Universidad Zaragoza
1542

Trabajo Fin de Grado

Implementación y condicionamiento de la herramienta Museformer para composicón automática de obras musicales

Conditioning and implementation of the Museformer tool for automatic generation of musical scores

Autor

Alexandru Cosmin Lancrajan

Director

Jose Ramón Beltrán Blázquez

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2024

AGRADECIMIENTOS

Agradezco a Jose Ramón por la proposición de este trabajo ya que combina la música que es uno de mis tópicos favoritos con un tema tan interesante, actual y sujeto aún a mejora como es la IA. También quiero agradecer a la EINA por haber puesto a mi disposición los recursos necesarios para el desarrollo de este trabajo de fin de grado, ya que sin estos medios hubiera resultado difícil hacer parte del mismo con las herramientas personales de las que disponía. Por último me gustaría incluir a Yann LeCun y Alfredo Canziani de la NYU por haber dispuesto de manera gratuita el curso de Deep Learing que imparten en dicha universidad, ya que ha sido de gran ayuda a mi formación y entendimiento en general dentro del mundo del Deep Learning, y en particular para la herramienta utilizada en este trabajo.

Museformer

RESUMEN / ABSTRACT

Las redes neuronales han experimentado un desarrollo importante en los últimos años, especialmente en el procesado natural del lenguaje mediante el uso de una arquitectura llamada Transformer, la cual ha impulsado el desarrollo de herramientas como Chat-GPT o BERT Large. Dado que la música se puede representar en formato MIDI, y consecuentemente en tokens, el *equipo de Microsoft Asia* desarrolló Museformer basándose en la arquitectura Transformer y realizando unas variaciones dentro de la misma para poder componer música de forma más precisa mediante el uso de dos tipos de mecanismos de atención, fino y grueso. La atención fina se encarga de relacionar compases dentro de la canción y la atención gruesa de generar variación dentro de la pieza, además de reducir la complejidad computacional del mecanismo de atención. Mediante un entrenamiento en una base de datos MIDI amplia, se pueden inferir canciones a partir de ruido en la entrada. Para finalizar se describe de forma teórica cómo debería implementarse un condicionamiento a la red para poder introducir texto en la entrada en vez de ruido para indicarle el estilo, género, artista, etc.

Neural networks have experienced significant development in recent years, especially in natural language processing using an architecture called Transformer, which has driven the development of tools such as Chat-GPT or BERT Large. Since music can be represented in MIDI format, and consequently in tokens, the Microsoft Asia team developed Museformer based on the Transformer architecture and made some variations within it in order to compose music more accurately by using two types of attention mechanisms, fine and coarse. The fine attention is in charge of relating measures within the song and the coarse attention is in charge of generating variation within the piece, in addition to reducing the computational complexity of the attention mechanism. By training on a large MIDI database, songs can be inferred from random noise given to the input layer. Finally, we describe theoretically how a network conditioning should be implemented to be able to introduce input text instead of noise to indicate style, genre, artist, etc.

Índice

Capítulo 1

Motivación y objetivos del trabajo

El auge de las redes neuronales en los últimos años ha generado una importante mejora tecnológica en diversidad de ámbitos como pueden ser la robótica, la visión por computador, la conducción, los motores de búsqueda Web, la traducción, el lenguaje natural, etc. En particular en los últimos dos años la arquitectura Transformer ha provocado una importante mejora dentro del Procesado Natural del Lenguaje (*NLP en inglés*) con la llegada de Chat-GPT, BERT y otros modelos de lenguaje adicionales. Esto ha provocado que diversos investigadores buscaran modelar la música de forma que encajara con esta arquitectura ya que por una parte la música admite representación mediante tokens[1] que se comportan como elementos que son reconocidos por la red, y por otra parte la música se puede entender como un lenguaje con sus elementos básicos y relaciones entre ellos.

Una de los modelos propuestos para generar composiciones musicales simbólicas de forma automática es Museformer por parte del equipo de *Microsoft Asia*. Su explicación se encuentra detallada en la sección 4. Este modelo resuelve los problemas que tienen los modelos Transformer para secuencias largas, que es el coste de computación, ya que la capa de atención tiene un orden $\mathcal{O}(n^2)$ para la secuencia de entrada, y la relación entre la propia estructura musical de las obras, ya que existen compases relacionados que se encuentran muy lejos entre ellos, y también hay compases que no tienen ninguna relación.

El objetivo de este trabajo es implementar el modelo Museformer[2] para poder entrenarlo y hacer inferencia a partir del mismo. Como paso adicional y si hay tiempo condicionarlo en el sentido de poder introducir datos de entrada para guiar al modelo a la hora de generar piezas, por ejemplo, si queremos componer una pieza de Pop o Rock se lo indicaremos como texto de entrada.

Para lograr este objetivo principal se tiene que seguir una serie de pasos u objetivos

[1] Explicados en su respectiva sección. (4)

[2] Repositorio Github: `https://github.com/microsoft/muzic`

secundarios.

1. Entendimiento de la arquitectura Transformer (3.4).

2. Creación de un entorno para la implementación de Museformer (4).

3. Aprender a clonar un repositorio de GitHub[3].

4. Ejecutar los distintos archivos descargados.

5. Comprensión de los parámetros de entrenamiento.

6. Comprensión de los parámetros de inferencia y el paso de generación de la red.

7. Posible condicionamiento de la red para añadirle funcionalidad extra.

La memoria se divide en 7 apartados (incluyendo este) que se van a resumir a continuación.

1. El capítulo de Motivación y objetivos del trabajo (1) se encarga de explicar los objetivos del trabajo, y la motivación detrás del mismo.

2. El capítulo de Introducción a las redes neuronales (2) explica de forma resumida en qué consisten las redes neuronales junto a los conceptos más importantes de las mismas necesarios para implementar una red cualquiera.

3. El capítulo de Arquitecturas de red más comunes en modelos de lenguaje (3) se encarga de explicar los tipos de redes más comunes empleadas dentro del ámbito del Procesado del Lenguaje, más concretamente la arquitectura Transformer que es el pilar de este trabajo.

4. El capítulo de Museformer (4) explica la arquitectura y el funcionamiento del modelo Museformer, y sus ventajas e inconvenientes frente a otros tipos de modelos basados en Transformers.

5. El capítulo de Implementación, entrenamiento e inferencia del Museformer (5) habla del proceso de la implementación en sí de la red dentro de una máquina concreta, la preparación de los datos, el entrenamiento con sus parámetros en sí junto a los resultados y la generación de las piezas con sus medidas y valoraciones correspondientes.

[3] `https://github.com`

6. El capítulo de Condicionamiento (6) habla de cómo se podría implementar modificaciones al modelo Museformer para que a la hora de la generación se puedan introducir datos por texto en vez de generar las piezas de forma aleatoria.

7. Por ultimo, el capítulo Conclusiones (7) presenta una conclusión objetiva sobre el cumplimiento de los objetivos propuesto en el trabajo junto a la valoración del Museformer como herramienta de trabajo, y una valoración subjetiva sobre el trabajo en sí.

Capítulo 2

Introducción a las redes neuronales

2.1. Red Neuronal

Los Capítulos 2 y 3 de esta memoria proporcionan una revisión de conceptos que se han obtenido del curso "Deep Learning Course"de la plataforma `https://atcold.github.io/NYU-DLSP21/` de Yann LeCun y Alfredo Canziani [1].

Una red neuronal es una función matemática compleja[1] que para cada valor de entrada genera su respectivo valor de salida. Dentro de esta función tenemos una serie de capas, que van desde la capa de entrada, capas intermedias, también llamadas capas ocultas, y una capa de salida. Dentro de estas capas tenemos una serie de nodos, a los cuales llamaremos neuronas y constan de dos partes:

1. Una combinación lineal de neuronas anteriores.

2. Una función no lineal evaluada elemento a elemento.

Entre las distintas capas existen una serie de conexiones entre neuronas a las que llamaremos *pesos*. Dada una capa $H^{(i)}$, estas conexiones determinan el grado de aportación que tiene cada neurona de la capa $H^{(i-1)}$ con respecto a una neurona de la capa $H^{(i)}$. Estos pesos son uno de los parámetros que se entrenan dentro de la red. El otro parámetro es el *bias* (o sesgo) que se puede añadir a la combinación lineal de las neuronas capa $H^{(i-1)}$ con los respectivos pesos asociados a cada neurona de la capa $H^{(i)}$. En resumen, si llamamos $a_k^{(i)}$ a la k-ésima neurona de la capa $H^{(i)}$ con $k \in 1, ..., n$ y $a_j^{(i-1)}$ la j-ésima neurona de la capa la $H^{(i-1)}$ con $j \in 1, ..., l$ obtenemos la siguiente expresión para los *pesos* y *bias* para una neurona arbitraria de la capa $H^{(i)}$.

$$a_k^{(i)} = \sum_{j=1}^{l} w_{k,j} \cdot a_j^{(i-1)} + b_k \tag{2.1}$$

[1]En el sentido de tener muchos parámetros dentro de la misma, ya que en realidad es una función que toma valores reales.

En esta expresión $w_{k,j}$ es el peso que relaciona la neurona $a_k^{(i)}$ con $a_j^{(i-1)}$ y b_k es el sesgo asociado a la neurona $a_k^{(i)}$.

Si definimos $\mathbf{a^{(i)}}$ como el vector de neuronas de la capa $H^{(i)}$ de tamaño $[n, 1]$ y $\mathbf{a^{(i-1)}}$ como el vector de neuronas de la capa $H^{(i-1)}$ de tamaño $[l, 1]$, entonces la expresión para $\mathbf{a^{(i)}}$ usando (2.1) queda de la siguiente forma:

$$\mathbf{a^{(i)}} = \mathbf{W a^{(i-1)}} + \mathbf{b} \tag{2.2}$$

expresión en la cual $\mathbf{W}$ es la matriz de pesos de tamaño $[n, l]$ y $\mathbf{b}$ es el vector *bias* de tamaño $[n, 1]$.

Con (2.2) obtenemos lo que se denomina *FC-Layer* (*Full-Connected Layer* o capa completamente conectada), es decir, todas las neuronas de una capa están conectadas con todas las neuronas de la siguiente capa. Este modelo es el más elemental dentro de las arquitecturas que existen en Deep-Learning, pero a la vez es el más costoso en términos de operaciones. Si todas las capas de una red son del tipo *FC-Layer* entonces llamaremos *FC-Network* a la red neuronal en cuestión.

Hasta ahora hemos tratado la parte de la combinación lineal de las neuronas, por lo que hablaremos brevemente de la parte no lineal de la neurona. Esta parte consta de una función **no lineal** evaluada elemento a elemento dentro de cada capa de la red, es decir, si llamamos $\boldsymbol{f}$ a la función no lineal en cuestión la expresión es la siguiente:

$$z_k^{(i)} = f(a_k^{(i)}) \qquad k \in 1, ..., n \tag{2.3}$$

donde $z_k^{(i)}$ es el resultado de aplicarle la función $\boldsymbol{f}$ a cada $a_k^{(i)}$. Como se puede ver el vector resultante $\mathbf{z^{(i)}}$ es del mismo tamaño que $\mathbf{a^{(i)}}$ ya que lo hemos definido de esta manera. Notar que de esta manera la combinación lineal de las ecuaciones (2.1) y (2.2) toman $\mathbf{z^{(i-1)}}$ como vector de entrada, ya que la función $\boldsymbol{f}$ produce la salida de la neurona.

Para finalizar esta sección mencionaremos que el hecho de usar funciones no lineales se debe a dos razones, siendo la primera de ellas la posibilidad de que la relación entre datos de un problema dado que queremos solucionar aplicando redes neuronales no tengan relación lineal entre ellos, por ejemplo si queremos clasificar imágenes según algún tipo de etiquetas como fotos de animales por el tipo de animal en cuestión, si un usuario hace click en un anuncio de una página, etc. La otra razón es que si tenemos un conjunto de capas lineales conectadas se comporta como una única capa ya que la combinación lineal de capas es lineal y se perdería el propósito del Deep-Learning, el cual se basa en añadir capas intermedias que se encargan de extraer distintas características en cada una de ellas.

2.2. Entrenamiento de una red neuronal

2.2.1. Función de coste y de pérdidas

Hasta ahora hemos visto una estructura básica de una red neuronal, por lo que el siguiente paso es ver cómo se puede entrenar para que nos produzca resultados correctos. Para ello definimos el algoritmo del Descenso por Gradiente (*Gradient Descent Algorithm*) y la forma de actualizar los *pesos* y *sesgos* de la red mediante *Backpropagation*.

Para hablar del algoritmo de Descenso por Gradiente, al cual llamaremos *Gradient Descent*, tenemos que definir una arquitectura que nos sirva para entrenar la red en cuestión. En la siguiente figura se puede ver una posible representación de la misma:

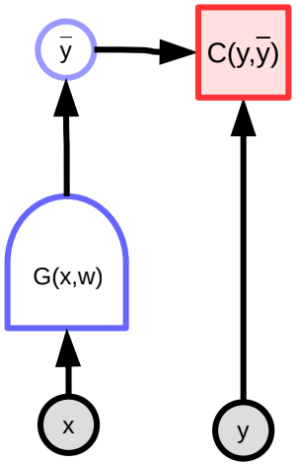


Figura 2.1: Arquitectura básica de red neuronal para entender el entrenamiento. Tomada de [1]

El elemento novedoso es la función de coste $C(y, \bar{y})$, la cual nos mide el error que se produce entre la predicción $\bar{y}$ de una red neuronal $G(x, w)$ dada una entrada arbitraria x, y el dato que queremos predecir y al cual llamaremos *label* (o etiqueta) y se usará exclusivamente durante la fase de entrenamiento.

La función de coste puede definirse de muchas formas, por ejemplo, la *norma Euclídea* o *L2*, o la *norma Manhattan* o *L1*. A partir de la función de coste podemos definir una función de pérdidas que será el objetivo a minimizar durante el entrenamiento para reducir el error entre la predicción de la red neuronal y la muestra real.

$$L(x, y, w) = C(y, G(x, w)) \tag{2.4}$$

Donde L es la función de pérdidas por muestra. Notar que tanto la función de coste como de pérdidas son paramétricas, donde los parámetros son los *pesos* de la red w,

los cuales además son implícitos, por lo que se puede complicar mucho el cálculo de los algoritmos propuestos.

Una vez definida la función de pérdidas por muestra definimos la función de pérdidas promedio como:

$$S = \{(x[p], y[p]) \mid p \in \{0, ..., P-1\}\}$$
$$L(S, w) = \frac{1}{P} \sum_{p=0}^{P-1} L(x[p], y[p], w) \tag{2.5}$$

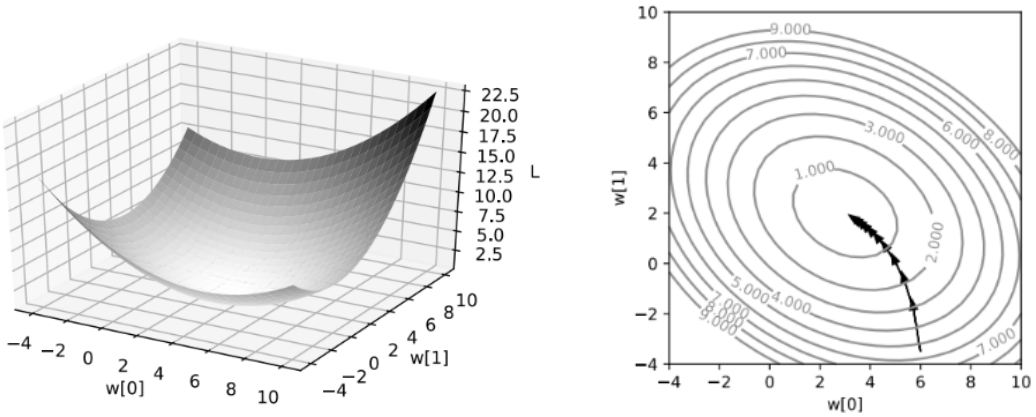
donde hemos definido un conjunto S de P pares de muestras (x, y).

2.2.2. Gradient Descent

Una vez definida la función de pérdidas ya podemos definir el algoritmo de *Gradient Descent* con la siguiente expresión, la cual se explicará debajo.

$$w \leftarrow w - \eta \frac{\partial L(S, w)}{\partial w} \tag{2.6}$$

Supongamos que la función de pérdidas L contiene dos pesos, w_0 y w_1, por lo que podemos representar esta función como una curva en 2D de la siguiente forma:



(a) L en función de w_0, w_1 (b) Algoritmo *Gradient Descent*

Figura 2.2: Visualización del algoritmo de Gradient Descent. Tomada de [1]

La figura 2.2b es la representación de los cortes de la función L que tienen el mismo valor numérico. Dado un valor arbitrario inicial a los pesos w_0 y w_1, el algoritmo consiste en encontrar la dirección espacial de máximo crecimiento con respecto al punto inicial e ir actualizando los pesos en la dirección contraria a este máximo crecimiento. Este proceso se repite iterativamente hasta llegar a un mínimo local[2] en la función de

[2]En este caso es absoluto, pero para funciones de millones de parámetros su forma es mucho más compleja.

pérdidas. Una analogía a este algoritmo sería estar perdido en una montaña y para llegar a un poblado hay que ir dando pasos en una dirección que nos guíe hacia abajo. En la expresión (2.6) la dirección de máximo con respecto a los pesos viene dada por la derivada parcial de la función de pérdidas con respecto a los pesos, y al añadirle el signo menos "descendemos en la función de coste". Para finalizar la expresión, el hiper-parámetro[3] η se llama *Learning Rate* y nos controla la velocidad de convergencia del algoritmo al mínimo de la función. Está definido en el intervalo $[0, 1]$ y cuanto más grande sea, más rápido es el algoritmo, pero corremos el riesgo de que acabe divergiendo debido a que los pasos sean tan grandes que se salten el mínimo y en vez "descender en la función" acabe creciendo desmesuradamente.

Por último cabe mencionar que la expresión (2.6) se denomina *Full-Batch Gradient* porque hace la media de la función de pérdidas para todos los datos de entrenamiento en cada paso. Esto no es eficiente de forma computacional, por lo que se recurre a una técnica llamada *SGD* (Stochastic Gradient Descent) que en vez de usar todos los datos usa un subconjunto de los mismos llamado *mini-batch*. Esto es más rápido computacionalmente, pero nos genera un camino al mínimo ruidoso. Existen modificaciones del algoritmo para ayudar a la convergencia rápida y eliminación de ruido, pero no se van a tratar en este trabajo ya que su objetivo no es el estudio de la convergencia. De todas estas técnicas, el Museformer usa el algoritmo de Adam [2] que es el más popular y eficiente de todos los que existen a día de hoy.

2.2.3. Backpropagation

Hasta ahora hemos definido nuestra red neuronal y visto un algoritmo para poder calcular los *pesos* y *sesgos*[4] de la misma, pero aun nos falta un ultimo paso que es actualizar los parámetros en sí de todas las capas. Esto se denomina *Backpropagation* y consiste en aplicar la regla de la cadena a la función de coste con respecto a todas las capas de la red neuronal. En concreto para la capa *i-ésima* queda la expresión de la siguiente manera:

$$\frac{\partial L(S, w, b)}{\partial w_i} = \frac{\partial L(S, w, b)}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial w_i} \tag{2.7}$$

$$\frac{\partial L(S, w, b)}{\partial b_i} = \frac{\partial L(S, w, b)}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial b_i} \tag{2.8}$$

donde z_{i+1} viene de la expresión (2.3) y hace referencia a la salida de la capa $H^{(i+1)}$ de la red. Si z_{i+1} tiene dimensión $[M, 1]$ y w_i tiene dimensión $[N, 1]$, entonces la relación

[3]Es un parámetro que controla otros parámetros dentro de la fase de entrenamiento, por eso lo denominamos así.

[4]No se ha mencionado de forma explícita, pero se hace de forma análoga a los pesos.

entre dimensiones cumple en la expresión (2.7) que $[1, N] = [1, M] \times [M, N]$. Ocurre algo de manera similar con (2.8) ya que el sesgo tiene la misma dimensión que los pesos de la capa *i-ésima*.[5] La expresión $\frac{\partial z_{i+1}}{\partial w_i}$ es una matriz jacobiana que cumple la siguiente relación para cada componente:

$$\left(\frac{\partial z_{i+1}}{\partial w_i}\right)_{k\,l} = \frac{(\partial z_{i+1})_k}{(\partial w_i)_l} \tag{2.9}$$

Es decir, la entrada (k, l) en la matriz es la componente *k-ésima* de la derivada parcial del vector z_{i+1} con respecto a la componente *l-ésima* de la derivada parcial del vector w_i. Para el caso de los sesgos solo cambia la notación por lo que quedaría de forma similar la expresión, solo que esta vez se tienen en cuenta los sesgos en vez de los pesos.

Aplicando de forma recursiva las expresiones (2.7) y (2.8), es decir, comenzando por la capa de salida de la red y retrocediendo capa por capa hasta llegar a la capa de entrada (sin incluir) acabamos actualizando todos los pesos y sesgos de la red para un paso de iteración del algoritmo *Gradient-Descent* empleado. El nombre de *Backpropagation* viene precisamente de ir hacia atrás propagando los pesos y sesgos actualizados.

2.3. Inferencia de una red neuronal

Una vez hecha la fase de entrenamiento ya tenemos la red ajustada para obtener los resultados correctos en función de unos datos de entrada "adecuados". Por ejemplo si estamos clasificando imágenes de coches la red sabrá discernir entre modelos de coches si se ha entrenado con una cantidad suficiente de ellos. Pero si nos salimos del ámbito de la red, es decir, le pedimos que nos genere un coche nuevo, dependiendo de como se haya entrenado la red obtendremos un resultado u otro. Por ejemplo si la función de coste es el error cuadrático medio el resultado será una mezcla de todos los modelos de coches que conozca la red y saldrá una imagen probablemente borrosa o sin sentido. Para solucionar este problema se recurre a unos modelos basados en energía [1] que se encargan de modelar una función de energía para unos conjunto de datos de entrada de tal manera que los valores de la función sean nulos en un entorno cercano a estos mismos (sustituyendo a la función de coste).

La función de coste $F(x, y)$ se define como una función escalar, es decir, para cada par de puntos (x, y) tenemos un único valor numérico real. Las redes que contienen este tipo de modelos de energía se entrenan de dos maneras:

[5] Por definición de gradiente su representación en forma vectorial es un vector fila.

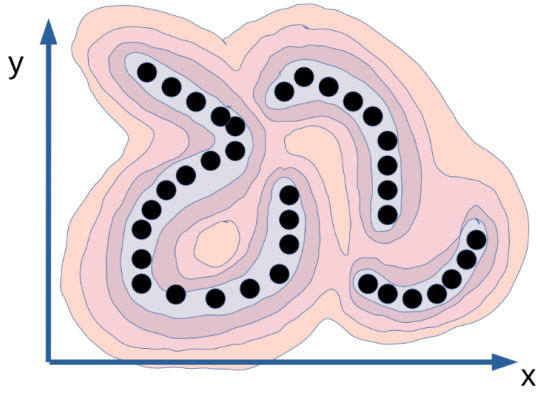


Figura 2.3: Representación de $F(x, y)$. Tomada de [1]

1. Métodos de contraste.

2. Métodos de regularización.

Los métodos de contraste se basan en comparar dos puntos dentro de la función, uno de ellos perteneciente a los datos de entrada, y el otro puede ser aleatorio o una perturbación del dato de entrada para posicionarse en un espacio cercano, de tal manera que vamos aumentando la energía en aquellos puntos que se van alejando de los datos, y vamos disminuyendo la energía en los puntos que están muy cerca de los datos de entrada de tal forma que acabemos con una función como en 2.3, en la cual el color gris representa valores de $F(x, y)$ pequeños y conforme nos vamos alejando obtenemos valores más grandes (naranja).

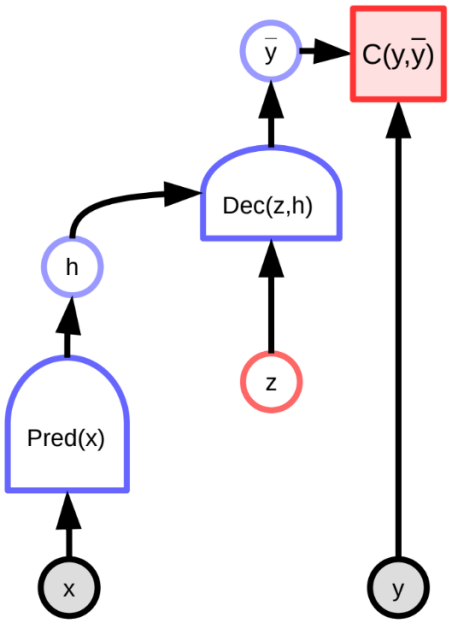


Figura 2.4: Arquitectura de red con variable latente. Tomada de [1]

Para los métodos de regularización se emplea una variable extra z representada en la figura 2.4 la cual se llama variable latente[6] (*Latent-Variable*) y su función es modelar características importantes de los datos de entrada que le vamos proporcionando durante la fase de entrenamiento. Por ejemplo si estamos analizando imágenes de coches esta variable aprende características como tipos de rueda, carrocería, maletero, puertas, etc.

Los métodos de regularización se encargan de fijar un "volúmen" alrededor del espacio de los datos de entrada y el resto del espacio tiene energía alta. Esta regularización aparece para restringir la capacidad de la variable latente z, ya que si no tiene restricciones z se adaptará a los datos de entrada y asignará energía cero a todo el dominio de la función $F(x, y)$, caso desfavorable.

Existen muchos modelos tanto de contraste como regularizados, e incluso de contraste con variables latentes, en el caso de este trabajo se utilizará el método de regularización L2, que consiste en añadirle la norma euclídea o L2 de los pesos a la función de pérdidas. En caso de querer profundizar más sobre los distintos tipos modelos se pueden ver en [1].

Una vez ajustada la función de coste a los datos de entrada, el siguiente paso es la inferencia y no es más el proceso de encontrar valores y tales que minimicen el valor de $F(x, y)$ dado por la siguiente ecuación:

$$\hat{y} \; = \; argmin_y \, F(x, y) \tag{2.10}$$

La ecuación (2.10) ya incluye el caso de variable latente de forma implícita ya que en [1] se explica como se puede redefinir cierta función de energía $E(x, y, z)$ para llegar a (2.10).

Para finalizar, hay que remarcar el hecho de que el tema de modelos basados en energía es amplio y en este apartado solamente se quería introducir de forma muy superficial la fase de inferencia para entender de forma general como funciona el Museformer tratado unas secciones más adelante.

2.4. Funciones no lineales más empleadas

Uno de los aspectos a tratar es que tipo de funciones no lineales podemos emplear para construir la red neuronal que resuelva el problema planteado. Hay muchas funciones que cumplen este criterio, pero aquí vamos a tener en cuenta las más utilizadas.

[6]El valor de esta variable es implícito.

La primera función no lineal, y a su vez la más empleada es la función *ReLU* y viene dada por la siguiente expresión:

$$ReLU(x) = \begin{cases} \text{x}, & \text{si x} > 0 \\ 0, & \text{si x} \leq 0 \end{cases} \tag{2.11}$$

Como podemos observar la función solamente toma valores distintos de cero si la entrada es positiva. En este caso la no linealidad viene dada por el punto en el origen donde la función cambia de forma abrupta.

Representado la función obtenemos la siguiente figura 2.5:

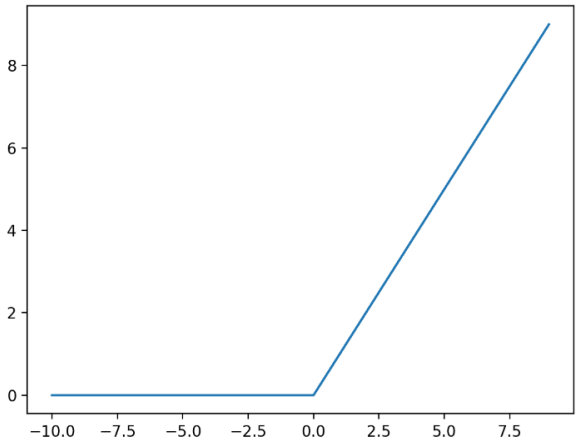


Figura 2.5: Representación de la función ReLU.

La siguiente función es la *tangente hiperbólica* y viene dada por la siguiente expresión:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.12}$$

Para ver las no linealidades de esta función la representamos primero para verlas de forma más sencilla en la figura 2.6.

En este caso vemos que para valores muy grandes tanto negativos como positivos la función converge a -1 y 1 respectivamente.

Para finalizar nos queda la función *sigmoide* que viene dada por la siguiente expresión:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.13}$$

La representación queda de la podemos ver en la figura 2.7:

Notar que esta función es similar a la tangente hiperbólica, solamente que en este caso esta desplazada en el eje positivo de tal manera que la convergencia de valores grandes tanto negativos como positivos ocurren en el intervalo $[0, 1]$.

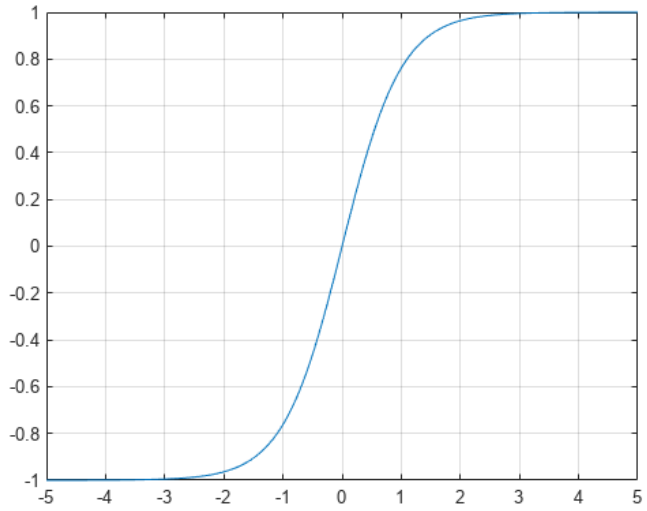


Figura 2.6: Representación de la función $tanh(x)$.

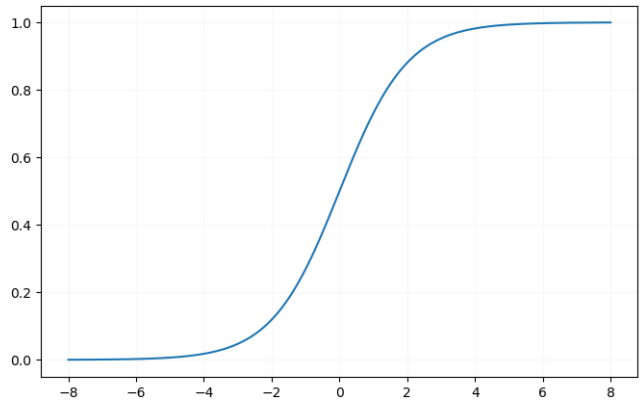


Figura 2.7: Representación de la función $\sigma(x)$.

Existen más funciones no lineales, pero estas tres mencionadas son las más utilizadas en redes neuronales debido a que debido a sus expresiones son fácilmente derivables[7], por lo que el entrenamiento es más rápido que utilizando otras funciones más complejas. También permiten clasificar de forma sencilla los datos de un problema bajo distintas etiquetas, por lo que tras varias modificaciones de dichos datos en sucesivas capas de red obtenemos las respectivas agrupaciones entre datos bajo una misma etiqueta[8].

Para finalizar, mencionar que de estas tres funciones la más utilizada es la función *ReLU* debido a que a el gradiente no satura durante la fase de entrenamiento porque, o es nulo para datos negativos y no se propaga, o es uno para datos positivos y se propaga directamente actualizando los respectivos pesos y sesgos. En los otros dos casos $tanh(x)$ y $\sigma(x)$ para datos con valores muy positivos o negativos el gradiente no es nulo, pero es muy pequeño, y a la hora de propagar el gradiente implica que los pesos y sesgos se

[7]En el caso de *ReLU* se modifica el origen para poder ser derivable, ya que en principio no lo es.
[8]Para más información véase el algoritmo *K-means* en [3]

actualizan de forma muy lenta empeorando el tiempo de entrenamiento.

2.5. Bloques básicos de redes neuronales y sus gradientes

Para finalizar esta sección de introducción a las redes neuronales vamos a mencionar brevemente los bloques que construyen los distintos tipos de arquitecturas de redes neuronales y sus respectivos gradientes para poder entrenarlas.

Comenzamos describiendo todos los bloques:

Bloque Lineal:

$$Y = W \cdot X$$
$$\frac{\partial C}{\partial X} = W^T \cdot \frac{\partial C}{\partial Y}$$

Este bloque hace referencia a las conexiones neuronales dentro de la red. El gradiente se obtiene transponiendo la matriz de pesos.

Bloque no lineal (ReLU):

$$Y = ReLU(X)$$
$$\begin{cases} \text{Si x} > 0 & \frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y} \\ \text{Si x} < 0 & \frac{\partial C}{\partial X} = 0 \end{cases}$$

En este caso el gradiente solamente se propaga si los valores de entrada son positivos.

Bloque de duplicación:

$$Y1 = X, \quad Y2 = X$$
$$\frac{\partial C}{\partial X} = \frac{\partial C}{\partial Y1} + \frac{\partial C}{\partial Y2}$$

Este bloque se usa en el caso de parámetros compartidos.[9]

Bloque suma

$$Y1 = X1 + X2$$
$$\frac{\partial C}{\partial X1} = \frac{\partial C}{\partial Y}; \quad \frac{\partial C}{\partial X2} = \frac{\partial C}{\partial Y}$$

Este bloque aparece junto a las capas de normalización de la red y ayuda a propagar el gradiente en casos en los que se pueda desvanecer por el camino mediante unas conexiones residuales[10]. Las capas de normalización no tienen una finalidad clara, pero ayudan a la convergencia del aprendizaje de la red neuronal.

Bloque máximo:

$$Y = max(X1, X2)$$

[9]Los parámetros compartidos se usan el redes convolucionales donde se puede explotar la redundancia de los datos. [1]

[10]En la sección 3.4 se verá en detalle.

$$\begin{cases} \text{Si X1} > \text{X2} & \frac{\partial C}{\partial X1} = \frac{\partial C}{\partial Y}; \frac{\partial C}{\partial X2} = 0 \\ \text{Si x1} < \text{X2} & \frac{\partial C}{\partial X2} = \frac{\partial C}{\partial Y}; \frac{\partial C}{\partial X1} = 0 \end{cases}$$

Este bloque como su nombre indica propaga el gradiente del valor máximo de dos entradas $X1$ y $X2$. Se puede usar dentro de los métodos de contraste explicados en la sección 2.3, o para clasificar datos en función del que mayor contribución tenga dentro de la red.

Bloque LogSoftMax:

$$Y_i = X_i - log[\sum_j e^{X_j}]$$

$$\frac{\partial C}{\partial X_i} = 1 - \frac{e^{X_i}}{\sum_j e^{X_j}}$$

Este bloque se usa principalmente en la capa de salida para asignar probabilidades a las neuronas de dichas capas, y así tener una medida sobre que datos tienen más impactos dado un problema arbitrario. La función viene de tomar el logaritmo a la función $SoftMax(X_i) = \frac{e^{X_i}}{\sum_j e^{X_j}}$ para obtener valores más normalizados y sin variaciones bruscas debido a la sensibilidad de la función $SoftMax$

Con esto concluimos esta sección de bloques básicos más usados dentro de la construcción de redes neuronales. Aunque no esté mencionado aquí también se usa tanto la función $tanh(x)$ como la $\sigma(x)$, pero no son tan comunes como la función $ReLU(x)$. El resto de bloques más avanzados se construyen a partir de estos mencionados.

Capítulo 3

Arquitecturas de red más comunes en modelos de lenguaje

3.1. Redes Neuronales Recurrentes y LSTM

Hasta ahora hemos visto el concepto de red neuronal, cómo se entrena a nivel básico y un poco más avanzado en modelos basados en energía, el proceso de inferencia para que la red sea capaz de predecir información nueva, algunas funciones no lineas utilizadas y los bloques básicos empleados para la construcción de redes neuronales.

Si bien es cierto que hay muchas arquitecturas distintas para diferentes ámbitos de trabajo e investigación como visión por ordenador, robótica, generación de fotogramas en videojuegos, generación de voz, etc., nosotros nos centraremos en aquellas que dan solución a problemas de lenguaje natural ya que Museformer, que es la arquitectura que vamos a emplear en este trabajo, viene dado por la arquitectura Transformer propuesta inicialmente en [4] para resolver el problema que tenían otras redes a la hora de traducir palabras de un idioma a otro.

El primer cambio que se puede hacer a una red neuronal básica en la que tenemos capas de entrada, ocultas y de salida es generar un bucle en algunas capas ocultas de la misma, de ello surge el concepto de *Red Neuronal Recurrente* (*RNN* en inglés) (ver figura 3.1). Esto se hace para dotar de cierta capacidad de memoria a la red ya que con la arquitectura básica *Feed-Forward* no tenemos esa capacidad.

El entrenamiento de una red de este estilo se basa en *"estirar"* en función del tiempo la red comenzando por la izquierda el instante temporal inicial y añadiendo a la derecha los sucesivos instantes temporales hasta llegar a un punto final, y aplicar el algoritmo de *Back-Propagation* a través del tiempo (*BPTT* en inglés) yendo hacia atrás desde la salida, pasando por los estados intermedios y llegando a la entrada finalmente. La figura 3.2 ilustra este proceso.

En la figura 3.2 se puede ver tres iteraciones de una red recurrente y el algoritmo de

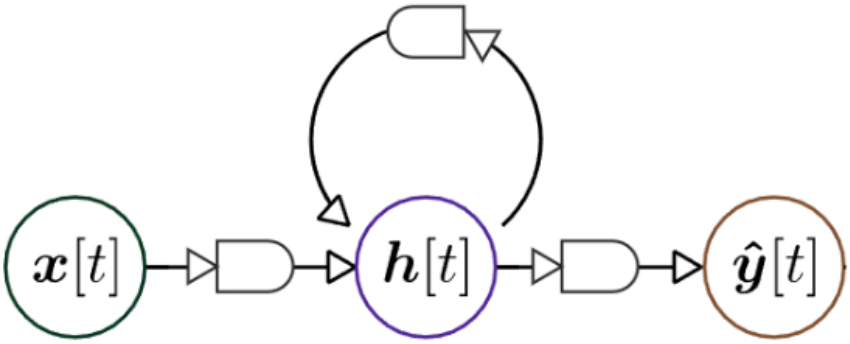


Figura 3.1: Ejemplo de Red Neuronal Recurrente. Tomada de [1]

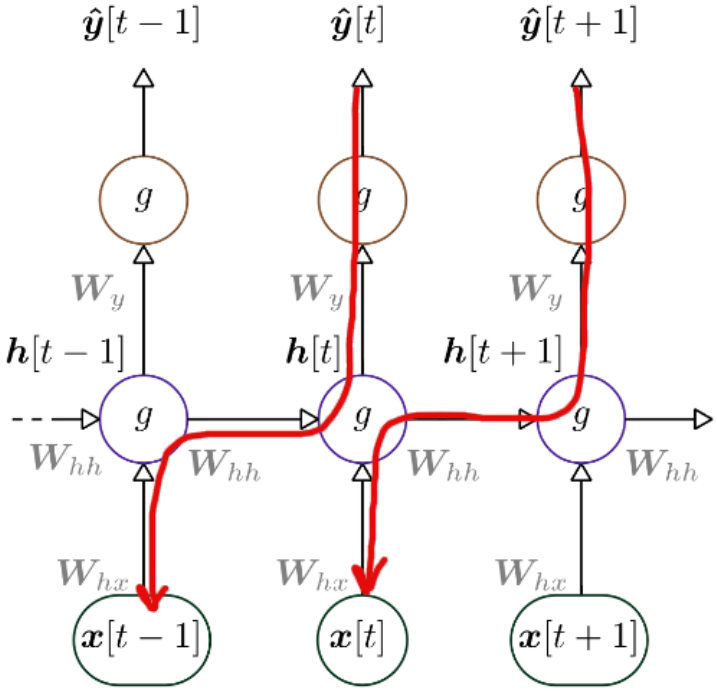


Figura 3.2: Back-Propagation Through Time. Tomada de [1]

BPTT esta indicado con las flechas rojas. Empezamos en la salida actual y propagamos el gradiente hacia el estado anterior y luego a la entrada. De la salida anterior hacemos lo mismo, pero con sus estados anteriores correspondientes. Esto se hace hasta llegar a la entrada en el tiempo inicial.

Aunque teóricamente este algoritmo es sencillo de implementar tiene dos grandes inconvenientes por los cuales no se usa. El primero de ellos es el desvanecimiento del gradiente (*Gradient Vanishing*) y ocurre cuando tenemos redes recurrentes de gran tamaño en el tiempo por lo que al utilizar el algoritmo *BPTT* dependiendo de los parámetros de la red el gradiente puede anularse o hacerse infinito antes de llegar al tiempo inicial con lo que nos limita el tamaño temporal que puede tener la red. Existen métodos para mitigar los efectos del desvanecimiento del Gradiente [1], pero

no se usan porque aun así la capacidad de memoria de la red se queda corta en la mayoría de los casos. El otro problema es la gran cantidad de información que hay que guardar en memoria para una red de tamaño considerable, ya que dependiendo del desarrollo temporal usado hay que guardar todos los valores de las capas intermedias y de salida en todos los instantes temporales.

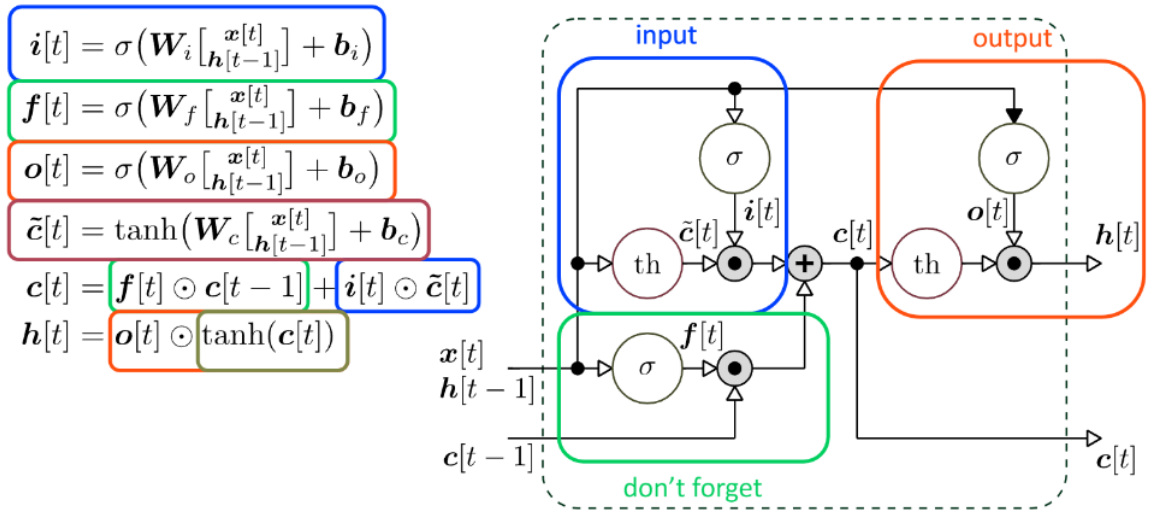


Figura 3.3: Red LSTM. Tomada de [1]

El otro tipo de red que viene a solucionar el problema principal de las *RNN* se denomina *LSTM* (*Long-Short Term Memory*) o Memoria de Corto-Largo plazo. Es más compleja la arquitectura que tiene (figura 3.3) y los detalles del funcionamiento se pueden leer en el documento de sus autores en [5], pero en esencia se encarga de abrir o cerrar caminos dentro de una *RNN* desarrollada a lo largo del tiempo como se puede ver en la figura 3.4.

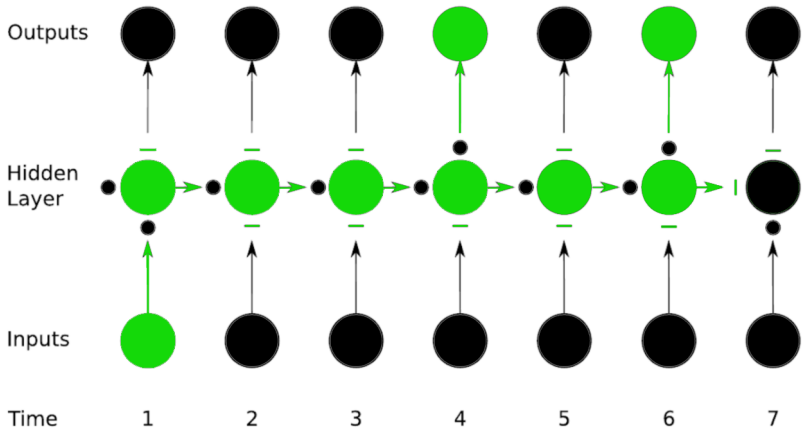


Figura 3.4: *BPPT* en LSTM. Tomada de [1]

En esta figura los círculos representan los caminos por donde se pueden propagar los gradientes y las barras por donde no lo pueden hacer. Esto permite tanto agrandar

la capacidad de memoria de la red como reducir el número de parámetros que hay en la red ya que aquellos caminos cerrados son valores nulos directamente.

3.2. Seq2Seq

Usando las redes de la sección previa se puede construir mediante la concatenación de varios *LSTM* lo que se conoce como una red *Seq2Seq* (Secuencia a Secuencia) creada por Sutskever en [6], y consiste introducir una secuencia de datos en la entrada, por ejemplo una oración en inglés obtener un vector de características intermedio y en la capa de salida hacer el proceso inverso para obtener otra secuencia distinta, por ejemplo la misma oración en alemán. La siguiente figura ilustra este párrafo.

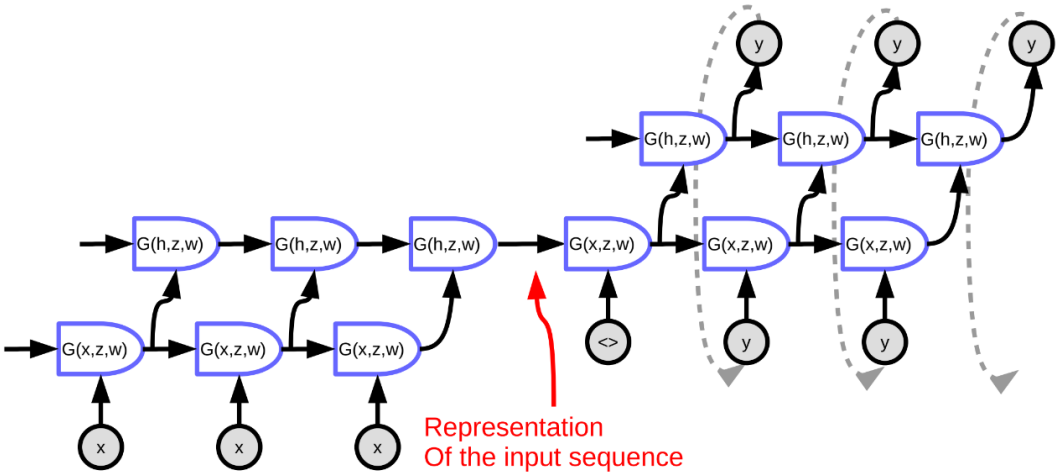


Figura 3.5: Representación de *Seq2Seq*. Tomada de [1]

Uno de los problemas que tiene esta propuesta es la cantidad de *LSTM* necesarios, ya que por cada entrada se necesitan dos de estos bloques, por lo que la escalabilidad es reducida para secuencias largas. También existe un límite de secuencia que si se sobrepasa la red deja de funcionar correctamente. Yann LeCun mencionó en [1] que era de 30 palabras mas o menos.

3.3. Seq2Seq con mecanismo de atención

Una modificación que se puede hacer a esta red es añadirle lo que se conoce como mecanismo de atención, el cual se encarga de darle importancia a unos datos de entrada frente a otros. Matemáticamente es un bloque que hace una combinación lineal ponderada de las entradas con pesos de mayor tamaño a aquellas entradas que

aportan más valor para un instante determinado. Este mecanismo se profundizará más en la sección 3.4 ya que es el pilar fundamental del mismo y su popularidad actual.

El resultado de añadirle el bloque de atención a esta red se ilustra en la figura 3.7, en la cual se puede apreciar que no solamente se consigue mejorar el resultado de la salida por usar las entradas adecuadas, sino que también reducimos el coste computacional para hacerlo ya que en cada instante de tiempo solo tenemos las entradas con sus respectiva codificación[1] $G(x, z, w)$, el bloque de atención y la correspondiente decodificación $G(h, z, w)$.

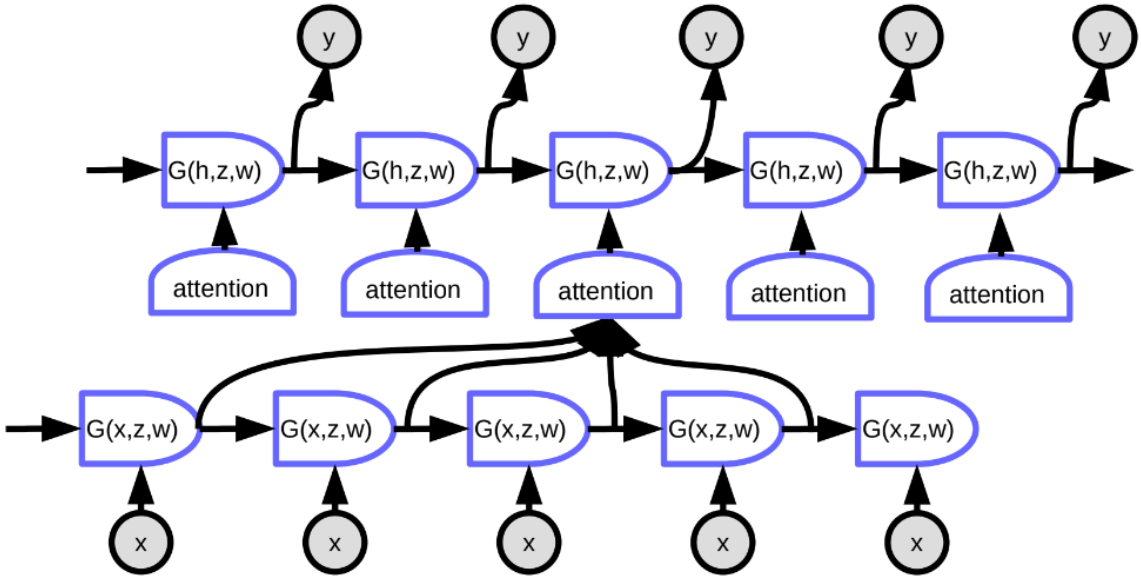


Figura 3.6: Representación de *Seq2Seq* con mecanismo de atención. Tomada de [1]

3.4. Transformers

Para finalizar este apartado vamos a describir un poco más en detalle el mecanismo más popular en el ámbito del lenguaje natural y también utilizado en el Museformer[2], que es el Transformer.

El Transformer es un tipo de arquitectura de red basada en un bloque llamado atención. Este bloque tiene como entrada un conjunto de valores arbitrario, y a partir de ellos mediante unas matrices de pesos calcula tres conjuntos de valores llamados *Query*, *Key* y *Value*. Los valores de *Query* son el equivalente a las preguntas que hace el módulo al conjunto de datos de entrada, por ejemplo si queremos comprar una guitarra eléctrica, la pregunta puede ser si en una tienda tienen guitarras de la marca *Fender*. Los valores de *Key* tienen el contenido "clave" para responder las preguntas,

[1] Transformación en tipos de datos útiles para la red.
[2] La parte -former viene precisamente de Transformer

normalmente una de las características de los datos de entrada. En el ejemplo anterior serían las propias marcas de guitarra que hay en la tienda en la que estamos. Por último, tenemos los valores *Value* que son todas las características que contienen los datos de entrada que no forman parte de la pregunta. En el ejemplo de las guitarras puede ser el tipo de cuerpo, cuerdas, madera de construcción, color, etc.

Para poder obtener la respuesta dentro del bloque se calcula lo que se llama como matriz de atención, la cual viene dada por la ecuación (3.1) y tiene los resultados de proyectar los valores de *Query* con respecto a todos los de *Key*, es decir, nos da una medida, que puede ser una puntuación o probabilidad, para saber qué valores de *Key* son los que más se acercan a la *Query*.

$$A = [soft]\,(arg)max_{\beta}(K^T Q) \tag{3.1}$$

En la expresión (3.1) si el conjunto de entrada $\mathbf{x}$ tiene dimensión $[n, 1]$, entonces $\mathbf{Q}$ es la matriz de *Query* de todos los datos de entrada, viene dada por $\mathbf{Q} = \mathbf{W_q}\,\mathbf{x}$ y tiene tamaño $[d, n]$ donde d es la dimensión de *embedding*[3]. $\mathbf{K}$ es la matriz de *Key*, viene dada por $\mathbf{K} = \mathbf{W_k}\,\mathbf{x}$ y tiene tamaño $[d, n]$, por lo que $\mathbf{A}$ tiene dimensión $[n, n]$. Una vez hecho el producto de las dos matrices anteriores tomamos el máximo que puede ser [soft] o [hard] (aunque en la expresión ponga [soft]). Esto nos da unos valores de medida de la respuesta que pueden ser una distribución continua [soft] o un único valor y el resto nulos [hard]. Para finalizar tenemos el parámetro β el cual se denomina temperatura y se encarga de controlar la uniformidad de los valores de la distribución de la función $argmax$. En el caso de los transformer se toma $\beta = \frac{1}{\sqrt{d}}$ para evitar que la dimensionalidad usada en el *embedding* afecte a los resultados.

Una vez obtenida la matriz $\mathbf{A}$ con la matriz $\mathbf{V}$ de *Value*, que viene de $\mathbf{V} = \mathbf{W_v}\,\mathbf{x}$ y tiene dimensión $[d', n]$[4], por lo que al final tenemos una salida que es una matriz $\mathbf{H} = \mathbf{V}\,\mathbf{A}$ de tamaño $[d', n]$ con los valores a la respuesta obtenida. En la figura 3.7(A) se puede ver el diagrama de bloques utilizado para implementar este párrafo y el anterior.

Hasta ahora hemos utilizado este bloque para calcular la respuesta a una pregunta dada sobre un conjunto de datos arbitrario, pero puede surgir la necesidad de hacer muchas más preguntas sobre un mismo conjunto de datos, por ejemplo queremos preguntar por más de una marca de guitarras ya que no solamente nos interesa Fender, por lo que se para solventar este pequeño inconveniente se utiliza lo que se conoce como *Multi-Head Attention* (figura 3.7(B))y no es más que la concatenación de varios módulos de atención simples para poder realizar varias preguntas al mismo tiempo.

[3]Es un valor utilizado dentro de las capas de red para representar información contextual.

[4]d' puede ser distinto a d ya que pueden haber más valores de *Value* que valores de *Key*.

La cantidad de preguntas que podemos realizar al mismo tiempo viene dado por los módulos encadenados y viene dado por el tamaño de las cabezas (*Heads*) que lo denotamos por H. Esta solución no solo permite hacer más preguntas, sino que paraleliza el proceso al hacerlo al mismo tiempo y nos ahorra tiempo de cómputo.

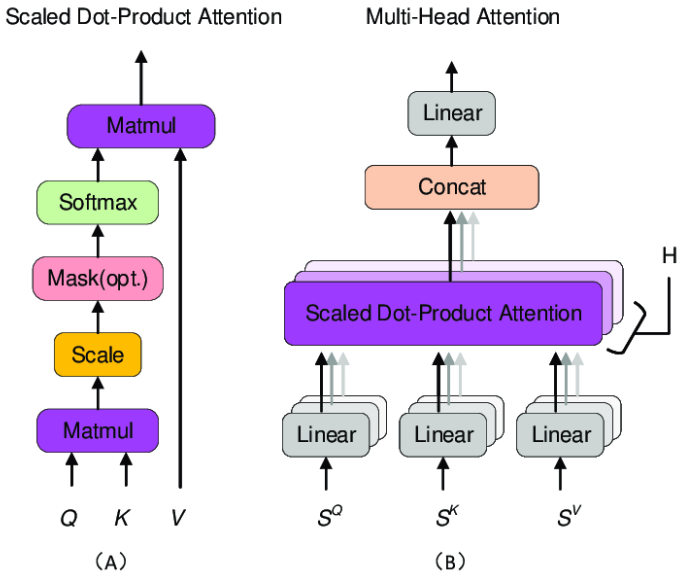


Figura 3.7: Mecanismo de atención. Tomado de [1]

Este proceso empleado utilizando el mismo conjunto de datos de entrada para todo se denomina *Self-Attention*. Si utilizamos otro conjunto de datos para responder a las preguntas que obtenemos de un conjunto inicial lo denominaremos *Cross-Attention* y el método es el mismo, solamente que si la matriz $\mathbf{K}'$ tiene muchos más valores $[\tau, d]$, expresión que viene de $\mathbf{K}' = \mathbf{W}'_{\mathbf{K}}\,\mathbf{x}'$ tal que $\{\mathbf{x}'_{\mathbf{i}}\}_{i=0}^{\tau}$, entonces la matriz $\mathbf{A}$ tiene tamaño $[\tau, n]$.

Con los bloques de *Self-Attention*, *Cross-Attention* y algunos más podemos construir la arquitectura completa. La figura 3.8 es el Transformer original del artículo *Attenion is all You Need* [4] y los bloques extra son una capa de suma y normalización, empleada para ayudar a la convergencia del entrenamiento de la red, ya que evita el *Gradient-Vanishing*, una capa de *Feed-Forward* que se usa para ajustar los valores de salida de una capa de atención previa para "tener una forma adecuada" en la capa posterior una capa llamada *Positional Encoding* que se encarga de asignar un orden a los datos de entrada ya que los módulos de atención no asignan posición a los datos de entrada.

La arquitectura final consiste en intentar predecir un valor deseado mediante una entrada $\mathbf{x}$ que nos proporciona información contextual mediante el mecanismo de *Self-Attention*, hacer un *Cross-Attention* con un valor de referencia $\mathbf{y}$ con un retardo de una unidad para poder predecir un resultado futuro y la salida de ese módulo evaluarla

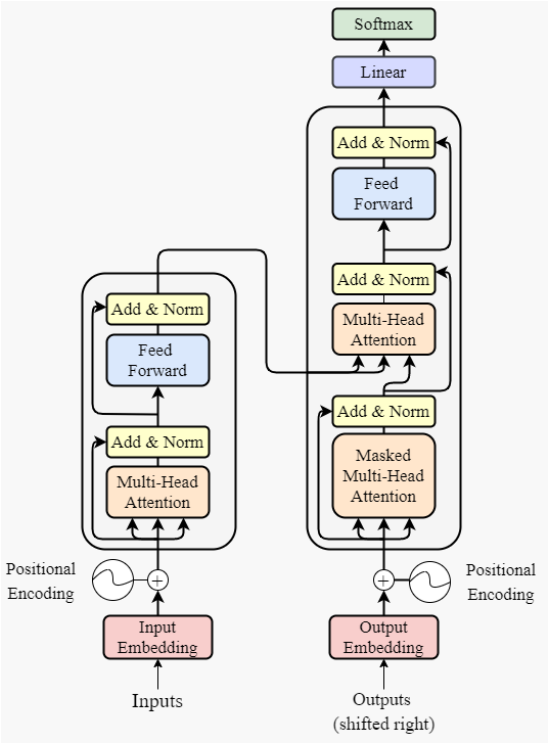


Figura 3.8: Arquitectura Transformer de referencia. Tomado de [4]

mediante una función de coste arbitraria. En el caso de utilizar modelos basado en energía se utiliza una función de energía y el entrenamiento se hace ya sea mediante métodos de contraste o de regularización. La inferencia se puede hacer a partir de datos propios o con valores arbitrarios. Una representación de la arquitectura viene dada por la siguiente figura:

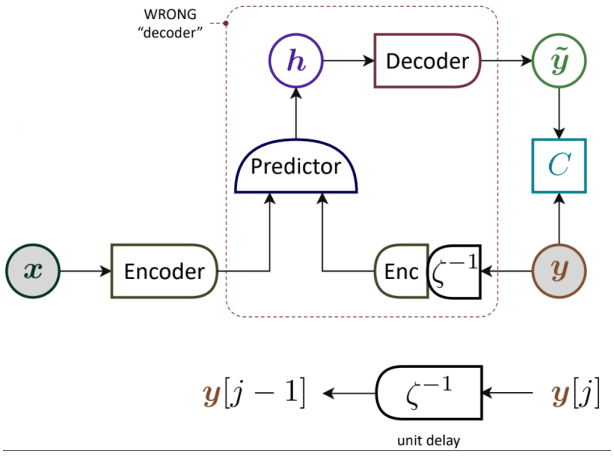


Figura 3.9: Arquitectura Transformer completa. Tomado de [1]

Cabe destacar que gracias al potencial que tiene la arquitectura Transformer tanto en sus módulos de atención como en su capa *Feed-Forward*, se puede escalar bien y el tiempo necesario tanto para entrenar como para inferir es drásticamente menor que

para una red recurrente arbitraria ya que los gradientes se actualizan al mismo tiempo no tener bucles. Por contra el módulo de atención tiene un orden de operaciones $\mathcal{O}(n^2)$ por lo que puede llegar a limitar el número de datos de entrada que puede aceptar.

Capítulo 4

Museformer

4.1. Introducción

La herramienta Museformer [7] se basa en una arquitectura Transformer ligeramente modificada en la capa de atención para intentar adaptarse de la mejor manera posible a la estructura de una obra musical. Esta modificación consiste en usar dos métodos de atención distintos, *Coarse-Grain Attention* y *Fine-Grain Attention* (ajuste de grano grueso y de grano fino).

Para entender los tipos de atención descritos en el párrafo anterior hay que recordar que la música se puede describir de forma simbólica mediante partituras, las cuales tienen sus elementos básicos de representación como son los compases, las notas, silencios, BPM[1], etc. La figura 4.1 muestra un ejemplo de representación mediante partitura. Para poder emplear esta información es necesaria traducirla a un lenguaje que entienda la máquina como puede ser MIDI, donde toda la información de la partitura se representa mediante distintos códigos numéricos, y de esta manera permite la comunicación entre instrumentos electrónicos y máquinas como ordenadores. Aunque esta solución permite que la máquina sea capaz de interpretar la información, no es la forma más adecuada de representación de cara a la red neuronal, por lo que se recurre a otro tipo de representación que son los *tokens*, y consiste en traducir la información MIDI a un lenguaje de más alto nivel en el que se especifica elementos más relacionados con la música como puede ser la frecuencia de las notas, su duración, el BPM, la duración del compás, intensidad de la nota, etc. Esto se puede ver en la figura 4.1.

El conjunto de tokens descritos en el párrafo anterior son la secuencia de entrada de la capa de atención modificada del Museformer, se describen de la siguiente manera $X = X_1, X_2, ..., X_b$ donde X hace referencia a la secuencia de tokens completa y b al número de compases de la obra. Para el i-ésimo compás tenemos

[1]Beats o pulsaciones por minuto

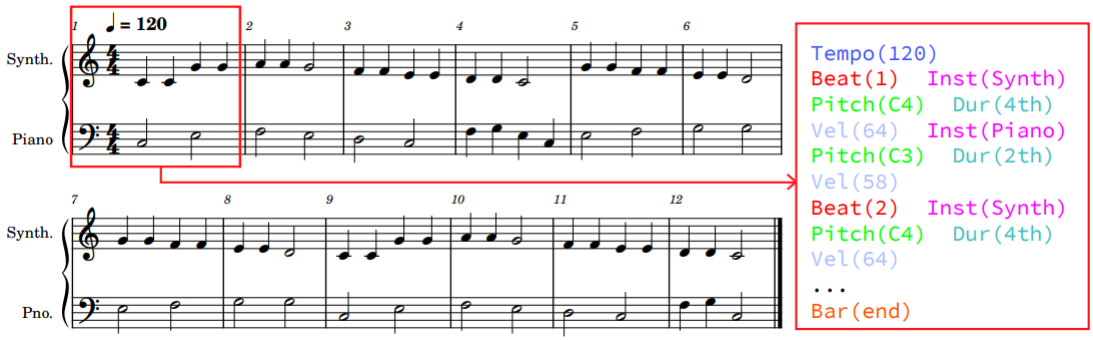


Figura 4.1: Partitura y su respectiva representación en tokens. Tomado de [7]

$X_i = x_{i,1}, ..., x_{i,|x_i|}$ tokens correspondientes. Tras cada compás se agrega un token de resumen s para el mecanismo de ajuste grueso, por lo que la representación final queda $X = X_1 s_1, ..., X_b s_b$. Esta representación se pasa por una capa de incrustación (embedding) para convertir la información en vectores dentro de un espacio vectorial y se concatenan los compases incrustados con los *beats* incrustados como información posicional[2] seguidos de una proyección lineal [7]. Las capas del Museformer se encargan de modelar la representación contextual, y la salida de la última capa oculta se pasa por un clasificador *SoftMax* para predecir el siguiente token. Esto se puede representar mediante un diagrama de bloques para un *head* descrito en la figura 4.2.

El mecanismo de atención se puede describir mediante la siguiente expresión[3]:

$$Attn(x_i', \mathbf{X}) = softmax\left(\frac{x_i' \mathbf{W_Q} (\mathbf{X W_K})^{\mathbf{T}}}{\sqrt{d}}\right) \mathbf{X W_V} \tag{4.1}$$

Donde $\mathbf{X} \in \mathbb{R}^{n_e \times d}$ es la secuencia de entrada con n_e el tamaño de la secuencia de entrada y d la dimensión de *embedding*, $\mathbf{X'} \in \mathbb{R}^{n_s \times d}$ es la secuencia objetivo con n_e el tamaño de la secuencia objetivo, $\mathbf{W_Q}, \mathbf{W_K}, \mathbf{W_V} \in \mathbb{R}^{d \times d}$ son las matrices de pesos correspondientes a la filosofía *Query, Key, Value* mencionada en la sección Transformer. Este mecanismo obtiene la representación contextual para cada $x_i' \in \mathbb{R}^{1 \times d}$ de la secuencia objetivo X'.

4.2. Atención de ajuste grueso y fino

Hasta ahora hemos descrito el proceso general del mecanismo de atención y la forma de organizar tokens de entrada de la manera adecuada para poder entrenar la red. A continuación vamos a describir los mecanismos que utiliza Museformer en sí para obtener los dos tipos de atención mencionados previamente.

[2]Recordemos que el Transformer no incluye un mecanismo de ordenación por defecto.
[3]Se ha descrito para una sola cabeza para simplificar la expresión.

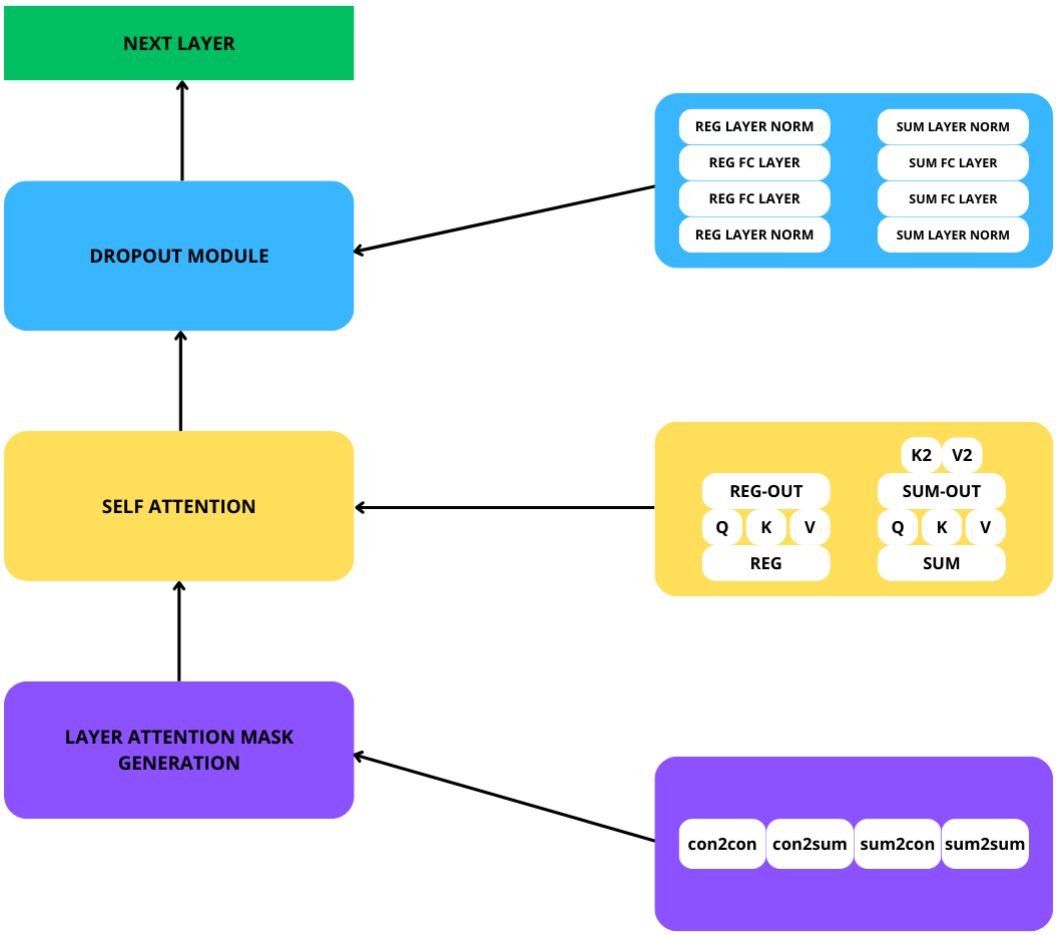


Figura 4.2: Diagrama de bloques de la arquitectura Museformer (un head). Imagen propia.

4.2.1. Resumen (*Summarization*)

Este es un proceso en el cual obtenemos los tokens de resumen s_i que nos darán la información para el ajuste grueso. La forma de obtener estos tokens vienen dados por la siguiente expresión.

$$\tilde{s_i} = Attn(s_i, [X_i, s_i]) \tag{4.2}$$

En esta expresión obtenemos el token de resumen para el compás i-ésimo mediante el mecanismo de atención sobre los propios tokens de dicho compás dado por la expresión $X_i = \{x_{i,1}, ..., x_{i,|X_i|}\} \in \mathbb{R}^{|X_i| \times d}$ y el propio token de resumen asignado a dicho compás s_i. La operación $[\cdot]$ es la concatenación.

4.2.2. Agregación (*Aggregation*)

En este proceso actualizamos cada token mediante el mecanismo de atención teniendo en cuenta qué compases tienen relación directa con el token que se va a

calcular, y cuáles no son tan importantes de manera que se utiliza el token de resumen de dichos compases. La expresión para un token en concreto es la siguiente:

$$\tilde{x}_{i,j} = Attn(x_{i,j}, [X_{R(i)}, X_{i,k\leq j}, \tilde{S}_{\tilde{R}(i)}]) \tag{4.3}$$

$x_{i,j}$ es el j-ésimo token del i-ésimo compás. $X_{R(i)}$ es la matriz de tokens del conjunto de compáses relacionados $R(i)$ con el compás i-éismo. $X_{i,k\leq j}$ es el conjunto de tokens del compás i-ésimo definido por $X_{i,k\leq j} = \{x_{i,k} \,|\, x_{i,k} \in X_i,\, k \leq j\}$, es decir, los tokens previos al token $x_{i,j}$ dentro del compás i-ésimo, y $\tilde{S}_{\tilde{R}(i)}$ es el conjunto de tokens de resumen de los compases que no están relacionados con el i-ésimo, $\tilde{R}(i)$.

De esta forma obtenemos los dos mecanismos de atención, ya que los compases que tienen relación entre ellos se calculan a nivel de token (ajuste fino) y los que no utilizan los tokens de resumen (ajuste grueso).

4.3. Compases relacionados

En la sección previa hemos definido la forma de obtener los mecanismos de atención del Museformer, pero no hemos definido los compases relacionados. Los compases relacionados son aquellos compases que tienden a repetir la estructura musical, ya sea mediante la estructura melódica, rítmica o ambas. Para ello los autores de Museformer definen el conjunto de notas del compás i-ésimo como $N(i)$, y la medida utilizada es una medida de similaridad entre pares de conjuntos de notas dada por la expresión:

$$l_{i,j} = \frac{|N(i) \cap N(j)|}{|N(i) \cup N(j)|} \tag{4.4}$$

Esta expresión nos da un número entre 0 y 1, dependiendo de cuántas notas comunes haya entre dos compases arbitrarios. Dos notas son iguales si tienen la misma frecuencia, duración y posición dentro del compás [7]. Finalmente se toma la media de similaridad entre pares de compases que se encuentran t intervalos dentro de un conjunto de entrenamiento dado D. Esto es:

$$L_t = media\left(\sum_{D}\sum_{j=i+t} l_{i,j}\right) \tag{4.5}$$

Aplicando esta medida al conjunto de datos de entrenamiento del Museformer que es la base de datos MIDI Lack MIDI Dataset, LMD[4] con más de 100.000 archivos MIDI, y luego a otra base de datos TopMAGD[5] para 12 géneros musicales distintos, se

[4] `https://colinraffel.com/projects/lmd/`
[5] `http://www.ifs.tuwien.ac.at/mir/msd/TopMAGD.html`

obtiene la siguiente gráfica en la que se puede ver un patrón de repetición cada 2 y 4 compáses de media.

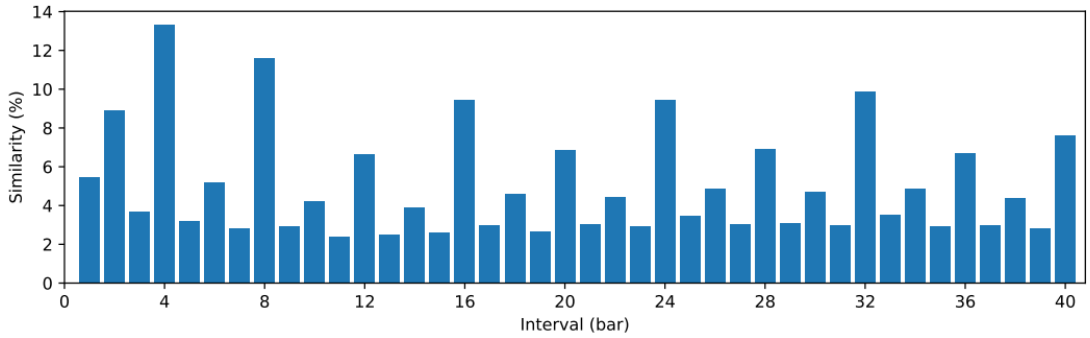


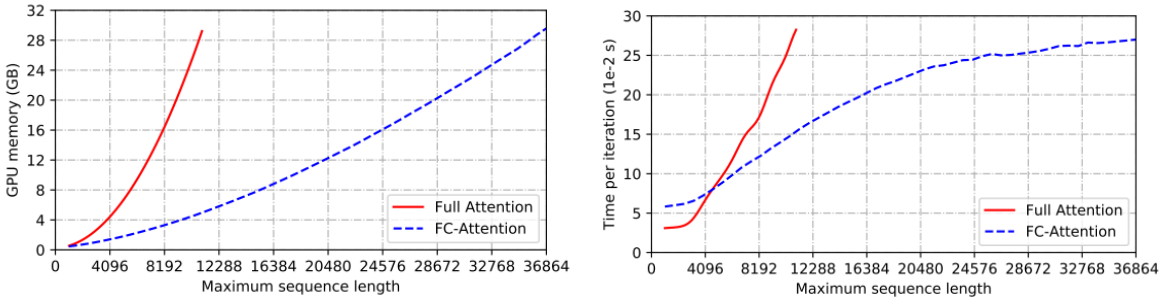
Figura 4.3: Medida de similaridad entre compases para los datos de entrenamiento del Museformer. Tomado de [7]

4.4. Análisis de complejidad

Para finalizar este apartado vamos a observar si realmente el hecho de modificar la capa de atención para implementar los mecanismo de ajuste grueso y fino reducen el orden de operación de $\mathcal{O}(n^2)$ a un orden inferior como puede ser lineal.

Sea n la longitud de la secuencia de tokens, b el número promedio de compases, m la longitud promedio de un compás, y k el número de compases relacionados seleccionados. El orden de operación del paso de resumen (*Summarization*) es $\mathcal{O}(n)$, y el paso de agregación (*Aggregation*) es de $\mathcal{O}((km+b)n) = \mathcal{O}((km+n/m)n)$m, es decir, dependiendo de la longitud de promedio del compás tenemos entre un orden cuadrático y un orden lineal. Como en promedio m es mayor a 100 [7] se reduce en gran medida este orden y, por ende, la complejidad de las operaciones.

Las figuras 4.4a y 4.4b muestran cómo afecta esto a la capacidad y tiempo de entrenamiento en comparación otro tipo de Transformer que tiene complejidad cuadrática.



(a) Memoria GPU utilizada.

(b) Tiempo por iteración.

Figura 4.4: Análisis de tiempo y capacidad para el entrenamiento del Museformer en función de la longitud de secuencia de tokens. Tomada de [7]

Los resultados del entrenamiento, validación e inferencia del Museformer se pueden ver en el papel original [7], pero el hecho de utilizar el mecanismo de atención de ajuste grueso y fino mejora tanto la generación de música en términos de los parámetros de medida utilizados[6] en comparación a otros tipos de Transformer mencionados en dicho papel, y permite tratar secuencias de mayor longitud (Figura 4.4) al no enfocar la atención en todos los tokens.

[6]Se verán en la siguiente sección.

Capítulo 5

Implementación, entrenamiento e inferencia del Museformer

5.1. Implementación

El proceso de implementación consiste en hacer funcionar la red neuronal de manera que se pueda entrenar, hacer inferencia y modificarla a necesidad de los distintos objetivos propuestos. En este caso para implementar el Museformer se ha seguido una serie de pasos, algunos de los cuales se pueden encontrar en su repositorio en *GitHub*[1], y que se detallarán en este apartado junto con las aportaciones y modificaciones realizadas al mismo.

En primer lugar se ha determinado los recursos hardware necesarios para poder implementar de manera eficiente el Museformer, en este caso se ha usado una máquina de la propia universidad con las siguientes características importantes:

- Tarjeta de Vídeo: NVIDIA RTX 3090 24 GB VRAM GDDR6X
- Procesador: Intel Core i7-6700 8 núcleos (16 Hilos) @ 3.40 GHz
- OS: CentOS 7

En segundo lugar se ha determinado que el lenguaje de programación es Python, en concreto la versión 3.8, así que se ha optado por implementar un entorno virtual mediante el gestor de paquetes *Anaconda*[2], aunque por ahorrar espacio en la máquina empleada y para el propósito de este TFG se ha usado su versión ligera *miniconda3*. Una vez creado el entorno, se han instalado todas las dependencias mencionadas en el repositorio del Museformer, junto al *VS Code*[3] para modificar código, *MidiProcessor* para transformar los archivos MIDI a tokens de los propios autores del Museformer,

[1] `https://github.com/microsoft/muzic`
[2] `https://www.anaconda.com/`
[3] `https://code.visualstudio.com/`

triton[4] como alternativa al lenguaje *CUDA* y *Musescore 3*[5] para poder leer y escuchar los archivos MIDI.

Para comprobar el funcionamiento de la implementación se han obtenido las piezas mencionadas por los autores del Museformer a partir de la base de datos *LMD*[6] sin preprocesar, a diferencia de como lo han implementado ellos. Dado que no existía ningún mecanismo para obtener los archivos directamente se ha tenido que escribir un script propio en Python[7] para poder generar un directorio con todos los archivos a partir de una lista en formato texto. Una vez hecho esto se han seleccionado de forma aleatoria unos 50 archivos dentro de ese directorio, se han convertido en tokens, mediante otros dos scripts propios se han separado en el formato estándar de entrenamiento train/valid/test, que en este caso es un 8/1/1, y estos archivos se han preprocesado mediante el un comando de la librería *Fairseq*[8].

Para el resto de tamaños empleados en el entrenamiento el procedimiento ha sido el mismo.

5.2. Entrenamiento

El primer entrenamiento ha sido con 500 datos totales, los cuales se han repartido de la siguiente forma: 8/1/1, es decir, de cada 10 datos, 8 son de entrenamiento, 1 es de validación y 1 es de test. Esto es una forma habitual de entrenar las redes neuronales en general, donde los datos de entrenamiento se usan para ajustar los parámetros de la red, los de validación[9] se usan para ajustar los hiper-parámetros de la red y los de test se usan para comprobar la precisión de la red una vez entrenada.

Los hiper-parámetros utilizados se pueden ver en la tabla 5.1:

Vamos a describir brevemente la utilidad de estos parámetros.

- Update hace referencia al paso completo de un dato de entrada a la red, es decir, el paso forward y backward.

- Update-Frequency es el tamaño del batch, en este caso el gradiente se actualiza cada 5 muestras.

- El Epoch es la cantidad de veces que le pasamos los datos de entrenamiento a la red.

[4] `https://github.com/openai/triton`
[5] `https://musescore.org/es/download`
[6] `https://colinraffel.com/projects/lmd/`
[7] Los códigos de los scripts se pueden ver en el anexo.
[8] `https://github.com/facebookresearch/fairseq`
[9] Las palabras validación y test pueden intercambiarse dependiendo del autor, pero en general se denotan de esta manera.

– Peak Learning Rate (PLR) es la tasa de aprendizaje máxima.

– Warm-up Updates son las actualizaciones en la que la tasa de aprendizaje crece de forma lineal hasta PLR.

– Lr-Scheduler es un planificador que modifica la tasa de aprendizaje a partir de la PLR multiplicando la inversa de la raíz cuadrada del número de actualización actual.

– El optimizador es Adam y su utilidad es complementar a la tasa de aprendizaje para optimizar el entrenamiento, se determina por tres parámetros mencionados en la tabla previa.

– Los tokens per sample indican la cantidad máxima de tokens que puede tener cada dato de entrada a la red.

– Con2con y con2sum indican que compases van a ser de ajuste fino y grueso respectivamente.

Nombre	Valor
Update	1
Update-Freq	5
Epoch	5000
Peak Learning Rate (PLR)	5×10^{-4}
Warm Up Updates	16000
Lr Scheduler	$1/\sqrt{Update_{actual}}$
Adam	$\beta_s = (0{,}9, 0{,}98)$
	$\epsilon = 1 \times 10^{-9}$
	$w_d = 0{,}01$
Tokens per sample	100000
Layer Number	4
con2con	cada 4 compases
con2sum	el resto de compases

Tabla 5.1: Hiper-parámetros entrenamiento 500 datos.

El resultado del entrenamiento se ha medido mediante dos parámetros objetivos que son el error de coste y la perplejidad. El error de coste mide el error entre la predicción de la red y los datos de entrada para ajustar, y la perplejidad es una medida de la calidad de predicción de la red para un token en particular. Tras llegar a 1129 Epochs se cumplió el criterio propuesto de error inferior a 0.1 y los resultados se pueden ver en la tabla 5.2:

El tiempo de entrenamiento ha sido de 24 horas aproximadamente.

Epoch	Loss	Perplexity
1129	0.099	1.07

Tabla 5.2: Resultado entrenamiento con 500 muestras.

En este caso interesa que tanto la pérdida como la perplejidad sean lo más pequeñas posible. Para comprobar si realmente el entrenamiento ha sido útil hacemos la validación con los datos de test mencionados al principio del párrafo, el resultado se puede ver en la siguiente tabla:

Test	Loss	Perplexity
-	1.836	3.57

Tabla 5.3: Validación entrenamiento con 500 muestras.

Como podemos apreciar a pesar de que el error cometido en el entrenamiento era pequeño, al pasarle datos desconocidos a la red no ha sido capaz de interpretar los resultados correctamente y ha cometido muchos fallos. Esto se debe principalmente al hecho de haber utilizado pocos datos de entrenamiento, ya que solamente conoce los datos que se le han pasado y no es capaz de aprender todas las características necesarias para adaptarse a cualquier pieza musical en general. La solución a este problema y en general a cualquier problema de redes neuronales es utilizar una cantidad amplia de datos para que pueda aprender una gran cantidad de características determinantes a la hora de resolver un problema dado.

A continuación vamos a realizar un entrenamiento mucho más extenso en cuanto a la cantidad de datos empleada, y vamos a contrastarlo con el caso anterior para ver como evoluciona la capacidad de la red según los datos de entrada proporcionados[10].

En este caso hemos utilizado 9300 datos totales con la misma división anterior 8/1/1. En la tabla 5.4 se resumen los hiper-parámetros empleados, de los cuales solamente hemos modificado el tamaño del Batch y el número de Epochs ya que para el caso de 500 datos los demás hiper-parámetros ya nos han dado buenos resultados.

Los resultados del entrenamiento se pueden ver en la tabla 5.5:

El tiempo de entrenamiento para este caso ha sido de 72 horas aproximadamente, a un ritmo de 31 minutos por epoch aproximadamente. Esto es debido al hecho de aumentar el tamaño del batch haciendo que el entrenamiento converja de forma más robusta, pero tarde más tiempo.

En cuanto a la validación los resultados se muestran en la tabla 5.6:

[10]Debido a la limitación de Hardware, solamente se proporcionan estos dos casos, pero se puede seguir ampliando hasta la cantidad que queramos y contrastar los resultados

Nombre	Valor
Update	1
Update-Freq	8
Epoch	1200
Peak Learning Rate (PLR)	5×10^{-4}
Warm Up Updates	16000
Lr Scheduler	$1/\sqrt{Update_{actual}}$
Adam	$\beta_s = (0{,}9, 0{,}98)$
	$\epsilon = 1 \times 10^{-9}$
	$w_d = 0{,}01$
Tokens per sample	100000
Layer Number	4
con2con	cada 4 compases
con2sum	el resto de compases

Tabla 5.4: Hiperparámetros entrenamiento 9300 datos.

Epoch	Loss	Perplexity
183	0.533	1.45

Tabla 5.5: Resultado entrenamiento con 9300 muestras.

Como podemos observar la principal diferencia entre ambos casos es la capacidad de adaptación a los datos, es decir, como hemos utilizado más datos de entrenamiento la red no se "sorprende" tanto al introducirle datos nuevos ya que entiende mejor las estructuras de la canciones y se adapta mejor. En cuanto a las pérdidas vemos que son superiores en el caso con más datos y esto es normal ya que la dimensionalidad de los datos es mucho mayor en este caso y tiene que ir adaptándose a todos ellos, además de haber entrenado menos *epochs* que en el otro caso, aunque el tiempo de entrenamiento ha sido mucho mayor. Este segundo problema se arregla entrenando durante más tiempo la red, pero debido a falta de tiempo para ello en la memoria se presentan estos resultados.

5.3. Inferencia

Una vez entrenada la red vamos a comprobar la eficacia de los resultados generando unas piezas en particular. Para ello a partir de ruido en la entrada le indicamos a la red que se encargue de obtener resultados, el procedimiento es el siguiente.

1. Indicamos la carpeta de salida de las piezas generadas.

2. Generamos una semilla para crear el ruido que le vamos a pasar por la entrada.

Test	Loss	Perplexity
-	0.637	1.55

Tabla 5.6: Validación entrenamiento con 9300 muestras.

3. Introducimos el script con la cantidad de piezas a generar y tras un tiempo nos genera un archivo log.

4. Con dos comandos proporcionados por los autores generamos primero los tokens y luego los archivos MIDI.

Los detalles de los comandos se pueden ver en la documentación del Museformer, aquí se va a describir brevemente el proceso de generación de las piezas y los resultados propiamente dichos.

Primero observamos los parámetros del script que se encarga de generar los resultados.

Nombre	Valor
Top-K Sampling	8
Beam	1
nBest	1
Min-Length	2048
Max-Length-b	2048
seed	1

Tabla 5.7: Parámetros para la generación de piezas en el caso de 500 muestras.

En NLP (Procesado Natural del Lenguaje) hay varias formas de generar palabras o tokens a partir de una red entrenada, pero en nuestro caso se usa un método llamado *Top-K Sampling* el cual se encarga de generar el conjunto de los K siguientes tokens más probables a partir del actual y escoge uno de ellos de forma aleatoria. Los parámetros asociados *beam* y *nbest* se utilizan en un algoritmo llamado Beam-Search[11] que consiste en buscar uno o varios caminos (*nbest*) con el conjunto de palabras más probables a partir de una palabra actual, con la profundidad del camino dada por el parámetro *beam*. La figura 5.1 ilustra este proceso de forma más clara con *beam = 2* y *nbest = 1*. En el caso del Museformer necesitamos extraer un token del conjunto de K palabras a una distancia de una unidad, luego se mantienen ambos valores a 1.

El tamaño de las piezas generadas viene dado por los parámetros *Min-Length* y *Max-Length-b*, es decir, se generan siguiendo una distribución uniforme [Min-Length,

[11] `https://en.wikipedia.org/wiki/Beam_search`

Max-Length-b]. En nuestro caso por temas de tiempo se han puesto ambos valores del mismo tamaño y no muy grande para comprobar los resultados generados.

Por último, el parámetro *Seed* se utiliza para generar el número pseudo-aleatorio que le vamos a pasar como input a la red para que genere las piezas deseadas.

Con el entrenamiento de 500 datos se han generado 5 piezas de longitud 2048 tokens, los cuales equivalen aproximadamente a menos de un minuto de música, aunque esto es relativo ya que depende de que tipo de tokens genere la red puede haber más o menos compáses en función de los instrumentos y las notas.

Para la generación a partir del entrenamiento con 9300 datos solamente se ha modificado la longitud de las piezas a 4096 tokens para poder apreciar los efectos de los ajustes grueso y fino a la hora de generar una estructura musical.

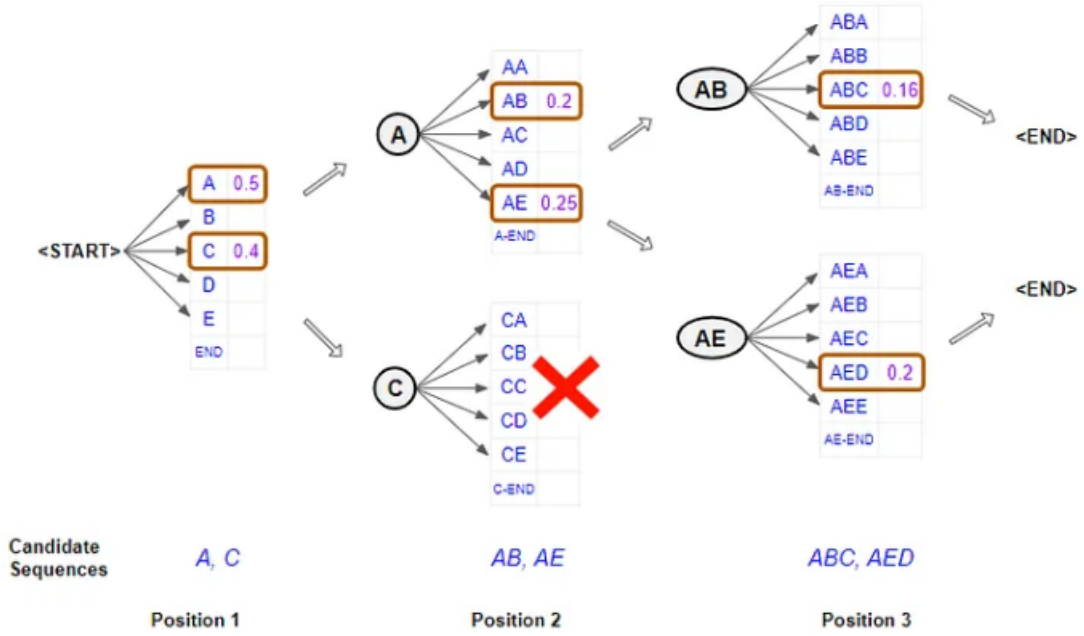
En cuanto a tiempo de generación podemos resumir los distintos casos realizados según el tamaño de token.

Nº Piezas	Nº Tokens	Tiempo(min)
5	1024	6
5	2048	12
5	4096	41

Tabla 5.8: Tiempo de generación para distintos tamaños de piezas.

Estos tiempos son válidos para todo entrenamiento realizado y se puede apreciar que el tiempo se duplica, e incluso se triplica al aumentar el tamaño de los tokens. Si bien es cierto que al aumentar el tamaño de los tokens a la red le cuesta más tiempo generar una pieza, sigue sin ser justificable el tiempo total, ya que es demasiado elevado para este proceso, porque en comparación con otras arquitecturas Transformer a estas les cuesta como mucho unos pocos minutos generar estructuras con muchos tokens, por lo que la conclusión en este caso es que hay algún fallo de implementación por parte de los autores del Museformer en algún punto de la generación.

Para finalizar este apartado, la descripción subjetiva que puedo hacer sobre las 5 piezas a nivel general es que a nivel de ritmo la estructura está bien definida, pero a nivel de armonía no termina de darle coherencia a las piezas, por lo que se nota el efecto de entrenar con pocos datos. Con el segundo entrenamiento sigue ocurriendo algo similar, solo que esta vez la red mezcla los distintos elementos aprendidos de cada canción de entrenamiento quitándole coherencia a las piezas generadas, pero aun así se puede observar la mejora.



Beam Search example, with width = 2 (Image by Author)

Figura 5.1: Ejemplo de algoritmo de Beam-Search. El algoritmo esta sacado de la siguiente página: https://towardsdatascience.com

Capítulo 6

Condicionamiento del Museformer (Teoría)

Hasta ahora hemos aprendido el concepto de una red neuronal, en particular una variación de la arquitectura Transformer, la hemos implementado, entrenado y generado música con ella, pero esto ha sido a partir de generar un ruido en la entrada de la red, es decir, a su voluntad. Pero, ¿Y si queremos indicarle nosotros que queremos una pieza musical basada en cierto instrumento, artista, género, ...? En este caso hay que hacer lo que se conoce como condicionamiento de la red, es decir, tenemos que modificar los datos de manera que incluyan información extra como título de la obra, autor, género, etc. El procedimiento se va a describir de forma teórica a continuación.

El primer paso es usar un método de codificación que nos permita incluir información adicional a la hora de generar los tokens que le vamos a pasar a la red, en el caso de Museformer tenemos el método *REMIGEN* que consiste en codificar la información MIDI en tokens de compás, posición relativa, tempo, instrumento, pitch, duración y velocidad o intensidad. También existen otros métodos como *REMI+*, *Compound Word*, *TSD*, *Octuple*, etc[1]. En este caso aprovechando que se usa Python en el entorno, se puede instalar el paquete *Miditok* [8], el cual contiene los métodos mencionados anteriormente junto a algunos más, e incluso se puede crear un método propio.

Como segundo paso hay que plantear uno de los dos siguientes escenarios, que no necesariamente son los únicos que se pueden plantear en general.

1. Sin modificar la estructura del Museformer, entrenamos la red con los nuevos tokens añadidos de meta datos, los cuales irán colocados al principio de cada pieza musical y tendrían su propia estructura previamente codificada en el paso anterior. El cambio sería para la generación ya que en vez de pasarle como dato de entrada un número aleatorio, habría que pasarle uno o varios tokens de meta datos

[1]Explicación detallada en `https://miditok.readthedocs.io/en/latest/tokenizations.html`

a partir de un texto dado como input, traducirlo a los tokens correspondientes y, a partir de estos, la red tendría que buscar en el espacio de datos los que tengan mayor similitud para generar a partir de ese punto tal como lo hace por defecto mediante el algoritmo *Sampling Top k*.

2. El escenario anterior plantea muchos problemas, entre ellos la variabilidad de los meta datos de entrada, la implementación de la nueva generación de datos, ya que habría que estudiar las librerías *Fairseq-generation* y *Fairseq-interactive* empleadas para ver como se pueden traducir palabras a tokens, si es posible asociar los tokens al espacio de datos, etc. Por lo que en este caso se puede optar por modificar la arquitectura manteniendo la estructura original, pero en vez de utilizar el método de regularización empleado por parte de Museformer se optaría por un método de contraste explicado en *Text Conditioning*), que consiste en codificar una imagen (en nuestro caso una pieza musical) y un texto de entrada por separado (mediante un Transformer más reducido en tamaño por ejemplo), y mediante un pre-entrenamiento inicial, se pueden entrenar las redes para asociar los textos de entrada con sus respectivas piezas. Luego a la hora de la generación, mediante el texto de entrada correspondiente generaría las piezas más cercanas mediante el algoritmo utilizado por Museformer. Este método permite mitigar la variabilidad de los meta datos de entrada, pero a cambio introduce modificaciones y no se sabe a priori el impacto que puede tener de manera computacional y temporal en cuanto a entrenamiento y generación.

Como tercer paso, a la hora de la generación, hay que cambiar el método aleatorio por uno de entrada por texto y que la red sea capaz de entender ese texto y traducirlo a la información adicional de la pieza, es decir, si indicamos que genere una pieza de Pop basada en un autor determinado, la red tiene que interpretar ambas informaciones de manera correcta y relacionarlas con los tokens de información adicional respectivos.

Una problemática que genera este nuevo planteamiento es la necesidad de una base de datos muy extensa con piezas musicales en formato MIDI, los cuales además deberán tener bastante información de tipo meta datos con nombres de autor(es), género(s), duración, etc. La base de datos *TopMAGD*[2] contiene una gran cantidad de música catalogada por géneros, pero no contiene artistas y otro tipo de información extra, por lo que a la hora de implementar este condicionamiento lo restringiríamos al género en cuestión.

El otro problema que genera esta implementación adicional es la necesidad de una gran cantidad de tiempo para implementar cada uno de los tres pasos mencionados

[2] `https://www.ifs.tuwien.ac.at/mir/msd/TopMAGD.html`

anteriormente, junto a su correspondiente corrección de bugs o errores, además del requerido para construir la propia base de datos extensa con toda la información adicional.

Debido a las dos problemáticas mostradas anteriormente este apartado se va a quedar de forma teórica y, finalmente, no se ha implementado en este trabajo.

Capítulo 7

Conclusiones

Museformer es una red/herramienta que proporciona un método alternativo y eficiente de generar piezas musicales bien estructuradas musicalmente de larga duración dentro del ámbito de los Transformers, debido a que reduce la carga computacional y permite tratar una mayor cantidad de datos, además de reducir el tiempo de computación necesario para entrenarla. Como hemos podido observar los resultados mejoran a partir de introducirle más datos de entrenamiento a la red ya sea de manera objetiva, reduciendo la perplejidad, o de manera subjetiva, escuchando las piezas resultantes. Aun así se puede mejorar más el Museformer llegando a condicionar la red para que sea capaz ya no de generar piezas aleatoriamente, sino indicándole información extra como autor, género, etc. En cuanto a la lista de objetivos secundarios propuestos en la sección 1 se ha llegado a cumplir todos los propuestos al máximo posible, excepto la implementación del condicionamiento del Museformer. A pesar de ello, se ha desarrollado un planteamiento teórico de los pasos y los detalles necesarios para implementar el condicionamiento a partir del trabajo realizado.

Para finalizar a nivel personal este trabajo ha sido tanto de formación en materia casi totalmente desconocida como lo es el aprendizaje profundo, como aprendizaje de implementación en materia de entender de manera básica el lenguaje de programación Python, LaTeX, la distribución CentOS dentro de Linux para el manejo de la máquina del laboratorio, junto a una cantidad elevada de horas para solucionar problemas surgidos durante esta fase, entender de manera práctica la utilidad de los parámetros de entrenamiento e inferencia del Museformer y de forma teórica las posibles modificaciones que se le podrían hacer para implementar el condicionamiento de la red. Han sido unos meses largos de trabajo, pero gracias a este esfuerzo ahora tengo un entendimiento en un campo que hasta hace poco era desconocido para mí, junto a unas habilidades básicas en otros entornos que no he tratado durante los años que llevo en telecomunicaciones.

Capítulo 8

Bibliografía

[1] Yann LeCun and Alfredo Canziani. Deep learning course. `https://atcold.github.io/NYU-DLSP21/`, 2021.

[2] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[3] G.H. Ball and D.J. Hall. Isodata, a novel method of data analysis and pattern classification. *Stanford Research Institute, Menlo Park*, 1965.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *NIPS*, 2017.

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[6] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014.

[7] Botao Yu, Peiling Lu, Rui Wang, Wei Hu, Xu Tan, Wei Ye, Shikun Zhang, Tao Qin, and Tie-Yan Liu. Museformer: Transformer with fine-and coarse-grained attention for music generation. *Advances in Neural Information Processing Systems*, 35:1376–1388, 2022.

[8] Nathan Fradet, Jean-Pierre Briot, Fabien Chhel, Amal El Fallah Seghrouchni, and Nicolas Gutowski. MidiTok: A python package for MIDI file tokenization. In *Extended Abstracts for the Late-Breaking Demo Session of the 22nd International Society for Music Information Retrieval Conference*, 2021.

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Código propio implementado mediante scripts de Python

El objetivo de este anexo es proporcionar el código de los scripts utilizados para agilizar el proceso de procesamiento de los datos a la hora de entrenar la red del Museformer. En estos scripts los directorios están referidos al sistema operativo de Windows, ya que el tratamiento de datos se ha realizado desde mi maquina personal, además de incluir una ruta genérica que habría que sustituir por la asociada al directorio donde se haya implementado Museformer en caso de querer probarlos. Para Linux o Mac habría que consultar el formato de directorio que tengan.

Como primer script tenemos el que se encarga de filtrar las canciones utilizadas por parte del Museformer desde la base de datos original LMD.

```
# Import Module
import os

# Folder Path
path = "data"

# Change the directory
os.chdir(path)

# Read data
with open("meta/train.txt","r") as train_file:
    train_data = train_file.read().split('\n')
with open("meta/test.txt","r") as test_file:
    test_data = test_file.read().split('\n')
with open("meta/valid.txt","r") as valid_file:
    valid_data = valid_file.read().split('\n')

total_data = train_data + test_data + valid_data

# Move necesary data from directory A to B and check if it is doing
# correctly.
i = 1
total_data_prime = total_data[1:len(total_data)]
for elem in total_data_prime:
    src_path = 'X:\\...\\muzic\\museformer\\data\\lmd_full\\'+elem
    dst_path = 'X:\\...\\muzic\\museformer\\data\\midi_filt\\'+elem
```

```
    print(i,": "+elem)
    i = i + 1
    os.rename(src_path,dst_path)

#Just for debugging purposes.
print(len(total_data))
```

Como segundo script tenemos el que se encarga de generar los ficheros train, test y valid para el paso de split data.

```
#Necesary libraries.
import random
import os

#Define path
path = "X:\\...\\muzic\\museformer"
os.chdir(path)

with open("data\\token\\token_names.txt", "r") as f:
    data = f.read().split('.txt\n')

#Shuffling data to make it random and not ordered.
random.shuffle(data)

# From here to the end of code it's just data organization and
# writting on file.
train_L = round(len(data) * 0.8)
test_L = round(len(data) * 0.1)
valid_L = round(len(data) * 0.1)

train_data = data[0:train_L]
test_data = data[train_L+1:train_L+test_L]
valid_data = data[train_L+test_L+1:train_L+test_L+valid_L]

meta = path + "\\data\\meta"
os.chdir(meta)

train = open("train.txt","w")
test = open("test.txt","w")
valid = open("valid.txt","w")

for data in train_data:
    train.write(data)
    train.write("\n")
train.close()


for data in test_data:
    test.write(data)
    test.write("\n")
test.close()

for data in valid_data:
    valid.write(data)
    valid.write("\n")
valid.close()
```

El tercer y último script se encarga de eliminar las extensiones .txt que le añade

MidiProcessor a los tokens, los cuales provocan que el paso de split no se ejecute correctamente.

```
import os

path = "X:\\...\\muzic\\museformer\\data\\meta"
os.chdir(path)

# Change filename of the files to v"...".txt before executing this
# code just in case to prevent erasing the files.
with open("vtrain.txt", "r") as f:
    Ltrain = f.read().split('.txt\n')
    Ltrain.clear(".txt") #This is for the final element.

with open("vtest.txt", "r") as f:
    Ltest = f.read().split('.txt\n')
    Ltest.clear(".txt")

with open("vvalid.txt", "r") as f:
    Lvalid = f.read().split('.txt\n')
    Lvalid.clear(".txt")

train = open("train.txt","w")
test = open("test.txt","w")
valid = open("valid.txt","w")

for index,element in enumerate(Ltrain):
    train.write(element)
    if(index != len(Ltrain-1)):
        train.write("\n")
train.close()

for index,element in enumerate(Ltest):
    test.write(element)
    if(index != len(Ltest-1)):
        test.write("\n")
test.close()

for index,element in enumerate(Lvalid):
    valid.write(element)
    if(index != len(Lvalid-1)):
        valid.write("\n")
valid.close()
```