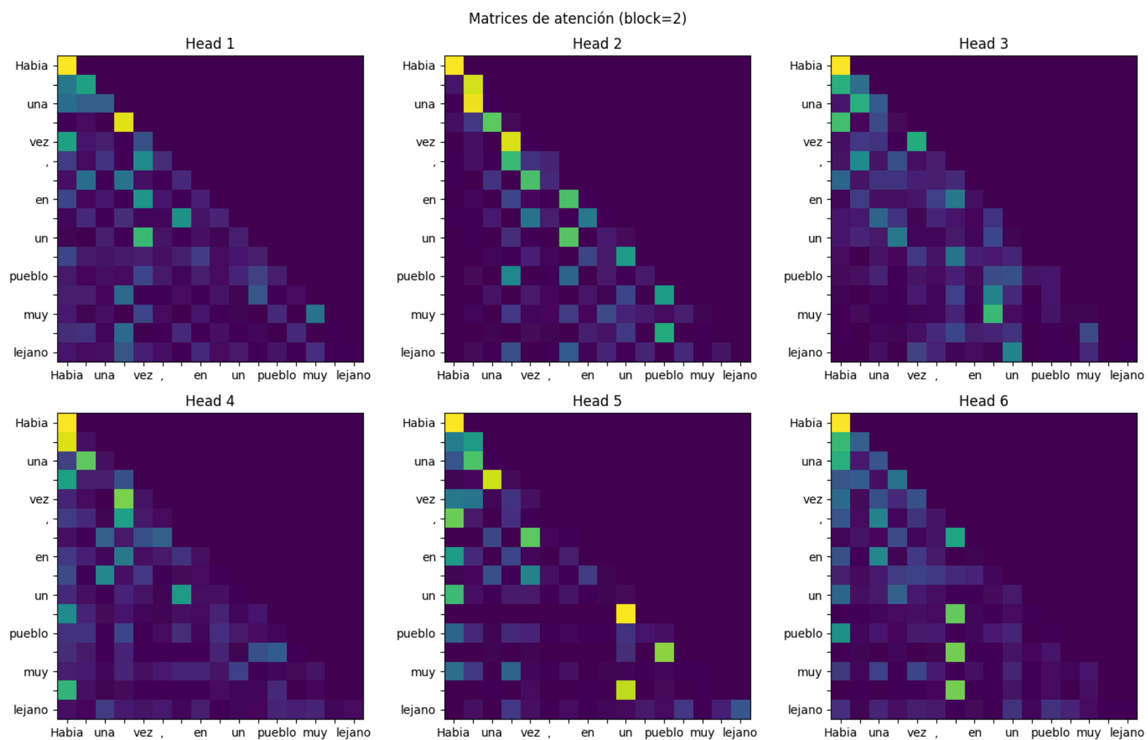

Figura 11: Matrices de atención del tercer bloque Transformer para la secuencia de entrada, donde se observa la estructura triangular inferior inducida por la máscara causal.

Se observa que todas las matrices son triangulares inferior, lo que es consistente con la causalidad impuesta en el modelo. En entradas posteriores se revisarán más en detalle algunas propiedades conocidas acerca de estas matrices de atención.

El siguiente capítulo revisará algunas variantes de la arquitectura GPT y de la arquitectura Transformer en general, las cuales han permitido extender aún más los límites de este tipo de modelos en las arquitecturas modernas. Además, se revisarán otro tipo de modelos, no

necesariamente autorregresivos, que pueden aprovechar la arquitectura Transformer para otro tipo de tareas como el procesamiento de imágenes o tareas discriminativas en general.

Referencias

[1] A. Radford, K. Narasimhan, T. Salimans, y I. Sutskever, «Improving Language Understanding by Generative Pre-Training», technical report, 2018. [En línea]. Disponible en: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[2] J. L. Elman, «Finding Structure in Time», *Cognitive Science*, vol. 14, n.º 2, pp. 179-211, 1990, [En línea]. Disponible en: https://onlinelibrary.wiley.com/doi/10.1207/s15516709cog1402_1

[3] I. Loshchilov y F. Hutter, «Decoupled Weight Decay Regularization», en *International Conference on Learning Representations (ICLR)*, 2019. [En línea]. Disponible en: https://arxiv.org/abs/1711.05101

[4] A. Vaswani *et al.*, «Attention Is All You Need», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [En línea]. Disponible en: https://arxiv.org/abs/1706.03762

[5] S. Ioffe y C. Szegedy, «Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift», en *International Conference on Machine Learning (ICML)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1502.03167

[6] Y. Wu y K. He, «Group Normalization», en *European Conference on Computer Vision (ECCV)*, 2018. [En línea]. Disponible en: https://arxiv.org/abs/1803.08494

[7] J. L. Ba, J. R. Kiros, y G. E. Hinton, «Layer Normalization», *arXiv preprint arXiv:1607.06450*, 2016, [En línea]. Disponible en: https://arxiv.org/abs/1607.06450

[8] D. Bahdanau, K. Cho, y Y. Bengio, «Neural Machine Translation by Jointly Learning to Align and Translate», en *International Conference on Learning Representations (ICLR)*, 2015. [En línea]. Disponible en: https://arxiv.org/abs/1409.0473

[9] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, y R. Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks from Overfitting», *Journal of Machine Learning Research*, vol. 15, pp. 1929-1958, 2014, [En línea]. Disponible en: https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

[10] D. Hendrycks y K. Gimpel, «Gaussian Error Linear Units (GELUs)», *arXiv preprint arXiv:1606.08415*, 2016, [En línea]. Disponible en: https://arxiv.org/abs/1606.08415

[11] K. He, X. Zhang, S. Ren, y J. Sun, «Deep Residual Learning for Image Recognition», en *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016. [En línea]. Disponible en: https://arxiv.org/abs/1512.03385