Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Democratizando el fine-tuning: Transfer Learning eficiente con modelos pre-entrenados de habla para identificación de hablantes

Tesis de Licenciatura en Ciencias de la Computación

Erik Ernst

Directora: Luciana Ferrer, ICC-CONICET/UBA
Codirector: Leonardo Pepino, ICC-CONICET/UBA
Buenos Aires, 2025

TRANSFER LEARNING EFICIENTE CON MODELOS PRE-ENTRENADOS DE HABLA PARA IDENTIFICACIÓN DE HABLANTES

En los últimos años, el avance acelerado del aprendizaje automático ha transformado el procesamiento de datos, ofreciendo modelos pre-entrenados capaces de capturar representaciones semánticas complejas. Sin embargo, esta evolución ha venido acompañada de un creciente costo computacional, centralización de recursos en pocas instituciones y un enfoque orientado a maximizar los resultados, dejando en segundo plano la comprensión de las razones detrás de muchas decisiones de diseño de la arquitectura y el entrenamiento de modelos.

En esta tesis, exploramos técnicas eficientes de *transfer learning* aplicadas a modelos pre-entrenados de habla para la tarea de identificación de hablantes (*Speaker Identification*, SID). En este marco, nuestro objetivo principal fue comprender en profundidad el impacto de configuraciones clave en el diseño y entrenamiento de modelos. Primero, experimentamos con múltiples arquitecturas e hiperparámetros con el fin de encontrar el mejor modelo downstream utilizando WavLM Base+ como modelo upstream. En este proceso, analizamos factores como la tasa de aprendizaje, diferentes mecanismos de *pooling* y normalización. Entre nuestros hallazgos más significativos, demostramos que la incorporación de mecanismos de atención en el *pooling* temporal y de capas puede ofrecer ventajas significativas, alcanzando resultados estado del arte con una cantidad de parámetros ampliamente inferior. A su vez, investigamos técnicas de *full fine-tuning* y de fine-tuning eficientes en parámetros (*Parameter Efficient Fine-Tuning*, PEFT), en particular, LoRA y las ventajas que puede traer su uso.

Palabras claves: Identificación de Hablantes, Modelos Pre-entrenados de Habla, Procesamiento del Habla, Fine-Tuning, Transfer Learning, Transformers.

EFFICIENT TRANSFER LEARNING FOR PRE-TRAINED SPEECH MODELS IN SPEAKER IDENTIFICATION

In recent years, the rapid advancement of machine learning has transformed data processing, enabling pre-trained models to capture complex semantic representations. However, this progress has been accompanied by increasing computational costs, a centralization of resources within a few institutions, and a predominant focus on maximizing results, often at the expense of understanding the rationale behind many architectural and training design decisions.

In this thesis, we explore efficient *transfer learning* techniques applied to pre-trained speech models for Speaker Identification (SID). Within this framework, our primary objective was to gain a deeper understanding of the impact of key configurations in model design and training. We conducted extensive experimentation with multiple architectures and hyperparameters to identify the optimal downstream model, using WavLM Base+ as the upstream model. In this process, we analyzed factors such as learning rate, different *pooling* mechanisms, and normalization techniques. Among our most significant findings, we demonstrated that the incorporation of attention mechanisms in temporal and layer-wise *pooling* can provide substantial benefits, achieving state-of-the-art results with a significantly smaller number of parameters. Furthermore, we investigated *full fine-tuning* techniques and Parameter-Efficient Fine-Tuning (PEFT) approaches, specifically LoRA, and explored the benefits it can offer.

Keywords: Speaker Identification, Pre-trained Speech Models, Speech Processing, Fine-Tuning, Transfer Learning, Transformers.

AGRADECIMIENTOS

Agradecimientos, que diga unas palabras; me vería con las manos en la cara, parado tan lejos de todo con la vergüenza empujando mi cabeza al piso. No es para tanto, tantos lo hicieron mejor, pensaría. Tendría razón. No es para tanto si es una lista de logros, si solo es llegar al próximo lugar.

Mis ojos miran hacia arriba llenos de felicidad, de gracias, de palabras, de esperanzas que no entiendo. El haber llegado, la vida entre las cosas, el camino. En este último tiempo, aprendí lo que tantas veces se me fue olvidando. Ni la profesión, ni las letras, ninguna de las obligaciones de la vida que llevo con mis tontas ambiciones. Los amigos, la familia, la íntima compañía. La pregunta, la risa, el abrazo, la caricia, una mirada. Con mil vidas viajaría años por el mundo, me tomaría cada tren que existe, tendría mil profesiones distintas. Pero solo tengo una y la quiero cerca de las personas que amo.

Cuando quepa una vida entre mis recuerdos solo el amor me traerá el alivio. Gracias.

Gracias a mis directores, Leo y Luciana. Gracias a Leo, por toda la paciencia, por todo el tiempo que me dedicaste y por enseñarme tanto. Es un enorme privilegio trabajar con ustedes y aprender en el camino. Pensaba que todas estas cosas las hacían otros, muy lejos y en grandes lugares, gracias por acercarme a ellas, hice mucho más de lo que sabía imaginar.

Gracias a Agustín y Ramiro por su enorme generosidad, por introducirme en la investigación y contagiarme su pasión, que ahora también es mía.

Gracias a todo y todos los que estuvieron antes e hicieron tanto más fácil que yo llegue hasta acá. Gracias a la universidad pública, a la UBA, a sus docentes, a esas clases inolvidables, al ILSE, al Scholem Aleijem.

Gracias a todos los amigos y compañeros de Exactas y Computación, a Sujito, a Dani, Pato, Franco, Colo, Ale, Abi, Blas, Luciano, Luciana, Ivo, Charles, Vicky.

Gracias y gracias a todos los que amo, mi familia y mis amigos. Mamá, papá, Iván, abuelos, tíos y primos. Wander. Mi otra familia, Vicky, Ana Pau, Facu, Fausti, Guille, Lay, Frankie. Iancho, Tomi, Cande, Luli.

Índice general

1. INTRODUCCIÓN

En los últimos años la comunidad científica ha experimentado una gran cantidad de avances en el campo del aprendizaje automático, que con una enorme vertiginosidad entregan cada vez mejores resultados en las aplicaciones más diversas. En campos donde se utilizaban técnicas variadas y específicas a sus problemas, se comienza a observar una convergencia en el uso de herramientas de aprendizaje automático como *redes neuronales*, *fine-tuning*, *embeddings*, *atención*. Tanto los modelos como los datos con los que son entrenados han crecido exponencialmente, logrando una revolución en la representación de distintos datos como textos, audios e imágenes, llegando incluso a encontrar cierto sentido *semántico* en estas representaciones.

Sin embargo, si bien actualmente es mucho más sencillo llegar a grandes resultados, esta evolución ha hecho que los recursos necesarios para desarrollar estos modelos sean cada vez más restrictivos. Mientras estos modelos crecieron en complejidad, la capacidad de entender sus resultados se ha visto afectada, y en general, ignorada. Asimismo, las limitaciones de recursos hicieron que la investigación de estos modelos grandes se concentre en pocas empresas e instituciones. Esto último y el afán de poseer el mejor modelo (o la supuesta carrera por una inteligencia artificial general) devino en que el foco de la investigación y las publicaciones sea más los resultados que la comprensión de los mismos. En reiteradas ocasiones, muchas de las configuraciones del entrenamiento son expuestas sin ser justificadas, y las razones viven en la intuición de quienes realizan los experimentos.

A raíz de lo expuesto, consideramos necesario democratizar estos conocimientos implícitos. En este trabajo nos proponemos emprender un camino de construcción de uno de estos modelos compartiendo los aprendizajes que sustentan las intuiciones posteriores. Elegiremos un ambiente acotado donde podremos explorar algunos de los fundamentos de la arquitectura y entrenamiento de este tipo de modelos.

Los datos que utilizaremos son audios, más precisamente, audios de habla humana. El aprendizaje de la representación de audios ha quedado un tanto rezagado en comparación a imágenes y texto. En particular, los audios de habla se ven expuestos a diversas complejidades, como exponen de manera clara Mohamed et al. (2022). Son datos secuenciales, como el texto, pero su segmentación o tokenización es un problema en sí mismo. Más aún, las señales de habla son continuas por lo que no es directa la definición de un vocabulario finito. Por otro lado, una representación semántica de un audio de habla debería tener información muy variada, hasta incluso ortogonal, ya que las tareas de procesamiento de habla son muy diversas: tareas como transcripción requieren extraer información del contenido mientras que ignoran la información paralingüística, y otras tareas como identificación de hablantes se concentran en aquella información paralingüística, el cómo está dicho el contenido (prosodia, timbre de la voz, entre otros).

Por otra parte, la tarea que seleccionamos para resolver es *identificación de hablantes* (SID, por sus siglas en inglés). Esta tarea consiste en clasificar al hablante presente en un audio dado. Si bien puede haber ruido de fondo, entre los que puede haber voces, estos audios deberían poseer una voz principal claramente marcada. La elección de esta tarea se debe a que es una de las tareas mejor definidas en habla, las etiquetas no poseen ambigüedades ni subjetividad como puede ser el caso de reconocimiento de emociones. A su vez, como el proceso de etiquetado es sencillo, hay una gran disponibilidad de datos

que se pueden utilizar para entrenar.

SID es una de las tareas más fundamentales en el procesamiento del habla. En cuanto a su importancia, se pueden destacar algunos puntos clave. La primera y más importante, es que, como mencionan Lin et al. (2024), identificación de hablantes junto con reconocimiento de fonemas, son dos tareas que por sí solas sirven como buenos estimadores de la performance de un modelo en el SUPERB *benchmark* (Yang et al., 2021) [1]. Nótese que estas dos tareas cubren el espectro de contenido y forma mencionado anteriormente. SID cubriría lo relativo a la forma así como elementos para-lingüísticos que caracterizan la identidad individual de cada hablante. Adicionalmente, está la importancia para el desarrollo de modelos de verificación de hablantes, para los cuales es usual comenzar con una fase de entrenamiento previo en SID.

De esta manera, en primer lugar, en el capitulo 2 se presenta el marco teórico necesario para comprender la investigación. En segundo lugar, en el capítulo 3 se presenta el ambiente de experimentación: los datos y los recursos que se van a utilizar. Luego, en el capítulo 4 comienza la exploración buscando el mejor modelo downstream para la tarea seleccionada. La experimentación continúa en una segunda parte en el capítulo 5, donde se entrena también al modelo upstream. Finalmente, en el capítulo 6 se presentan las conclusiones del trabajo así como también posible trabajo futuro.

[1] El *benchmark* más completo y utilizado para evaluar modelos pre-entrenados de habla en las tareas más populares.

2. MARCO TEÓRICO

En este capítulo, introducimos los conceptos fundamentales de la tesis necesarios para el entendimiento de todo el trabajo. El tratamiento de estos conceptos es acotado por la naturaleza de la sección, y está fuertemente basado en los trabajos de Mitchell (1997); James et al. (2023); Goodfellow et al. (2016).

El presente trabajo se enmarca en el campo del **aprendizaje automático** o ***machine learning***, una disciplina que se centra en el desarrollo de modelos capaces de aprender patrones y representaciones a partir de datos. Esta disciplina se encuentra dentro del campo de la inteligencia artificial (IA) o inteligencia computacional. A diferencia de lo que se estudia en otras áreas de IA donde el aprendizaje es basado en conocimiento, el aprendizaje automático es basado en datos.

Por su parte, los problemas que se resuelven en el área suelen dividirse en dos categorías: **regresión** y **clasificación**. Los problemas de regresión son aquellos donde se tiene como resultado una predicción de un valor, el ejemplo clásico es la predicción del precio de un inmueble dadas sus características. En contraparte, los problemas de clasificación tienen como resultado una clase o categoría; SID es un ejemplo de ello, donde se espera la identificación de una persona como resultado.

A su vez, otra subdivisión que surge en aprendizaje automático es el aprendizaje **supervisado** y el **no supervisado**. En el aprendizaje supervisado se tiene para cada entrada una etiqueta o resultado esperado asociado. El objetivo es aprender una función que, a partir de observaciones etiquetadas, relacione las entradas con sus resultados esperados, de manera que pueda generalizar para predecir los resultados en nuevas entradas. Métodos como regresiones lineales, regresiones logísticas y *support vector machines* están dentro del dominio de esta área. En contraposición, en aprendizaje no supervisado no se posee una respuesta asociada a la entrada. En estos contextos se busca encontrar relaciones entre las entradas, siendo una de las herramientas más populares en estos casos el *clustering*, que separa automáticamente en grupos a las entradas. Veremos una técnica dentro del aprendizaje no supervisado llamada ***Self-Supervised Learning***, que es usada para el desarrollo de ***Foundation Models***, modelos que pueden representar o transformar las entradas en una señal, o *embedding*, que facilite la resolución de tareas con respecto a utilizar directamente las entradas.

A continuación, presentaremos los temas principales dentro de la disciplina que son el núcleo de este trabajo. Abordaremos ***redes neuronales artificiales***, así como mecanismos y arquitecturas como ***Atención*** y ***Transformers***. También introduciremos el campo de ***Representation Learning*** y el uso de ***Transfer Learning***.

2.1. Redes Neuronales Artificiales

Las redes neuronales artificiales, o simplemente redes neuronales, son la piedra fundamental del aprendizaje profundo. Estos modelos tienen como objetivo aproximar funciones. En problemas de clasificación, por ejemplo, la red aprende a aproximar una función que asocia cada entrada con su correspondiente etiqueta. Más aún, Cybenko (1989) muestra que una red neuronal es capaz de aproximar la mayoría de las funciones y su error en la aproximación tiene una cota superior.

En su concepción más simple, la unidad fundamental de una red neuronal es el *perceptrón* o *neurona*. Dada una entrada $X = x_1, x_2, ...x_n \in \mathbb{R}^n$, el perceptrón calcula la siguiente función:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) = f(W^T X + b) \tag{2.1}$$

Donde, $W = [w_1, w_2, ..., w_n]^T$ son los pesos, b es el *bias* o sesgo, y f es la función de activación. Como se puede observar, la función es muy similar a la que se calcula en una regresión lineal, a excepción de la función de activación que se le aplica a la suma de la combinación lineal de la entrada con el *bias*. Por su parte, la función de activación es una transformación no lineal que le permite a los modelos capturar relaciones complejas, no linealidades e interacciones. Específicamente, existen diversos ejemplos de función de activación, la más clásica es la función sigmoidea aunque hoy en día la más popular es ReLU ($f = max(0, x)$) y funciones derivadas de ésta. De esta manera, durante el entrenamiento se ajustarán los valores de los pesos y sesgos, mientras que f es una elección de diseño.

Dada una neurona como la presentada anteriormente, se pueden conectar neuronas entre sí formando una red neuronal denominada perceptrón multicapa (MLP, por sus siglas en inglés). En cuanto a su arquitectura, las neuronas en una MLP se organizan en una secuencia de capas, donde una capa es un conjunto de neuronas que operan en paralelo. Más aún, una MLP consiste en tres tipos de capas: una *capa de entrada* que recibe los datos originales, una o más *capas ocultas* que realizan transformaciones intermedias, y una *capa de salida* que produce la predicción final.

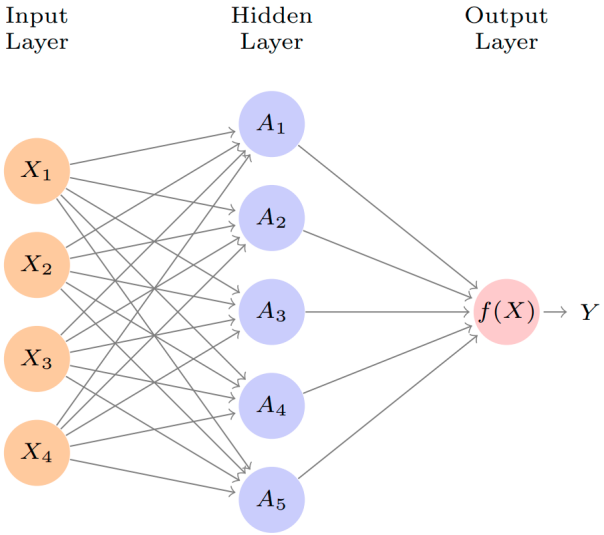


Fig. 2.1: Una red neuronal con una única capa oculta. La capa oculta computa las activaciones $A_1, ...A_5$, transformaciones no lineales de la combinación lineal de la entrada $X_1, ...X_4$. Figura extraída de James et al. (2023)

En cada capa, las neuronas calculan su activación siguiendo la misma fórmula que el perceptrón. Estas activaciones se propagan a través de la red, donde la salida de cada neurona sirve como entrada para las neuronas de la capa siguiente. De este modo, se define una composición jerárquica de funciones que puede expresarse como:

$$
\begin{aligned}
a_1 &= f_1(W_1 X + b_1) \\
a_i &= f_i(W_i a_{i-1} + b_i), \quad \forall i = 2, \ldots, L \\
y &= a_L
\end{aligned} \tag{2.2}
$$

donde a_i es la activación de la capa i, L es el número de capas, W_i son las matrices de pesos para cada capa, b_i son los vectores de sesgos, y f_i son las funciones de activación respectivas a cada capa.

Respecto a su estructura, cuando en la unión de neuronas cada una obtiene como entrada la salida de una capa anterior (enlaces *forward*) o la entrada de la red, la red es llamada *feed-forward network* ya que el flujo de información en estas redes es siempre hacia adelante. Por otra parte, cuando cada neurona de una capa está conectada con todas las neuronas de la capa siguiente, estas capas se denominan completamente conectadas o densas (*fully-connected layers* o *dense layers*), siendo este tipo de arquitectura el más común en las redes feed-forward. Asimismo, si estas redes se extienden con conexiones hacia atrás (*feedback*), de manera que una neurona puede recibir la salida de una neurona posterior, se denominan redes neuronales recurrentes (RNNs). Por último, todos los valores que son ajustados durante el entrenamiento de las redes neuronales se denominan *parámetros*, en cambio, los que no se ajustan son *hiperparámetros*, y su elección es parte del diseño del modelo.

Las dimensiones de la capa de entrada y de salida están limitadas respectivamente por la definición del problema, esto no ocurre con las capas ocultas, cuya dimensión es un hiperparámetro más llamado *tamaño oculto*. En general, cada capa puede tener una dimensión diferente, en cuyo caso no hay un único tamaño oculto, sino uno por capa. En este trabajo, al mencionar un único tamaño oculto estaremos refiriéndonos a utilizar el mismo tamaño oculto para todas las capas. A su vez, la cantidad de capas en una MLP es otro de los principales hiperparámetros. El término *red neuronal profunda* se acuña cuando la red tiene múltiples capas ocultas. De esta manera, se suele hablar del ancho de la red haciendo referencia al tamaño oculto, y el largo refiere a la cantidad de capas.

Resta explicar cómo se ajustan los parámetros de una red neuronal durante el entrenamiento. En primer lugar, debemos notar que dados valores fijos para los parámetros de la red, ésta calcula una función que es la composición de las funciones que calcula cada neurona, como se mostró en 2.2. En segundo lugar, para entrenar redes neuronales se define una **función de pérdida**, una medida del error en la estimación. Hay múltiples funciones de pérdida posibles, las clásicas son error cuadrático medio para regresión y entropía cruzada para clasificación. Teniendo en cuenta estos dos factores, el objetivo del entrenamiento de las redes neuronales va a ser minimizar el error en la estimación. Así, se tiene que el problema de entrenar redes neuronales es un problema de búsqueda de los parámetros que minimicen el error. La clave de los dos puntos mencionados es que la función de pérdida va a ser diferenciable, por lo que se puede computar el gradiente. El gradiente de una función en un punto dado indica la dirección de máximo crecimiento de esa función. Por lo tanto, para minimizar la función de pérdida podemos mover los parámetros en la dirección donde la función decrece más rápidamente, que es precisamente la dirección opuesta al gradiente.

Sobre estas bases, funciona el algoritmo de **descenso por el gradiente**, entre otros algoritmos, que iterativamente calcula el gradiente del error sobre todos los datos, ajusta los parámetros y sigue iterando hasta converger. Para calcular el gradiente, el algoritmo que se utiliza es **backpropagation**, que aprovechando la estructura jerárquica de las redes

neuronales, utiliza la regla de la cadena para calcular de manera eficiente los gradientes de la función de pérdida con respecto a los parámetros de cada capa. De esta manera en cada iteración se ajustan los valores de los parámetros siguiendo la siguiente ecuación:

$$\theta^{t+1} = \theta^t - \eta \nabla L(\theta^t, \mathcal{D}) \tag{2.3}$$

θ es el conjunto de todos los parámetros del modelo, mientras η se denomina la *tasa de aprendizaje* y $\nabla L(\theta^t, \mathcal{D})$ es el gradiente de la función de pérdida L con respecto a θ^t y a los datos de entrenamiento $\mathcal{D}$. El superíndice t indica el instante de tiempo o paso (*step*, cantidad de actualizaciones). La tasa de aprendizaje es un hiperparámetro de gran importancia en el entrenamiento que controla el tamaño del paso que se da en la dirección opuesta al gradiente durante la actualización de los parámetros.

Por último, debemos mencionar que el descenso por el gradiente resulta ineficiente por lo que en la práctica se utiliza una variación llamada **descenso por el gradiente estocástico**. El proceso de calcular el gradiente sobre todo el conjunto de datos para cada actualización de parámetros es muy costoso computacionalmente, especialmente considerando que se requieren numerosas iteraciones para alcanzar la convergencia. Es por esto que la variante mencionada propone que en cada paso se seleccione un subconjunto (generalmente, aleatorio) llamado lote (*batch*) y que la actualización se haga en base al gradiente promedio en ese lote. De esta forma, la ecuación de actualización de los parámetros (2.3) en vez de calcular la función de pérdida sobre todos los datos lo hará exclusivamente sobre un lote. A su vez, la tasa de aprendizaje adquiere una mayor complejidad porque pondera la importancia del lote, *i.e.*: un η muy grande sobreajustará a cada lote limitando la capacidad de generalizar a todo el conjunto de entrenamiento.

2.1.1. Mecanismos de Atención y Transformers

En muchos problemas de aprendizaje automático, los datos pueden representarse como vectores independientes, lo que permite modelarlos eficazmente con redes MLP. Sin embargo, en áreas como el procesamiento de habla, los datos tienen una estructura secuencial, donde cada punto de la señal guarda una relación de orden con los puntos anteriores y posteriores. Modelar estas dependencias es crucial para capturar patrones temporales y contextuales. Para abordar este desafío, es común dividir la señal en *frames*. Un frame es una ventana o segmento temporal corto —generalmente entre 20 y 40 ms— del audio original que se utiliza como unidad básica de procesamiento en los modelos de procesamiento de audio. Los MLPs no están diseñados para capturar relaciones entre frames, ya que procesan cada entrada de forma aislada. Más aún, modelar estas secuencias de frames trae grandes desafíos, ya que:

1. La longitud de las secuencias es variable y puede ser muy extensa.
2. Existen dependencias a corto y largo plazo entre distintos frames.
3. La información relevante no está distribuida de manera uniforme a lo largo de la secuencia.

Esto ha llevado al desarrollo de arquitecturas especializadas para secuencias, partiendo de modelos más simples basados en MLPs como las Time-Delayed Neural Networks (Waibel et al., 1989; Snyder et al., 2015), como las ya mencionadas RNNs (Graves et al., 2013),

CNNs temporales (Abdel-Hamid et al., 2013) y, más recientemente, los mecanismos de atención y Transformers, que permiten modelar de manera más efectiva las dependencias temporales en la señal de habla.

El mecanismo de atención, introducido por Bahdanau et al. (2014) en el contexto de traducción automática, ha revolucionado el campo del aprendizaje profundo al permitir que los modelos se enfoquen dinámicamente en diferentes partes de la entrada según la tarea. La atención ha demostrado ser particularmente útil en secuencias largas, donde redes como RNNs tienen limitaciones significativas debido a problemas como el desvanecimiento del gradiente y la capacidad limitada de capturar dependencias a largo plazo (abordando directamente el desafío 2 mencionado anteriormente). En términos generales, el mecanismo de atención permite a los modelos calcular una representación ponderada de la entrada, donde los pesos reflejan la importancia relativa de cada elemento (respondiendo al desafío 3: distribución no uniforme de información).

Tres años después, Vaswani et al. (2017) introduce los *Transformers*, una red neuronal que se centra en el mecanismo de atención, eliminando por completo la necesidad de estructuras recurrentes o convolucionales. Fundamentalmente, esto se logra a través de una arquitectura que combina múltiples capas de auto-atención (*Self-Attention*) con redes *feed-forward* posicionadas entre ellas. En *Self-Attention* lo que se pondera es la importancia de las partes de la entrada misma. Formalmente, se expresa de la siguiente manera:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{2.4}$$

Q (*queries*, consultas), K (*keys*, claves) y V (*values*, valores) son matrices que se obtienen al proyectar linealmente la entrada, mientras que d_k actúa como un factor de escalado. El resultado de $QK^\top$ obtiene el nombre de matriz de atención, y sus valores se transforman al rango $(0, 1)$ mediante la función *softmax*, que también asegura que la suma de los valores sea 1. Conceptualmente, se puede interpretar la operación como un promedio pesado de los *values*, donde los pesos surgen de la interacción entre Q y K.

Respecto a su implementación, la arquitectura de los *Transformers* se ilustra en la figura 2.2. A continuación, se detalla cada uno de los bloques principales que conforman esta arquitectura:

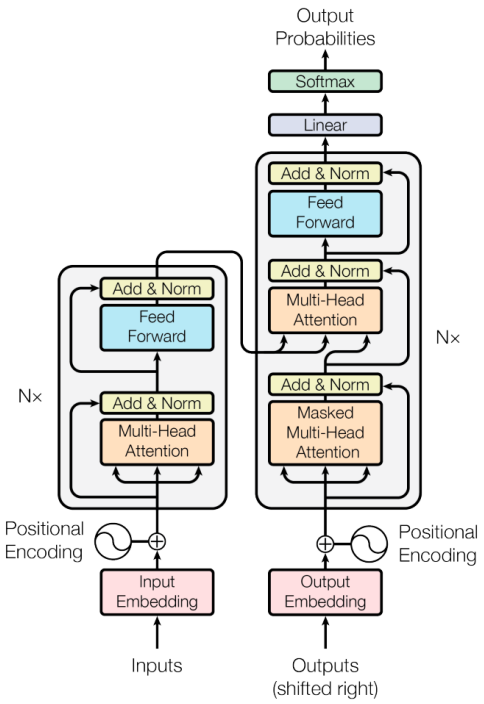


Fig. 2.2: Esquema de la arquitectura *Transfomer* presentada por Vaswani et al. (2017).

- ***Encoder-Decoder***: El *encoder* (bloque de la izquierda) tiene la finalidad de procesar la secuencia de entrada y generar una representación de la misma, que luego el *decoder* (bloque de la derecha) utilizará para generar de manera auto-regresiva la secuencia de salida. Como es posible notar en el esquema, en el segundo bloque de atención del *decoder*, parte de la entrada es la salida proveniente del *encoder*. Más precisamente, K y V provienen de la salida del encoder, mientras que Q proviene de la salida de la capa anterior del *decoder* para el token previo (por el shift-right).

- **Auto-Atención Multi-Cabeza** (*Multi-Head Self-Attention*): Extiende el mecanismo de atención al calcular múltiples representaciones en paralelo, lo que permite al modelo capturar relaciones en diferentes subespacios de la entrada. De manera formal:

$$\begin{aligned} \text{MultiHead}(Q, K, V) &= \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)\, W^O \\ \text{donde head}_i &= \text{Attention}(Q\, W_i^Q, K\, W_i^K, V\, W_i^V) \end{aligned} \tag{2.5}$$

Donde $W_i^Q, W_i^K, W_i^V \in \mathbb{R}^{d_{\text{model}} \times d_h}$ son matrices que aprenden a proyectar la entrada de cada cabeza con $d_h = d_{\text{model}}/h$. Como se ilustra en 2.3, luego de que cada cabeza aplique atención sobre su entrada, las salidas son concatenadas y nuevamente proyectadas linealmente con $W^O \in \mathbb{R}^{d_{\text{model}} \times d_{\text{model}}}$. Cabe destacar, que con este mecanismo la cantidad de parámetros del modelo no varía al modificar la cantidad de cabezas.

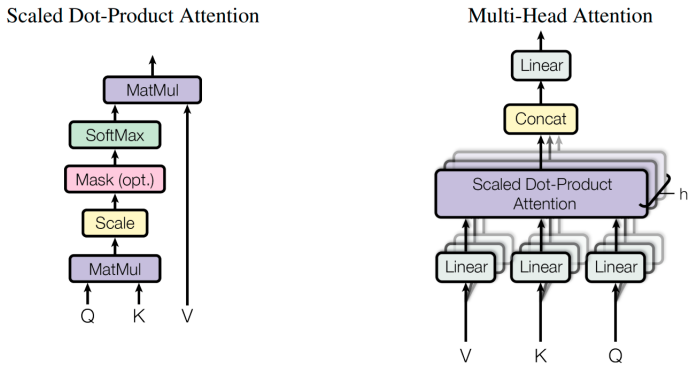


Fig. 2.3: Esquema del funcionamiento de *Multi-Head Self-Attention*.

- ***Positional Encoding*** (PE): Dado que los *Transformers* no tienen un mecanismo intrínseco para capturar el orden de las secuencias, se introduce información posicional explícita a los embeddings de entrada mediante funciones sinusoidales o embeddings aprendidos. En concreto, a la entrada se le suma un valor en cada posición de manera que el modelo tenga la posibilidad de reconocer la posición en la secuencia al reconocer el patrón de esta perturbación. La codificación más clásica sigue la siguiente definición:

$$PE_{(pos,2_i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right), \quad PE_{(pos,2_{i+1})} = \cos\left(\frac{pos}{10000^{\frac{2i}{d}}}\right) \tag{2.6}$$

  Donde *pos* es la posición en la secuencia, i es el índice de la dimensión en el embedding, y d es la dimensión del embedding.

- **Regularización**: Técnicas como normalización por capas (*Layer Normalization*) y *Dropout* son incorporadas para estabilizar el entrenamiento y prevenir el sobreajuste. El mecanismo de *Dropout* coloca ceros de manera aleatoria en elementos de la entrada y ha demostrado ser una técnica efectiva de regularización que evita la co-adaptación de neuronas (cuando distintas unidades tienen un comportamiento altamente correlacionado) (Hinton et al., 2012). *Layer Normalization* aplica una operación de normalización en cada batch con parámetros que son aprendidos durante el entrenamiento según describe Ba et al. (2016). De esta manera, se reduce la variación no informativa en los valores de salida entre capas.

2.2. Representation Learning y Transfer Learning

Las tareas de procesamiento de información (texto, audio, imágenes) pueden ser muy sencillas o muy difíciles dependiendo de cómo se representa a la información. Una forma de ilustrarlo es mediante una analogía: si estamos buscando a nuestros anteojos y tenemos miopía (baja resolución a la distancia) encontrarlos puede ser muy difícil, mientras que con una buena resolución esto es mucho más sencillo. *Representation Learning* es el campo que estudia cómo aprender las mejores representaciones, ¿Y qué es una buena representación? Una primera respuesta indica que es aquella que hace que las tareas subsiguientes sean más fáciles.

En redes neuronales, se puede pensar que las capas ocultas están aprendiendo representaciones. Particularmente en clasificación, la última capa suele ser una capa lineal a la que se le aplica softmax, las capas anteriores aprenden representaciones que hacen que la clasificación sea más sencilla para la última capa. Dicho de otro modo, las capas intermedias llevan la entrada a un espacio más fácil de separar en las clases de interés. Ya décadas atrás, Rumelhart et al. (1986) mencionó cómo con *backpropagation* las capas ocultas representan features relevantes para la tarea de interés.

Una arquitectura que resulta de particular interés en el área son los *Auto-Encoders* (Lecun and Soulie Fogelman, 1987) que en una o más capas llevan la entrada a una dimensión menor (codificación), y luego en una o más capas vuelven a la dimensión original (decodificación). Para el entrenamiento se configura como objetivo que estas redes retornen como salida lo mismo que la entrada. El resultado logrado es que las capas ocultas aprenden representaciones de menor dimensión, alcanzando dimensión mínima en la capa del medio. Nótese que la función de pérdida se define automáticamente sin necesidad de una etiqueta, por lo que pueden ser entrenadas en grandes volúmenes de datos sin etiquetar.

Goodfellow et al. (2016) explica que el renacimiento moderno del aprendizaje profundo se da con el descubrimiento del *Greedy Layer-Wise Unsupervised Pre-training* (Hinton et al., 2006). No entraremos en detalles del algoritmo; nos quedaremos con las bases que cimienta, el **pre-entrenamiento no supervisado**. El entrenamiento es dividido en dos etapas, una primera que entrena a parte de la red neuronal, y otra que ajusta a todo el modelo.

La primera etapa es llamada *pre-training*, en este pre-entrenamiento el objetivo es aprender una tarea auxiliar que hará a la tarea de interés más sencilla. El resultado deseado de esta tarea es llevar la entrada a un espacio en el que separar en las clases deseadas es más sencillo: aprender una representación. En la literatura, es usual encontrar que este espacio vectorial es llamado espacio latente o espacio de *embeddings*, un espacio donde los elementos que se parecen entre sí están ubicados más cerca unos de otros.

Luego de esta primera etapa, los modelos pre-entrenados llevan el nombre de *upstream*. Posteriormente, se incorpora un modelo específico para la tarea de interés, conocido como *downstream*, que se entrena desde cero utilizando como entrada las representaciones generadas por el modelo pre-entrenado.

A continuación, en la segunda etapa, se realiza el *fine-tuning*, donde partiendo de los pesos resultantes de la primera etapa se ajustan algunas capas (comúnmente las últimas) o todas a la vez en conjunto con el modelo downstream. Con este método se han alcanzado mejores resultados que entrenando a todo el modelo en una única etapa de entrenamiento. En algunos casos, también puede haber una etapa intermedia en la que primero se entrena únicamente el modelo *downstream* manteniendo fijo el modelo upstream, antes de proceder al ajuste conjunto en el *fine-tuning*.

Más tarde, ésta metodología llevaría a una revolución ya que abre las puertas a entrenar sobre una enorme cantidad de datos sin necesidad de etiquetar y trayendo grandes beneficios. En *procesamiento del lenguaje natural* (NLP) se avanzó en la representación de palabras (Mikolov et al., 2013; Pennington et al., 2014) y luego en representaciones teniendo en cuenta su contexto (Dai and Le, 2015; Howard and Ruder, 2018; Radford and Narasimhan, 2018). En particular, nos detendremos en BERT (Devlin et al., 2019), que marcó un hito en *pre-training* al lograr resultados estado del arte en tareas muy variadas con esta técnica. Su arquitectura, compuesta únicamente por bloques bidireccionales de *Transformers*, se pre-entrena inicialmente con texto sin etiquetar y posteriormente se

adapta a tareas específicas mediante fine-tuning, proceso en el cual solo se agrega una capa de salida especializada.

En cuanto a los detalles del pre-entrenamiento, el proceso se llevó a cabo mediante dos tareas complementarias:

- Predicción de palabras enmascaradas: se enmascara un porcentaje aleatorio de la entrada y luego se entrena al modelo para que prediga el texto enmascarado. El objetivo de esta tarea es que el modelo aprenda a representar el sentido de las palabras dentro de una oración. Basada en Taylor (1953), la idea detrás de esta tarea también fue adaptada en imágenes mediante oclusiones y en audio enmascarando segmentos.

- Predicción de la siguiente oración: esta tarea ayudaría al modelo a entender la relación entre oraciones.

Todas estas técnicas de aprendizaje no supervisado se engloban dentro de lo que se denomina ***Self-Supervised Learning*** (SSL). Con SSL se pueden aprovechar de manera sencilla grandes volúmenes de datos para llevar a cabo tareas pretexto (*pretext tasks*), como las mencionadas, dando como resultados a estos modelos pre-entrenados. Estos modelos llevan también la definición de *foundation models* (Bommasani et al., 2021) por el cambio de paradigma que representan en el campo de la inteligencia artificial. El término enfatiza que estos modelos, si bien son incompletos por si mismos, sirven como base común sobre la cual se construyen modelos específicos para distintas tareas mediante adaptación.

Por su parte, en audio, los datos disponibles son más escasos y más aún la cantidad de datos etiquetados, cuyas etiquetas son mucho más costosas de generar. Aún así, aunque un tanto rezagado, el procesamiento de audio siguió un camino paralelo con redes convolucionales, como por ejemplo en Wav2Vec (Schneider et al., 2019) y VQ-Wav2Vec (Baevski et al., 2019)). Más tarde el desarrollo de modelos continuó integrando Transformers siguiendo la idea de enmascarado en el pre-entrenamiento. Algunos ejemplos son Wav2Vec 2.0 (Baevski et al., 2020), HuBERT (Hsu et al., 2021) y WavLM (Chen et al., 2021)). En algunos casos se trabaja directamente sobre formas de onda y en otros con representaciones espectrales.

En particular, haremos un breve hincapié en WavLM, que es un modelo que utilizaremos en este trabajo. En cuanto a su arquitectura, podemos observar en la figura 2.4 que consiste fundamentalmente en un encoder convolucional seguido de un encoder Transformer, al igual que Wav2Vec2.0 y HuBERT. A diferencia de estos últimos dos modelos, WavLM incorpora el mecanismo *Gated Relative Position Bias* (Chi et al., 2022) en las capas Transformer, que modifica la ecuación 2.4 de atención agregando un término (*bias*) al que se le aplica *softmax*. Esta técnica, basada en el mecanismo de *Gated Recurrent Unit* (GRU; Cho et al. (2014)), funciona como un *positional embedding* que no solo tiene en cuenta la posición sino también el contenido del frame. En el contexto de WavLM, un frame es el resultado de dividir la señal de audio continua en pequeños fragmentos de igual duración en el encoder convolucional. La intuición otorgada detrás del uso del mecanismo es que el *offset* de las distancias entre dos frames tiene distintos roles dependiendo del contenido de los frames, los autores dan como ejemplo a cuando un frame pertenece a un segmento de silencio y otro a uno de habla.

Por otro lado, en cuanto a las tareas de pre-entrenamiento de WavLM, los autores propusieron un *framework* de eliminación de ruido y predicción de segmentos enmascarados.

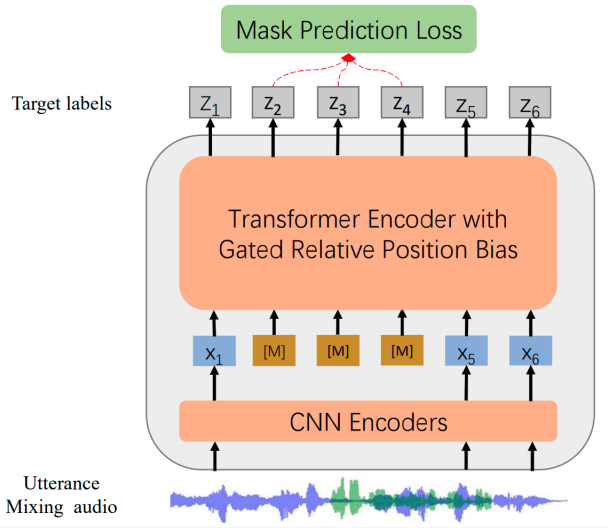


Fig. 2.4: Arquitectura del modelo WavLM (Chen et al., 2021) y diagrama de su pre-entrenamiento, donde "$[M]$"hace referencia a los segmentos enmascarados de la entrada. Al mismo tiempo, las distintas ondas de colores en la entrada hacen referencia al agregado de habla superpuesta y ruido, mientras que los z_i son las pseudo-etiquetas correspondientes a cada frame.

Específicamente, durante el entrenamiento se simularon entradas ruidosas y con superposición de voces; seleccionaron aleatoriamente algunos segmentos de habla y las mezclaron con ruido de fondo o con un segmento del habla secundario. Tanto el audio de ruido como la uterancia secundaria se seleccionan al azar del mismo batch, se recortan aleatoriamente y se escalan según una proporción también aleatoria de la energía de la fuente.

Luego, la tarea consiste en predecir pseudo-etiquetas del audio original en regiones (secuencias de frames) enmascaradas. Estas pseudo-etiquetas se generan utilizando un modelo de *k-means clustering*, que asigna cada frame a una clase basándose en representaciones espectrales (como MFCCs) o representaciones latentes de otros modelos. La función de pérdida utilizada, conocida como *Mask Prediction Loss*, se calcula como la entropía cruzada entre las predicciones del modelo para los frames enmascarados y estas pseudo-etiquetas. Mediante esta tarea de predicción y eliminación de ruido en regiones enmascaradas, el modelo aprende a predecir información faltante en la señal, la cual podría corresponderse con fonemas y características tímbricas y prosódicas de la voz. Para lograr un buen desempeño en esta tarea, el modelo debería aprender a utilizar el contexto, desarrollando una capacidad de identificar fonemas, tono y timbre de voz, entre otras características de la señal. A su vez, al utilizar contextos con ruido durante el pre-entrenamiento se promueve robustez ante entornos acústicos complejos. De esta manera, esta estrategia permite al modelo desarrollar robustez ante entornos acústicos complejos.

Al realizar un fine-tuning con un modelo downstream estamos realizando lo que se denomina ***transfer learning***, cuando un modelo que fue entrenado en un ambiente (tarea y datos) lo usamos para entrenar en otro ambiente distinto. En la presente sección, ya hemos expuesto un ejemplo de esta técnica: la utilización de representaciones aprendidas con SSL para resolver una tarea de clasificación.

Sobre la evaluación de estos modelos, hay diferentes *datasets* y *benchmarks* que evalúan a estos modelos pre-entrenados con downstreams fijos para cada tarea. En habla, como

fue mencionado, uno de los benchmarks más usados es SUPERB (Yang et al., 2021). Puntualmente para SID, uno de los *datasets* más utilizados es VoxCeleb1 (Nagrani et al., 2017), que es utilizado por SUPERB para evaluar en esta tarea. El *baseline* para SID sin pre-entrenamiento es de una accuracy de 80,5 (Nagrani et al., 2017) utilizando una red convolucional de 67M parámetros. En SUPERB, con un downstream de menos de 1M de parámetros se publicaron los siguientes resultados:

Modelo	**Arquitectura**	**#Params**	**Accuracy Top 1 %**
Wav2Vec	19-Conv	32.54M	56.56
VQ-Wav2Vec	20-Conv	34.15M	38.80
Wav2Vec 2.0 Base	7-Conv 12-Trans	95.04M	75.18
Wav2Vec 2.0 Large	7-Conv 24-Trans	317.38M	86.14
HuBERT Base	7-Conv 12-Trans	94.68M	81.42
HuBERT Large	7-Conv 24-Trans	316.61M	90.33
WavLM Base+	7-Conv 12-Trans	94.70M	89.42
WavLM Large	7-Conv 24-Trans	316.62M	**95.49**

Tab. 2.1: Resultados reportados en SUPERB para SID en VoxCeleb1. La cantidad de parámetros corresponde al modelo pre-entrenado mientras que el modelo downstream es el mismo para todos los casos, con menos de 1M de parámetros.

2.3. Técnicas de fine-tuning eficientes en parámetros (PEFT)

Con el continuo crecimiento de la cantidad de parámetros de los modelos pre-entrenados, la adaptación a tareas downstream se ha vuelto un desafío, especialmente en ambientes con limitados recursos computacionales. Más aún, si bien el *fine-tuning* permite alcanzar mejores resultados en datos dentro de la distribución de entrenamiento, esta técnica puede resultar en un peor desempeño cuando se trabaja con datos fuera de dicha distribución (Kumar et al., 2022). Este fenómeno, conocido como *domain-shift*, hace que el *fine-tuning* no sea la técnica más robusta para luego realizar inferencia en escenarios variados. Adicionalmente, al modificar los pesos del upstream, las representaciones se ajustan específicamente a la tarea downstream, lo que requiere almacenar una copia completa del modelo para cada tarea.

Estas limitaciones motivaron el desarrollo de técnicas de fine-tuning eficientes en parámetros, conocidas como *Parameter Efficient Fine-Tuning* (PEFT). Houlsby et al. (2019) propone el uso de módulos adaptadores (*Adapters*), que incorporan una pequeña cantidad de parámetros entrenables dentro del modelo mientras dejan fijos los demás. En concreto, los *Adapters* son pequeñas capas que se insertan entre las capas existentes del modelo pre-entrenado. Típicamente, su arquitectura consiste en una proyección lineal que reduce la dimensión de los embeddings, seguida de una no-linealidad y una proyección que restaura la dimensión original (véase la figura 2.5). Esta configuración permite capturar adaptaciones específicas de la tarea mientras mantiene la mayor parte de los pesos pre-entrenados sin modificar, resultando en una significativa reducción de parámetros entrenables y menor riesgo de sobreajuste. De esta manera, no es necesario almacenar una copia completa del modelo por tarea sino solo una fracción.

Una limitación de este enfoque es que el adapter incrementa el costo temporal de inferencia al incorporar nuevas capas en la secuencia. Sin embargo, el éxito de los adapters

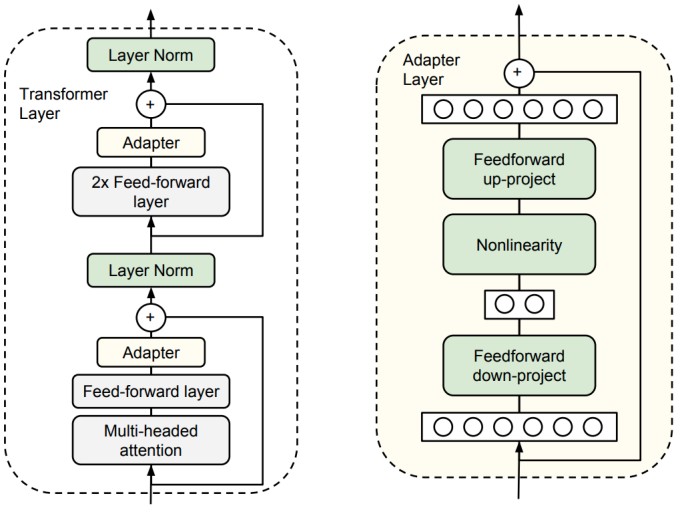


Fig. 2.5: Arquitectura del módulo adapter y su integración con *Transformer*. Izquierda: Se agrega el módulo adapter dos veces a cada capa del Transformer: después de la proyección que sigue a la atención multi-cabeza y después de las dos capas feed-forward. Derecha: El adapter consiste en un cuello de botella con una conexión residual. Durante el entrenamiento del adapter (*adapter tuning*), las capas verdes se entrenan con los datos downstream, esto incluye al adapter, los parámetros de *Layer Norm* y la capa de clasificación final (no mostrada en la figura). Extraído de Houlsby et al. (2019).

impulsó el desarrollo de diferentes alternativas. Entre los métodos PEFT más utilizados se destaca ***LoRA*** (*Low-Rank Adaptation*). En lugar de agregar nuevas capas, LoRA aproxima en paralelo las actualizaciones de los pesos mediante matrices de bajo rango:

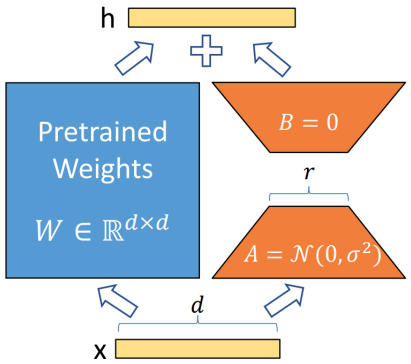


Fig. 2.6: Reparametrización de LoRA, que solo entrena las matrices A y B. r refiere al rango de las matrices y d a la dimensión oculta del modelo upstream. Las igualdades son las inicializaciones de cada matriz. Extraído de Hu et al. (2022).

El rango r que se observa en la figura 2.6 es un hiperparámetro nuevo. Además, LoRA introduce un segundo hiperparámetro α que escala el gradiente en la ecuación de actualización de pesos por $\frac{\alpha}{r}$. A su vez, al implementar matrices en paralelo, los tiempos de inferencia no se ven alterados. Más aún, si bien se incorporan nuevos parámetros con LoRA, estos pueden ser combinados con los pesos originales de las capas y mantener el mismo desempeño en inferencia. De esta manera, LoRA logra reducir significativamente

la cantidad de parámetros entrenables mientras mantiene un desempeño competitivo.

No obstante, persiste una brecha en el desempeño entre LoRA y el fine-tuning completo; Biderman et al. (2024) muestra que esta brecha es consistente en múltiples dominios, sin embargo, LoRA tiene un menor olvido del dominio original con el que fue entrenado el modelo y mantiene una capacidad de generalización más diversa. De esta manera, esta técnica motivó el desarrollo de nuevas variaciones que puedan reducir o superar la diferencia con el fine-tuning completo manteniendo las buenas propiedades.

3. CONFIGURACIÓN EXPERIMENTAL

Tesista, no hay modelo, se hace modelo al experimentar.

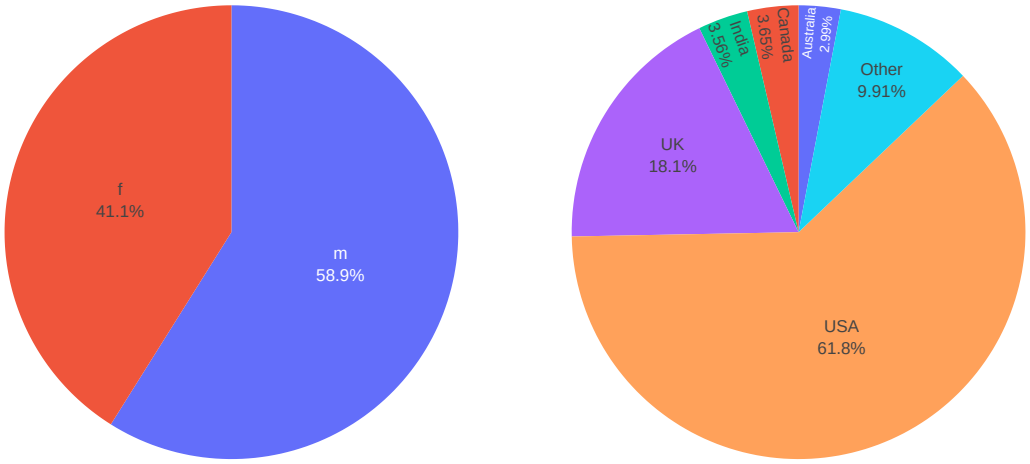
En este capítulo, describimos el diseño experimental adoptado para la búsqueda de hiperparámetros con el fin de encontrar una arquitectura óptima y eficiente para SID, considerando las restricciones de recursos computacionales disponibles. Nuestra metodología divide la exploración en dos fases principales: primero, la optimización del modelo downstream manteniendo congelado al modelo upstream (capítulo 4), y segundo, el entrenamiento conjunto de ambos modelos (capítulo 5).

Cabe notar que el código que sustenta los modelos y diferentes configuraciones creció a la par de los experimentos manteniendo buenas prácticas de desarrollo de software. Este código se encuentra disponible en un repositorio de GitHub[1], con el cual cada experimento puede volver a realizarse ejecutando un comando. Siempre que sea oportuno realizaremos consideraciones en cuanto a las limitaciones de recursos.

3.1. Dataset

Durante los siguientes experimentos vamos a estar utilizando VoxCeleb1 (Nagrani et al., 2017) como dataset, que es uno de lo más utilizados para SID y SV. VoxCeleb1 cuenta con 153516 audios extraídos de videos de YouTube que pertenecen a 1251 celebridades. Cada audio corresponde a un segmento de un video, estos segmentos fueron extraídos de 22168 videos. En cuanto a las estadísticas de los segmentos, la duración promedio es de 8,245 segundos, con un desvío estándar de 5,31, duración mínima de 3,96 y máxima de 144,92 segundos. Al mismo tiempo, cada hablante tiene en promedio 18 videos, como mínimo poseen 6 y como máximo 36. Están disponibles particiones oficiales en conjuntos para entrenamiento, evaluación y validación con 138361, 8251 y 6904 audios respectivamente. Disponemos de información sobre la nacionalidad y género de los hablantes, en la figura 3.1 podemos observar la proporción de cada una. Otra información relevante a la hora de identificar voces podría ser la edad, sin embargo, esta información no es provista en el dataset y debido a que la recopilación fue hecha en diferentes momentos de la vida de cada hablante esta tarea no es sencilla.

[1] https://github.com/erikernst4/speech-hypertuning



(a) Balance por género. *(b)* Distribución de nacionalidades.

Fig. 3.1: Estadísticas de VoxCeleb1.

En cuanto a la evaluación y selección de modelos, se utilizará como criterio principal el desempeño en el conjunto de evaluación, eligiendo aquél que obtenga las mejores métricas. De esta manera, al tomar decisiones sobre este conjunto, es importante señalar que esta metodología limita la comparación directa con otros modelos de la literatura. La causa por la que elegimos esta metodología es que ya sobre el conjunto de validación también se tomarán decisiones. Más precisamente, se determinará cuándo un modelo convergió observando las métricas sobre este conjunto de datos. En consecuencia, dado que se tomarán decisiones sobre el conjunto de validación durante el proceso de entrenamiento, éste no resulta apropiado para la comparación final entre nuestros modelos. De todos modos, hemos observado una correlación alta entre los resultados en validación y evaluación, y no hemos percibido diferencias significativas. En conclusión, una comparación rigurosa con otros trabajos requeriría un conjunto de datos completamente independiente, sobre el cual no se haya tomado ninguna decisión, ni durante el entrenamiento ni durante la evaluación.

3.2. Modelo

En primer lugar, antes de poder emprender la búsqueda del mejor modelo downstream hay que fijar un modelo upstream. En nuestro caso hemos optado por el modelo WavLM Base+, un modelo de 94.4M parámetros entrenado en 94k horas de audio sin etiquetar. La elección se debe a que es uno de los modelos que mejor se desempeñó en SUPERB, en particular en SID como se mostró en la tabla 2.1, con una cantidad de parámetros considerablemente menor a los modelos *large*. La implementación del modelo que hemos seleccionado es la que se encuentra disponible en S3PRL [2], código con los que son evaluados los modelos en SUPERB. Durante la búsqueda del mejor modelo downstream el modelo upstream estará congelado, es decir, no modificará sus pesos.

En segundo lugar, debemos definir un primer bosquejo del downstream para tener un

[2] https://github.com/s3prl/s3prl

modelo completo de clasificación. Antes, debemos tener algunas consideraciones sobre la salida del modelo upstream. Al hacer inferencia con un audio, WavLM Base+ nos devuelve las representaciones de 13 capas: la salida del encoder convolucional y cada una de las capas *Transformer*. A su vez, retorna una representación cada cierta cantidad de tiempo, dando lugar a C secuencias de representaciones con tamaño TxD, en donde C es la cantidad de capas (13 en este caso), T es la cantidad de 'frames', y D es la dimensionalidad de cada representación (768).

En este contexto, el *pooling* es una operación fundamental que nos permite reducir la dimensionalidad de las representaciones mientras preservamos la información más relevante para la tarea. Específicamente, necesitamos aplicar pooling en dos dimensiones: la dimensión temporal y la dimensión de las capas. El pooling temporal es necesario para condensar la información de todos los frames en una única representación que capture las características globales del audio, independientemente de su duración. Por otro lado, el pooling de capas nos permite combinar la información de las diferentes capas del modelo, conservando la información más útil para la tarea en cuestión.

Con estos contenidos en mente, se propone una arquitectura inicial que utilizará al promedio como pooling temporal, seguido de un promedio pesado como pooling de capas y un perceptrón multicapa final. La salida es un vector de tamaño igual a la cantidad de hablantes del problema. Por defecto, en el perceptrón multicapa configuramos en 2 la cantidad de capas y en 128 su dimensión oculta. Gráficamente, la arquitectura sigue la siguiente estructura:

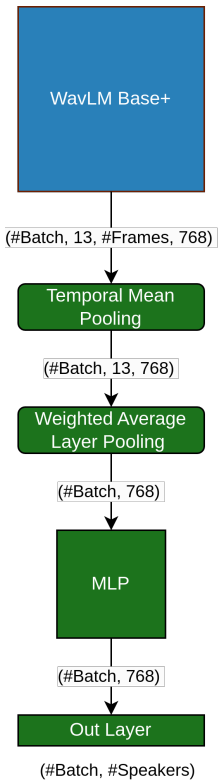


Fig. 3.2: Esquema del primer modelo propuesto. Los colores distinguen el upstream del downstream, mientras que en las flechas se indican las dimensiones de las salidas de cada capa.

Considerando que el factor más determinante en el tiempo de ejecución de una pasada

forward es el modelo upstream, y que no será entrenado en las primeras exploraciones, es deseable poseer la entrada pre-calculada a la parte del modelo que sí entrenaremos. Almacenar embeddings de dimensión (13, #Frames, 768) no es viable, ya que implica una enorme cantidad de memoria. Es por ello que se decidió el orden de los poolings propuesto, ya que el *Temporal Mean Pooling* no posee parámetros entrenables y es determinístico, por lo que el modelo a entrenar comienza con el *Weighted Average Layer Pooling*. De esta manera, se puede pre-calcular la entrada al pooling de capas y propagar las activaciones desde ese punto en el entrenamiento. Durante la experimentación distinguiremos la utilización de embeddings pre-computados y embeddings on-the-fly, que son calculados en el momento haciendo una pasada forward por el upstream.

3.3. Métricas de evaluación

Particularmente, las métricas que optimizamos durante el entrenamiento fueron **entropía cruzada** y **entropía cruzada normalizada** (Ferrer, 2022) en la partición de validación. La utilización de la entropía cruzada normalizada surge de que la cantidad de hablantes afecta directamente el rango de valores posibles de la entropía cruzada, lo cual dificulta la comparación directa entre modelos al variar esta cantidad. La versión normalizada divide a la entropía cruzada por $-\sum_{i=1}^{n} P(i)\log(P(i))$, la entropía de la distribución *a priori*, donde n es el cardinal del conjunto de etiquetas. De esta manera, esta métrica permite comparar el desempeño de diferentes modelos independientemente de la cantidad y balance de clases de la tarea, y obtenemos una métrica interpretable donde valores mayores a 1 indican que el sistema es peor que el *naive*, que en nuestro caso sería asignar una probabilidad proporcional a la cantidad de instancias de cada hablante.

A su vez, al evaluar también reportamos *accuracy* top-1 y accuracy top-5. La accuracy top-1 es la métrica clásica que refiere a la proporción de predicciones donde la etiqueta estimada coincide con la etiqueta verdadera. En nuestro contexto, esta métrica indica el porcentaje de segmentos de audio en los que el modelo asigna la probabilidad más alta al hablante correcto. Por su parte, la accuracy top-5 refiere a la proporción de predicciones donde la etiqueta verdadera se encuentra entre las cinco etiquetas con mayor probabilidad asignada por el modelo. Esta métrica resulta especialmente relevante en tareas de reconocimiento de hablantes con un elevado número de clases, donde interesa evaluar si el modelo al menos considera al hablante correcto entre sus predicciones más probables.

3.4. Consideraciones adicionales

A lo largo de la experimentación utilizamos *Adam* (Kingma and Ba, 2014) como optimizador, una de las variantes más usadas de descenso por el gradiente estocástico. A su vez, para agilizar la ejecución de experimentos recurrimos al mecanismo de *Early Stopping*, que consiste en detener el entrenamiento cuando cierta métrica monitoreada (en nuestro caso, entropía cruzada) no mejora después de una cantidad determinada de chequeos. A su vez, es posible definir un umbral mínimo a ser superado para considerar que ha habido una mejora. Por otro lado, la frecuencia con la que se monitoreó la entropía cruzada varió según experimento y tamaño del mismo, así como también la paciencia (pasos tolerados sin mejora) y el umbral mínimo de mejora.

3.5. Hardware

Sobre el hardware utilizado, la búsqueda del mejor modelo downstream en el cápitulo 4 fue realizada en una computadora con procesador Intel Core i7-12700 y placa gráfica NVIDIA GeForce RTX 3060. En esta configuración, disponemos de 32GB de memoria RAM y 12GB de memoria en la GPU. Más tarde, los experimentos en los que se entrena al upstream en el capítulo 5 se llevaron a cabo en otra computadora con procesador AMD Ryzen 9 7950X 16-Core Processor y dos GPUs NVIDIA RTX A5000, así disponiendo de 24GB por GPU y 128GB de memoria RAM.

4. EXPLORACIÓN DEL MODELO DOWNSTREAM

La búsqueda del mejor modelo downstream sigue una estrategia incremental. En este capítulo comenzamos explorando los hiperparámetros fundamentales en un contexto controlado y reducido, utilizando una arquitectura downstream simple y una porción limitada de los datos. Esta aproximación nos permite establecer una base y obtener intuiciones preliminares sobre el comportamiento del modelo. Posteriormente, expandimos nuestros experimentos al conjunto completo de datos, investigando aspectos más sofisticados como diferentes métodos de pooling, normalización de embeddings y la interacción entre distintas métricas de rendimiento.

En la experimentación realizada en esta tesis no se compilan resultados favorables de experimentos óptimos, los más valiosos aprendizajes fueron adquiridos en la experimentación misma. A lo largo de los experimentos comentaremos propuestas que no funcionaron, y mejoras en el diseño de experimentos que fueron incorporadas con posterioridad.

En búsqueda de una arquitectura y configuración óptima, como primer acercamiento, comenzaremos explorando los hiperparámetros más esenciales como tasas de aprendizaje, capas ocultas del modelo downstream y tamaño oculto. Para poder realizar muchos experimentos rápidamente, primero utilizaremos una porción pequeña de los datos: 100 hablantes (50 mujeres y 50 hombres) y 25 audios por hablante (19 para entrenamiento, 3 para validación y 3 para evaluación), donde cada audio corresponde a un video distinto. Para todos estos datos precalcularemos los embeddings del upstream de manera que no necesitemos hacer inferencia sobre el upstream durante el entrenamiento.

4.1. Tasas de aprendizaje

La tasa de aprendizaje (LR, por sus siglas en inglés) es uno de los factores más influyentes en el entrenamiento y desempeño de modelos (Wu et al., 2019a). Como primer experimento, exploramos la utilización de distintas tasas fijas de aprendizaje. El tamaño de batch fue de 64 para entrenamiento. Se realizó una validación por época, en este caso, esto ocurre aproximadamente cada 30 steps. Se configuró la paciencia del early stopping en 20 y un umbral mínimo de mejora en $0,001$.

Las primeras tasas con las que experimentamos fueron las predeterminadas y estándar en la mayoría de los casos: $0,001$, $0,0001$, $0,00001$ y $0,000001$.

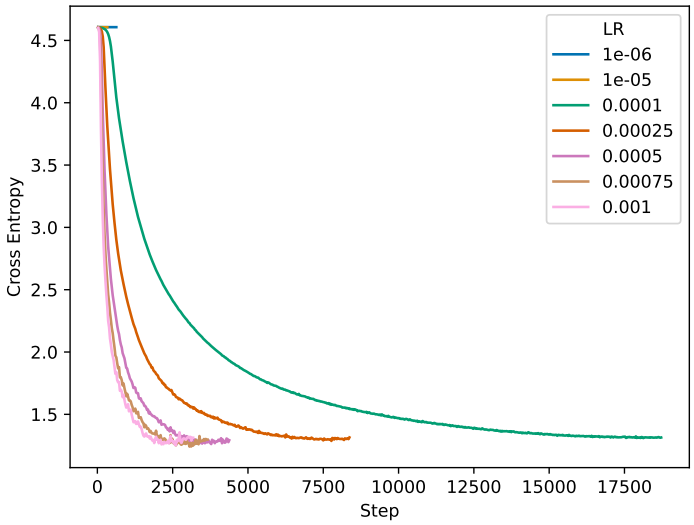


Fig. 4.1: Evolución del cross entropy en validación durante el entrenamiento al experimentar con diferentes LRs.

Como era esperable, observamos que LR más chicos llevan a converger más tardíamente. Sin embargo, si bien la intuición es que a menor LR mayor granularidad para recorrer el espacio de gradientes, esto no garantiza converger a un valor menor en la función de pérdida. A su vez, se observa en la figura 4.1 una pequeña curva azul que ilustra que con 10^{-6} no alcanzó a haber una mejora respecto al modelo inicial y se detuvo el entrenamiento al alcanzar la paciencia del *Early Stopping*.

En la siguiente tabla se describe el desempeño de cada modelo entrenado en la partición de evaluación:

LR	Cross Entropy	Accuracy Top 1	Accuracy Top 5
10^{-3}	1.3620	0.6500	0.9367
$7{,}5 * 10^{-4}$	1.3532	0.6433	0.9333
$5 * 10^{-4}$	**1.2622**	0.6367	**0.9400**
$2{,}5 * 10^{-4}$	1.2825	**0.6567**	0.9233
10^{-4}	1.2689	0.6333	0.9267
10^{-5}	2.5214	0.3400	0.7267
10^{-6}	4.6064	0.0100	0.0500

Tab. 4.1: Resultados en la partición de test de la experimentación con LR fijos. En color se distinguen los modelos que provienen de un refinamiento posterior.

Como muestra la tabla 4.1, con LR 10^{-3} se obtuvo mayor accuracy mientras que con LR 10^{-4} se obtuvo menor cross entropy entre los primeros valores explorados. Luego, se decidió explorar valores intermedios entre 10^{-3} y 10^{-4} buscando mejores resultados, y así experimentamos también con $2{,}5 * 10^{-4}$, $5 * 10^{-4}$ y $7{,}5 * 10^{-4}$. Esto llevó a una cross entropy mínima en la partición de test con LR $5 * 10^{-4}$, por lo que mantendremos este hiperparámetro con ese valor en los experimentos siguientes.

4.2. Capas ocultas

Otro hiperparámetro fundamental en la construcción del modelo downstream es la cantidad de capas de la MLP final. En tareas de clasificación es común el uso de una única capa final donde la entrada tiene dimensión igual a la dimensión de la representación de la última capa del modelo upstream y la salida tiene dimensión igual a la cantidad de clases. En el modelo propuesto (véase la figura 3.2), antes de esta capa final se propuso el uso de una MLP con 2 capas y tamaño oculto 128. En este experimento exploraremos la cantidad de capas con 128 de tamaño oculto.

Capas Ocultas	Accuracy Top1	Accuracy Top5	Cross Entropy
0	0.6333	0.8867	1.3254
1	**0.7033**	**0.9400**	**0.9998**
2	0.6367	**0.9400**	1.2622
3	0.4700	0.8900	1.6514

Tab. 4.2: Métricas en la partición de evaluación al experimentar sobre el número de las capas de la MLP.

El caso de 0 capas refiere a la no utilización de la MLP, es decir, que la salida del pooling de capas vaya directamente a la capa de salida. Como se puede observar en la tabla 4.2 obtuvimos los mejores resultados en la partición de evaluación con 1 capa oculta. Cabe notar que es posible que la cantidad óptima de capas pueda ser mayor con una mayor cantidad de datos. No obstante, revisitar todas las decisiones tomadas en el camino haría impracticable a la experimentación, por lo que continuaremos utilizando 1 capa oculta para próximos experimentos.

4.3. Tamaño oculto

El tercer hiperparámetro fundamental en la configuración de una MLP es el tamaño oculto, la dimensión de las capas ocultas. Así como la cantidad de capas determina el tamaño a lo alto de la MLP, la dimensión oculta determina el tamaño a lo ancho. Con este parámetro es habitual el uso de potencias de 2, por lo que decidimos experimentar con las potencias de 2 desde 64 a 4096.

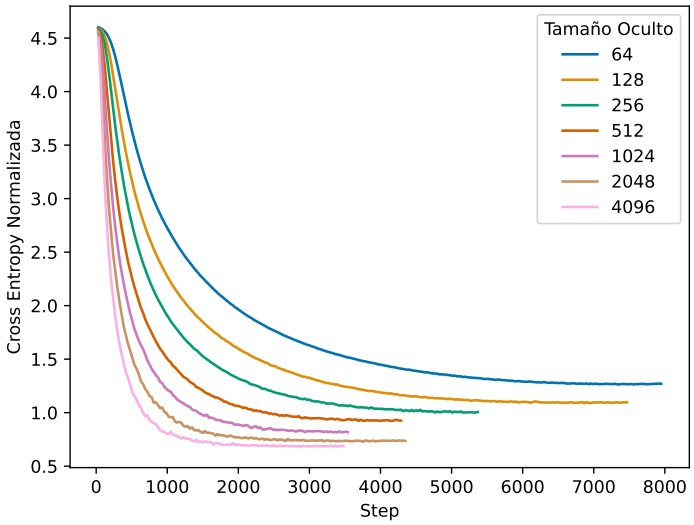


Fig. 4.2: Evolución de la cross entropy en la partición de validación durante el entrenamiento al experimentar con diferentes tamaños ocultos en la MLP.

Tamaño Oculto	**Accuracy Top1**	**Accuracy Top5**	**Cross Entropy**
64	0.6667	0.9033	1.2010
128	0.7033	0.9400	0.9998
256	0.7267	0.9433	0.9104
512	0.7633	0.9500	0.8432
1024	0.7733	0.9600	0.7724
2048	0.7833	0.9500	0.6829
4096	**0.7933**	**0.9633**	**0.6218**

Tab. 4.3: Métricas en la partición de evaluación al experimentar sobre el tamaño oculto en la MLP.

Se observa que a mayor tamaño oculto mayor es la accuracy y menor la entropía cruzada. Sin embargo, al aumentar el tamaño oculto, aumenta la complejidad del modelo y su cantidad de parámetros. La cantidad de parámetros entrenados del modelo downstream está dada por los parámetros de los poolings, la MLP y la capa de salida. El *Temporal Mean Pooling* no posee parámetros entrenables y el *Weighted Average Layer Pooling* tiene 13 parámetros entrenables. La MLP en este punto es una única capa y la cantidad de parámetros es el producto de la dimensión de la representación de las capas del modelo upstream (768) y el tamaño oculto. Los parámetros de la capa de salida son el producto del tamaño oculto de la MLP y la cantidad de hablantes. Concretamente, con tamaño oculto 4096 se tienen $768 * 4096 + 4096 * 100 = 3{,}555{,}328 \approx 3,5M$. En general, se espera que el tamaño del modelo downstream sea significativamente menor al upstream, siendo uno de los grandes beneficios de utilizar un modelo pre-entrenado. No debe perderse de vista que el tamaño de este downstream escala con la cantidad de hablantes. Usaremos 4096 como tamaño oculto en los próximos experimentos, que ha dado los mejores resultados y mantiene el modelo downstream con un tamaño razonable.

Por otro lado, en la Figura 4.2 se puede observar que a medida que crece la complejidad del modelo en general decrece la cantidad de steps necesarios para converger. En resumen,

modelos con mayor tamaño oculto en la MLP llegan antes a mejores resultados.

4.4. Relación entre el tamaño del batch y la tasa de aprendizaje

Hasta este punto, utilizamos 64 como tamaño de batch (BS, por sus siglas en inglés), un valor arbitrario. Sin embargo, con el modelo y la porción del dataset que estamos usando, podemos usar batches de mayor tamaño. En la literatura (Godbole et al., 2023), se suele recomendar el uso de batch con el tamaño más grande posible aunque no son acompañadas de demostraciones o fundamentos convincentes. Cuando los modelos entran en memoria, el siguiente factor más limitante para poder entrenar un modelo es el tamaño de batch. Por lo tanto, consideramos de importancia entender su influencia en el entrenamiento.

Por otro lado, la elección de un tamaño de batch guarda relación con el LR. Intuitivamente, si el BS es chico, no es representativo de la totalidad del dataset, por lo que un LR alto haría que se sobreajuste a cada batch sin poder llegar a aprender de todo el dataset simultáneamente. En el lado opuesto, batches muy grandes tardarán mucho en aprender con LR muy pequeños. Es por esto que nos propusimos explorar esta interacción para seleccionar los valores más adecuados. Barreremos valores de LR de 0,000001 a 1, y BS de 1 a toda la partición de entrenamiento ('*all*', 1900 segmentos de audio). A continuación, puede verse una visualización de la cross entropy en la partición de evaluación para cada una de estas configuraciones.

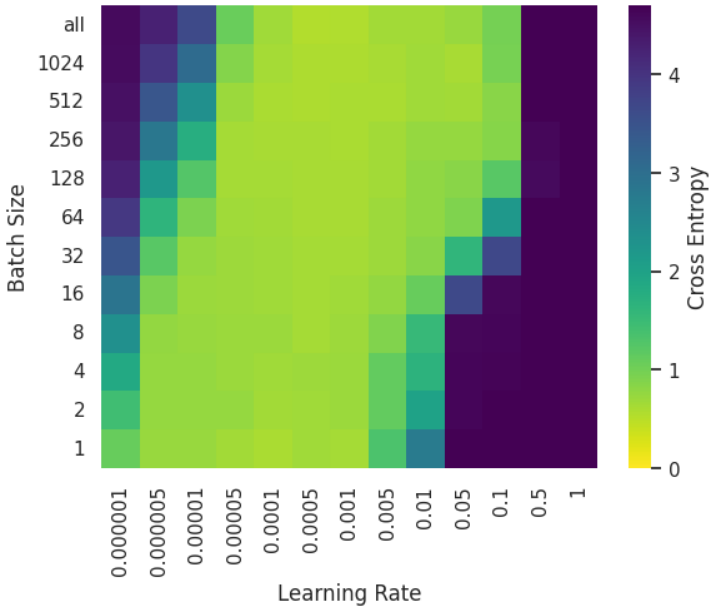


Fig. 4.3: Interacción entre el LR y BS al medir CE en la partición de evaluación.

Los valores más oscuros de la figura hacen referencia a configuraciones que no convergieron, en estos casos el *Early Stopping* detuvo los entrenamientos al no encontrar una mejora.

En cuanto a la selección de valores óptimos de BS y LR, considerando los experimentos que vendrán a continuación, en vez de elegir el punto mínimo (0,5448 de cross entropy en test con $BS = 1900$ y $LR = 0{,}0005$) elegimos buscar los valores más *estables*. Determinamos que los más estables son aquellos cercanos en promedio al mínimo cross entropy y con

menor desviación estándar, que serían los más consistentes en configuraciones distintas. Ambas estadísticas surgen de los resultados obtenidos con cada configuración dejando fijo un hiperparámetro y variando el otro, los valores de toda una fila o una columna en la Figura 4.3 respectivamente.

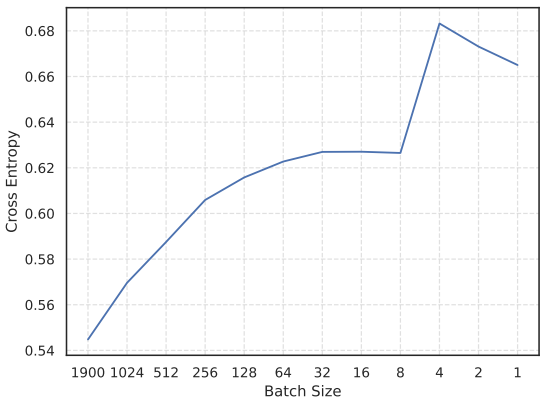
Learning Rate	Mean	Std
0.000001	3.2823	1.3048
0.000005	1.9571	1.3487
0.00001	1.4510	1.0361
0.00005	0.7311	0.1239
0.0001	0.6457	**0.0314**
0.0005	**0.6207**	0.0410
0.001	0.6312	0.0492
0.005	0.8168	0.2506
0.01	1.1848	0.6729
0.05	2.3626	1.8757
0.1	2.8411	1.7941
0.5	5.2534	0.8493
1	469.3508	1600.9756

Tab. 4.4: Estadísticas de Cross Entropy para diferentes Learning Rates

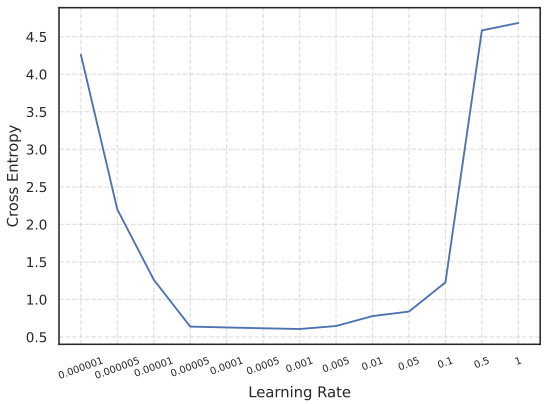
Batch Size	Mean	Std
all	429.1469	1539.5639
1024	2.4032	2.5779
512	2.8707	4.1958
256	2.3934	3.2403
128	**1.7666**	**1.6262**
64	1.8139	1.6908
32	1.8783	1.6527
16	2.1156	1.8646
8	2.2099	1.9562
4	2.1940	1.9235
2	2.2305	2.0086
1	2.3265	2.1465

Tab. 4.5: Estadísticas de Cross Entropy para diferentes tamaños de Batch.

De esta manera optamos 0,0005 como LR y 128 como BS. Esta combinación obtuvo 0,6158 de cross entropy en la partición de evaluación, 0,7967 de accuracy Top1 y 0,9633 de accuracy Top5. Si bien buscamos la elección de los valores más estables, es natural que en algún momento futuro de la experimentación se deba modificar alguna de estas elecciones por diferentes restricciones de tiempo o espacio. Para entender cómo esas modificaciones podrían influir, los siguientes gráficos (4.4) muestran cómo se modifica el desempeño al fijar uno de estos parámetros y alterar el otro:



(a) Con $LR = 0{,}0005$.



(b) Con $BS = 128$.

Fig. 4.4: Cross entropy en la partición de evaluación fijando uno de los hiperparámetros seleccionados.

De la anterior figura se desprende que con $BS = 128$ modificar el LR alrededor de 0,0005 no implica grandes cambios de desempeño, ya que se encuentra amesetado en ese punto. En cuanto al LR, en caso de tener que reducir el BS de 128 hasta 8 lleva a una degradación de la entropía cruzada menor al 0,01, lo cual no sería una alteración significativa al momento de seleccionar a los hiperparámetros óptimos.

4.5. Experimentando con el dataset completo

A partir de este punto, los siguientes experimentos realizados son utilizando la totalidad del dataset y con las particiones oficiales de VoxCeleb1.

En primer lugar, al momento de precalcular el resto de los embeddings del dataset nos encontramos con que los audios pueden ser muy largos, resultando en problemas de limitaciones de la memoria. Este problema existe tanto para precalcular los embeddings como para entrenar calculando embeddings on-the-fly directamente. Un factor determinante para los límites de memoria en este punto es el tamaño del batch. La diferencia es que al precalcular embeddings podría incluso usarse $BS = 1$ ya que solo se hará una vez, mientras que ese tamaño de batch no es viable para entrenar como se vio en el experimento 4.4. Para precalcular embeddings limitamos el tamaño de los audios a 70 segundos, mientras que para entrenar calculando embeddings on-the-fly utilizamos porciones (*chunks*) de 5 segundos tomadas aleatoriamente cada vez. Estos valores son razonables teniendo en cuenta las estadísticas de las duraciones de los audios en el dataset (presentadas en 3.1) y que la tarea es clasificación de hablantes, donde el hablante a identificar estará presente en la totalidad del audio, con excepción de los silencios. De esta forma, al calcular los embeddings on-the-fly nos vimos también limitados a usar $BS = 32$.

Así, se realizó un primer entrenamiento con el modelo que mejores resultados había dado hasta el momento. De los anteriores experimentos se desprende que el modelo con 4096 de tamaño oculto en la MLP, tiene el mejor desempeño en cuanto a cross entropy y convergencia. En este momento, fue cuando verificamos la observación hecha en el experimento 4.3 en cuanto al tamaño del downstream. Llevando el tamaño oculto a 4096 y teniendo 1251 hablantes en el dataset, la cantidad de parámetros del downstream alcanzó los $8{,}3M$. Por lo tanto, antes de continuar, probamos usando un tamaño oculto menor como muestra la Tabla 4.6. Incorporamos también el tamaño oculto 32768 que, en el otro extremo, posee una cantidad de parámetros en el orden del tamaño del upstream.

		Métricas en Evaluación				**Métricas en Entrenamiento**				
Tamaño	**#Params**	**Acc Top1**	**Acc Top5**	**CE**	**NCE**	**Acc Top1**	**Acc Top5**	**CE**	**NCE**	**Steps**
1024	2.1M	0.8571	0.9524	**0.6210**	**0.0890**	0.9871	0.9992	0.0488	0.0070	66000
4096	8.3M	**0.8614**	**0.9546**	0.6295	0.0903	0.9946	**0.9998**	**0.0196**	**0.0028**	**52000**
32768	66M	0.8538	0.9530	0.6480	0.0929	**0.9956**	0.9997	0.0197	0.0028	54000

Tab. 4.6: Resultados para diferentes tamaños ocultos en la MLP del downstream utilizando las particiones oficiales de VoxCeleb1, mostrando métricas tanto en evaluación como en entrenamiento. NCE: Normalized Cross Entropy.

En primer lugar, se puede observar que con tamaño oculto 1024 se pudo obtener resultados levemente mejores en entropía cruzada en los datos de evaluación. Al analizar los resultados en la partición de entrenamiento vemos que el modelo con tamaño oculto 4096 es quien alcanzó mejor desempeño en entropía cruzada, lo que es un indicador de un mayor *over-fitting* dado que el rendimiento de este modelo en los datos de evaluación es

peor que para el modelo más chico. A pesar de que con mayor tamaño oculto se obtuvo una más rápida convergencia, ésta está en el mismo orden de magnitud, mientras que la cantidad de parámetros entrenados es significativamente menor con tamaño oculto 1024, haciéndolo un downstream más razonable. Por su parte, el uso de 32768 como tamaño oculto no produjo resultados favorables y es el que muestra el mayor grado de *over-fitting*, alcanzando peor convergencia y desempeño que con 4096. En este momento, también aprovechamos para examinar el uso de 0 capas ocultas, el cual corresponde a usar un modelo lineal como downstream. Sin embargo el modelo convergía muy lentamente por lo que debimos interrumpir el entrenamiento luego de más de 200.000 steps sin llegar a buenos resultados. Por lo tanto, si bien 4096 tiene mejor accuracy en evaluación, se determinó continuar la experimentación usando tamaño oculto 1024 en la MLP por su menor tamaño y menor entropía cruzada.

4.6. Efecto del "modo evaluación"

El siguiente experimento surgió accidentalmente al agregar tests de regresión. Estos últimos fallaron al ser creados y la causa radicaba en que las salidas del modelo no eran siempre las mismas, y el origen no estaba en el downstream sino en el upstream. Los embeddings del upstream no eran los mismos aún utilizando el modo determinístico: requería que se use el modelo en modo evaluación. Este modo sirve para que capas o partes del modelo que deben comportarse distinto durante inferencia que durante entrenamiento se comporten en modo inferencia. Ejemplos de esto son las capas de *Dropout* (que durante el entrenamiento coloca aleatoriamente ceros en parte de la entrada), *LayerDrop* (que descarta capas enteras), o *Batch Norm* (que aprende a normalizar a cada batch).

Por lo tanto, la utilización de los embeddings precalculados del upstream sin este cuidado no sería del todo correcta. En la Tabla 4.7 se puede observar la diferencia en el desempeño entre usar los embeddings precalculados con y sin el modo evaluación, en ambos casos con $BS = 32$.

Modo evaluación	Acc Top1	Acc Top5	CE	Normalized CE	Steps
✗	0.8571	0.9525	0.6210	0.0890	**66000**
✓	**0.8649**	**0.9562**	**0.6128**	**0.0879**	78000

Tab. 4.7: Métricas en test comparando la utilización del modo evaluación en el modelo upstream.

Las diferencias son favorables al uso del modo evaluación. Cabe recordar, que calculando embeddings on-the-fly el modelo upstream recibe segmentos aleatorios de 5 segundos mientras que los embeddings precalculados corresponden a segmentos de hasta 70 segundos, lo que podría favorecerlo. Además, si bien la cantidad de steps del modelo sin el modo evaluación fue menor hay una gran diferencia en los tiempos de entrenamiento. Al utilizar embeddings on-the-fly la velocidad de entrenamiento en promedio es de $2{,}20 it/s$, mientras que con embeddings precalculados es de $31{,}55 it/s$, más de diez veces más rápido.

4.7. Normalización de los embeddings del modelo upstream

En la literatura, es usual encontrar que se refieren a la inferencia del upstream como extracción de *features*, así como también el uso intercambiado de espacio de embeddings y

espacio de features. En esa misma línea, puede pensarse al modelo upstream como la fuente de información de la que aprenderá el modelo downstream. Más aún, siguiendo las ideas de aprendizaje automático clásico, es razonable pensar en la posibilidad de normalizar estos datos de manera que los features puedan tener todos un tratamiento uniforme.

En este experimento exploraremos el uso de tres métodos para normalizar:

- **Length Normalization** (Garcia-Romero and Espy-Wilson, 2011): Normalizar los embeddings del upstream usando la norma 2 vectorial, con la norma 2 del propio embedding de cada audio.

- **Normalización Z Global**: Calcular la media y desviación estándar de los embeddings de toda la partición de entrenamiento de todas las capas. Luego a cada embedding se le restará esa media y se lo dividirá por la desviación estándar.

- **Normalización Z por capa**: Igual a la normalización Z Global, pero tomando la media y desviación estándar de cada capa en vez de todos los embeddings de todas las capas. Este método entendería a cada capa del upstream como una fuente de información distinta.

La utilización de normalización Z requiere agregar un paso previo al entrenamiento donde se calculan los correspondientes valores de medias y desvíos estándar de los embeddings del upstream.

Debido a que la utilización de los embeddings pre-calculados reduce el uso de memoria, esto nos permite aumentar el BS a 128, que era nuestro valor elegido en el experimento 4.4. Para mantener la frecuencia de validaciones y checkpoints por cantidad de actualizaciones del gradiente redujimos acordemente este intervalo de 2000 a 500. Asimismo, para tener un resultado comparable volvimos a correr el modelo sin normalización con estos cambios.

En el siguiente gráfico podemos observar las curvas de la cross entropy normalizada en la partición de validación durante el entrenamiento.

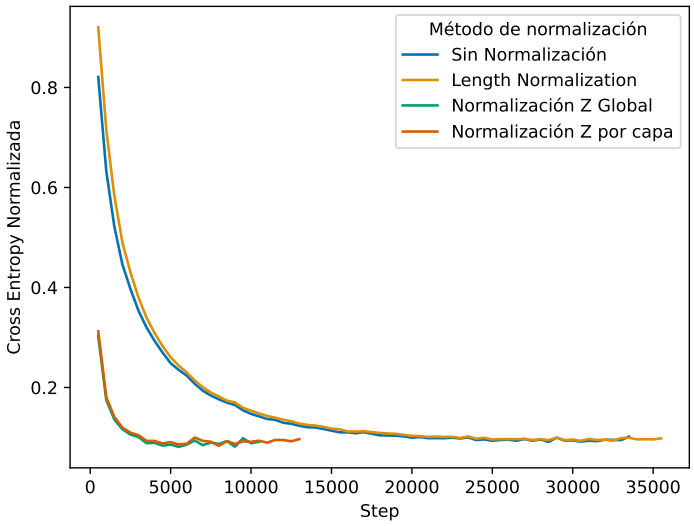


Fig. 4.5: Evolución de la Cross Entropy Normalizada en validación para los diferentes métodos de normalización.

Se puede observar que los modelos con normalización Z convergen significativamente más temprano y a valores menores de cross entropy. Por su parte, Length Normalization no produce una mejora sino un leve deterioro, lo que podría indicar que la utilización de este método significa una leve pérdida de información. Con este último método las diferencias en las magnitudes entre embeddings de distintos audios se pierde, si bien el espacio de embeddings se vuelve más compacto, el modelo parece beneficiarse de la riqueza de las diferentes magnitudes de los embeddings de WavLM.

Al observar que los modelos con normalización Z convergían tanto más temprano pero tenían valores menores en accuracy, probamos también reduciendo el LR de $5 * 10^{-4}$ a 10^{-4}. Los resultados de estos modelos tuvieron una convergencia similar al modelo sin normalización en la cantidad de steps pero no lo superaron en accuracy ni tuvieron menor cross entropy. En la siguiente tabla se exponen los resultados de estos modelos en la partición de test.

Método de normalización	Acc Top1	Acc Top5	CE	Normalized CE	Steps
Sin Normalización	**0.8560**	**0.9553**	0.6097	0.0874	28500
Length Normalization	0.8509	0.9524	0.6328	0.0907	30500
Normalización Z Global	0.8418	0.9519	**0.5647**	**0.0810**	**5500**
Normalización Z por capa	0.8477	0.9529	0.5757	0.0826	8000
Normalización Z Global con LR 10^{-4}	0.8520	0.9549	0.56701	0.0813	27500
Normalización Z por capa con LR 10^{-4}	0.8515	0.9538	0.57812	0.08290	27500

Tab. 4.8: Métricas en la partición de evaluación comparando la utilización de distintos métodos de normalización de los embeddings del modelo upstream.

En conclusión, notamos que la normalización Z por capa tiene una pequeña ventaja en accuracy sobre la normalización Z global, pero no así en cross entropy y convergencia. Por lo tanto, en adelante utilizaremos la técnica de normalización Z global, que proporcionó la menor cross entropy y la convergencia más rápida.

4.8. Métodos de pooling

En esta sección, exploraremos las capas de poolings, buscando entender las ventajas y desventajas de cada método.

4.8.1. Pooling en el tiempo

Hasta este punto, para realizar pooling sobre la dimensión temporal el modelo que hemos construído utiliza *Temporal Mean Pooling*. Si bien es una práctica usual, en esta sección exploraremos métodos alternativos que puedan ser superadores o que ofrezcan alguna ventaja comparativa.

Poolings fijos

En primer lugar, experimentaremos con diferentes poolings fijos. Llamamos poolings fijos a aquellos que no tienen parámetros entrenables y son determinísticos. La gran ventaja de estos poolings, como fue observado en el experimento sobre el modo de evaluación en el modelo upstream (4.6), es que pueden ser precalculados una única vez y luego utilizados durante el entrenamiento ahorrando memoria y tiempo. Usaremos estadísticas típicas sobre los valores de cada dimensión del embedding del upstream: mínimo (min), máximo (max),

desviación estándar (std), y combinaciones entre ellos concatenándolos. Cabe notar que al concatenar estas estadísticas la dimensión del input a la MLP se multiplica, por lo que aumenta la cantidad de parámetros entrenables del downstream.

Pooling	#Params	Acc Top1	Acc Top5	CE	Normalized CE	Steps
mean	2.1M	**0.8418**	**0.9519**	**0.5647**	**0.0810**	5500
std	2.1M	0.4139	0.6433	1.7533	0.2514	8000
min	2.1M	0.4530	0.6947	2.0504	0.2940	8000
max	2.1M	0.4492	0.6873	2.0757	0.2977	5500
min+max	2.9 M	0.5302	0.7570	1.4874	0.2133	5500
mean+std	2.9 M	0.7685	0.9102	0.6985	0.1002	4500
mean+std+min+max	4.4M	0.7332	0.8950	0.7781	0.1116	**3500**

Tab. 4.9: Comparación de distintos poolings fijos en la dimensión temporal. La cantidad de parámetros hace referencia a los parámetros del modelo downstream.

De los resultados se desprende que la media es el pooling temporal fijo que mejor se desempeña. En cuanto a la convergencia, se repite el resultado de que modelos más complejos convergen más rápido. Cabe destacar que *mean+std* es usado usualmente en modelos de verificación de hablante (Snyder et al., 2018), del cual no se observan mejoras salvo por la mejor convergencia. Puede objetarse que el LR es muy alto y con un menor LR los resultados podrían ser distintos, por lo que, también aprovechando la rápida convergencia de estos modelos, experimentamos con $LR = 10^{-4}$. Aún así, no se observaron mejoras. En casi todos los casos hay un leve empeoramiento, salvo pequeñas mejorías, ninguna superando el rendimiento de mean con $LR = 5 * 10^{-4}$.

Poolings con mecanismos de atención

Self-Attention El uso del promedio como pooling temporal tiene el efecto de dar un mismo peso a todos los frames, esto hace que frames que pueden contener silencio tengan la misma importancia que frames en los que la persona habla en su totalidad. En Mirsamadi et al. (2017) se muestra que con un mecanismo de atención local es posible sobreponerse a este fenómeno y otorgarle más peso a los frames con habla. En los siguientes experimentos, exploraremos el uso de *Self-Attention*, particularmente con la implementación de PyTorch de Vaswani et al. (2017). El uso que le daremos como pooling consiste en propagar las activaciones por esta capa y luego promediar en la dimensión temporal. Al haber pasado por el mecanismo de atención esperamos que los valores de los embeddings estén ponderados según su importancia para la tarea.

Es importante destacar que hasta aquí el pooling temporal no tenía parámetros entrenables, en cambio, la utilización de *Self-Attention* como mecanismo de pooling temporal implica $2{,}4M$ parámetros adicionales a entrenar. A su vez, tener parámetros entrenables implica tener que nuevamente calcular los embeddings on-the-fly en vez de precalcularlos.

De esta manera, ejecutamos un primer experimento con este pooling con una única cabeza de atención, obteniendo resultados peores que al usar *Temporal Mean Pooling*. Esto no era lo esperado ya que con atención se podría aprender a hacer un promedio también. Este comportamiento puede explicarse desde la perspectiva del *bias-variance tradeoff*: mientras que el promedio temporal es un mecanismo simple con alto sesgo pero baja varianza, el self-attention es más flexible y puede aprender patrones más complejos, lo que implica menor sesgo pero mayor varianza. Esta mayor capacidad expresiva puede llevar a

que el modelo prefiera ajustarse a patrones específicos del conjunto de entrenamiento en lugar de aprender la solución más simple (el promedio), resultando en overfitting.

Para abordar este desafío, exploramos dos aspectos del entrenamiento que podrían ayudar a controlar mejor el balance entre sesgo y varianza. En primer lugar, consideramos que el LR podría ser demasiado grande: al entrenar más parámetros, es posible que sean beneficiosos cambios más pequeños en cada paso para evitar que el modelo se ajuste demasiado rápido a patrones específicos (en este punto también podría ser útil experimentar con una etapa de *warm-up*, como será hecho luego en la sección 5.1). En segundo lugar, analizamos si la capa oculta podría estar actuando como un cuello de botella al proyectar los embeddings, limitando la capacidad del modelo para aprender representaciones útiles. Por lo tanto, continuamos explorando este mecanismo de pooling con diferentes configuraciones de LR y capas ocultas de la MLP. En la siguiente tabla se observan los resultados de cuatro configuraciones que fueron examinadas:

Capas Ocultas	LR	Acc Top1	Acc Top5	CE	Normalized CE	Steps
1	10^{-4}	0.7380	0.9137	1.2739	0.1827	48000
1	$5 * 10^{-4}$	0.7941	0.9350	1.1016	0.1580	22000
0	10^{-4}	0.8527	0.9590	0.6453	0.0925	102000
0	$5 * 10^{-4}$	**0.8655**	**0.9644**	0.6349	**0.0910**	30000
Temporal Mean Pooling		0.8638	0.9569	**0.5540**	**0.0794**	**18000**

Tab. 4.10: Comparación de resultados utilizando *Self-Attention* y Temporal Mean Pooling como métodos de pooling temporal bajo distintas configuraciones.

En primer lugar, puede observarse que los resultados con *Temporal Mean Pooling* no son los mismos que en su último experimento (4.8.1), esto se debe a que en aquel experimento se utilizaba 128 como tamaño de batch mientras que al calcular embeddings on-the-fly como entrada utilizamos 32. Por lo que, se volvió a entrenar ese modelo con $BS = 32$ y, en contra lo esperado, sus métricas mejoran levemente. Al momento de buscar el BS óptimo en el experimento 4.4 128 poseía una pequeña diferencia a favor por sobre 32, en aquel experimento no utilizábamos ningún método de normalización y la MLP poseía un tamaño oculto de 4096.

En segundo lugar, observamos que la utilización de 0 capas ocultas es lo que más afectó los resultados, y con ésto se supera la accuracy obtenida con *Temporal Mean Pooling*, lo que nos motiva a seguir explorando este mecanismo. Cuando habíamos probado 0 capas ocultas en el experimento 4.5 el modelo no alcanzaba a converger en un tiempo razonable. Por lo que, al ahora ser beneficioso, significaría que con el uso de *Self-Attention* se consiguen mejores representaciones para la tarea, ya que un modelo más simple las aprovecha mejor. En cuanto al LR, en todos los casos reducirlo condujo a peores resultados y una convergencia más lenta.

A raíz de los resultados que se observan en la tabla anterior, continuaremos la exploración de este pooling con la configuración que posee 0 capas ocultas y se entrena con LR fijo en $5 * 10^{-4}$. Por otro lado, cabe mencionar que estos cuatro modelos fueron entrenados sin la normalización Z global vista en el experimento 4.7. El siguiente paso será entonces analizar el efecto de su uso, que puede ser visto en la siguiente tabla:

Normalización Z Global	Acc Top1	Acc Top5	CE	Normalized CE	Steps
✗	**0.8655**	**0.9644**	**0.6349**	**0.0910**	30000
✓	0.8496	0.9604	0.6433	0.0923	**12000**

Tab. 4.11: Efecto del uso de la normalización Z global con *Self-Attention Time Pooling* con una cabeza de atención.

Notamos que en este caso la normalización Z global lleva a un leve deterioro del desempeño. En contrapartida, con normalización Z global el modelo converge en la mitad del tiempo. Con el objetivo de agilizar la experimentación y asumiendo que la normalización Z global no interactúa con los próximos hiperparámetros, mantendremos el uso de la normalización Z global y al final de este experimento entrenaremos al modelo con la mejor configuración de *Self-Attention Time Pooling* quitando la normalización Z global.

Como siguiente paso, examinaremos el uso de *Positional Encoding*. Específicamente, usaremos el *encoding* más clásico definido en la ecuación 2.6.

Positional Encoding	Acc Top1	Acc Top5	CE	Normalized CE	Steps
✗	**0.8496**	**0.9604**	**0.6433**	**0.0923**	12000
✓	0.6114	0.8446	2.0620	0.29567	**10000**

Tab. 4.12: Resultados al usar *Positional Encoding* con *Self-Attention Time Pooling* con una cabeza de atención, sin normalización Z global y con 0 capas ocultas en la MLP.

Los resultados arrojan que no es útil el *Positional Encoding* para el pooling temporal, ya que las métricas se deterioran mucho con su uso. Si bien el modelo con este método alcanza en una etapa más temprana del entrenamiento su mejor desempeño en validación, este comportamiento solo indica que el modelo comienza a sobreajustar a los datos de entrenamiento antes empeorando en validación. Un posible problema con el *encoding* propuesto es la escala, es posible que los valores del *encoding* dominen al valor original del embedding generando una pérdida de información. Tanto analizar posibles soluciones a esto como probar otras codificaciones queda como posible trabajo futuro.

El siguiente hiperparámetro con el que buscaremos la mejor configuración para esta capa es la cantidad de cabezas de atención. Suele decirse que cada una de estas cabezas se encarga de almacenar información de distinta índole; en NLP, Clark et al. (2019) muestra que diferentes cabezas de atención se especializan en diferentes aspectos de la sintaxis. Si bien no hay garantías de que esto pueda ser interpretado por humanos, al resultar en distintos segmentos del embedding de salida y tener diferentes conjuntos de pesos, cada una de estas cabezas es libre de aprender funciones distintas. Como los embeddings suelen tener tamaños múltiplos de 2, la cantidad de cabezas de atención suelen también seguir este patrón, aunque bien podrían ser números arbitrarios. En la siguiente tabla se observa las diferentes cantidades de cabezas de atención que hemos explorado y sus resultados:

Cabezas de Atención	Acc Top1	Acc Top5	CE	Normalized CE	Steps
1	0.8496	0.9604	0.6433	0.0923	12000
8	**0.8697**	0.9646	0.5963	0.0855	10000
16	0.8691	0.9663	**0.5492**	**0.0787**	**8000**
32	0.8684	**0.9670**	0.5554	0.0796	**8000**
Temporal Mean Pooling	0.8638	0.9569	0.5540	0.0794	18000

Tab. 4.13: Impacto del número de cabezas de atención en el rendimiento del modelo con *Self-Attention Time Pooling*.

Al sumar cabezas de atención, observamos que mejora la accuracy en dos puntos aproximadamente y al mismo tiempo mejora la convergencia. Siguiendo el mismo criterio que hemos adoptado a lo largo del trabajo, nos quedaremos con el modelo de 16 cabezas de atención ya que alcanzó la menor entropía cruzada.

Por otro lado, ocurrió posteriormente por azar y error humano, que ejecutamos un entrenamiento del modelo de 16 cabezas con $LR = 10^{-4}$ obteniendo ostensiblemente mejores resultados con una convergencia diez veces más lenta. Al ahondar en la causa de esto, encontramos que las cabezas de atención interactúan con el LR: más cabezas de atención se benefician de *learning rates* más bajos. No profundizaremos en esta cuestión, pero creemos que es un buen experimento explorar esta interacción, ya que es muy grande su influencia en los resultados, que pueden ser vistos en la siguiente tabla:

LR	Acc Top1	Acc Top5	CE	Normalized CE	Steps
10^{-4}	**0.9320**	**0.9839**	**0.3227**	**0.0463**	80000
$5 * 10^{-4}$	0.8691	0.9663	0.5492	0.0787	**8000**

Tab. 4.14: Evaluación del *Self-Attention Time Pooling* con 16 cabezas de atención variando la tasa de aprendizaje.

Finalmente, como la normalización Z global no había arrojado mejores resultados (véase 4.11) con una cabeza de atención, veremos qué ocurre con este último modelo con 16 cabezas de atención al quitar esta normalización:

Normalización Z Global	Acc Top1	Acc Top5	CE	Normalized CE	Steps
✓	**0.9320**	**0.9839**	**0.3227**	**0.0463**	**80000**
✗	0.9179	0.9773	0.4055	0.0581	152000

Tab. 4.15: Resultados de la utilización de normalización Z global con *Self-Attention Time Pooling* en su configuración con 16 cabezas de atención y $LR = 10^{-4}$.

Anteriormente, utilizar normalización Z global significaba un leve deterioro de la entropía cruzada y peor accuracy. En este caso, la normalización Z global es beneficiosa para todas las métricas que monitoreamos. Por lo que conservaremos el uso de la normalización Z global, y con esto concluimos nuestra búsqueda del mejor pooling temporal con *Self-Attention*. La configuración final es usar un downstream de 0 capas ocultas, $LR = 10^{-4}$, con normalización Z global, sin *Positional Encoding* y con 16 cabezas de atención.

Mecanismos alternativos de atención En la búsqueda del mejor mecanismo de atención, exploramos dos alternativas: *Transformers* (Vaswani et al., 2017) y *SummaryMixing*

(Parcollet et al., 2024). *Transformers* es una extensión natural del anterior experimento, ya que mantiene el bloque de *Self-Attention* mientras suma una capa de normalización e incorpora una MLP al final. Por su parte, *SummaryMixing* es una alternativa de menores recursos a *Self-Attention* propuesta para reconocimiento automático del habla (ASR), que reduce la complejidad cuadrática en la cantidad de *frames* de su contraparte a una complejidad lineal. A su vez, hay disponibles dos sabores de este mecanismo: su versión *lite* y su versión completa, que incorpora bloques de MLPs.

En cuanto a la configuración de *Transformers*, utilizamos un único bloque con una única capa en su MLP con tamaño oculto 2048 (default de PyTorch). Al mismo tiempo, mantuvimos la configuración óptima de *Self-Attention* que encontramos en el anterior experimento.

De esta manera, utilizamos estos mecanismos de atención alternativos siguiendo la misma idea que con *Self-Attention*, propagamos las activaciones por la capa que realiza este mecanismo y luego promediamos en la dimensión temporal.

Método de pooling	#Params	Acc Top1	Acc Top5	CE	Normalized CE	Steps
Temporal Mean Pooling	0	0.8638	0.9569	0.5540	0.0794	**18000**
Self-Attention	2.4M	0.9320	0.9839	0.3227	0.0463	80000
Summary Mixing Lite	50.4K	0.5433	0.7645	2.3168	0.3322	614000
Summary Mixing	1.1M	0.7458	0.8988	1.1641	0.1669	58000
Transformer	11M	**0.9435**	**0.9875**	**0.2229**	**0.0320**	48000

Tab. 4.16: Resultados al usar diferentes métodos de pooling en el tiempo basados en mecanismos de atención en comparación con los anteriores métodos vistos. La cantidad de parámetros corresponde a la capa de pooling temporal.

En primer lugar, observamos que *Transformer* logra superar a los otros métodos en todas las métricas menos convergencia. Aún así, converge en una menor cantidad de pasos que *Self-Attention*, lo cual es esperable debido a su mayor complejidad. En consecuencia, en lo sucesivo usaremos *Transformer* como pooling temporal.

En segundo lugar, notamos que *Summary Mixing*, aunque puede ser un buen reemplazo de *Self-Attention* dentro de la arquitectura *Transformer* para ASR, no es un método adecuado para realizar un pooling temporal. Los resultados obtenidos son significativamente peores que los de *Self-Attention* e incluso inferiores a los de *Temporal Mean Pooling*, que no posee parámetros entrenables.

4.8.2. Pooling de capas

El pooling de capas consiste en extraer información del embedding de cada capa. Luego de este pooling la dimensión de las capas es colapsada. En este experimento, probaremos si los métodos que fueron exitosos para el pooling temporal también son efectivos para el pooling de capas. De igual manera que con los mecanismos de atención en el pooling temporal, luego de pasar por estas capas promediaremos, esta vez, en la dimensión correspondiente a las capas. Hasta este momento hemos utilizado el promedio pesado como pooling de capas (*Weighted Average Layer Pooling*), que al entrenar un peso por cada capa nos otorga la capacidad de interpretar la importancia de la información de cada una para la tarea dada. En la práctica, la técnica más difundida al momento de usar embeddings es quedarse con la representación de la última capa. Por lo tanto, también exploraremos el uso de ésta. A su vez, sumaremos un método más, *Mejor Capa*, que consiste en di-

rectamente quedarse con el embedding correspondiente a la mejor capa. El criterio para determinar cuál es esta capa será seleccionar aquella cuyos pesos otorgados en el *Weighted Average* son máximos, en el siguiente gráfico puede observarse los pesos del experimento anterior:

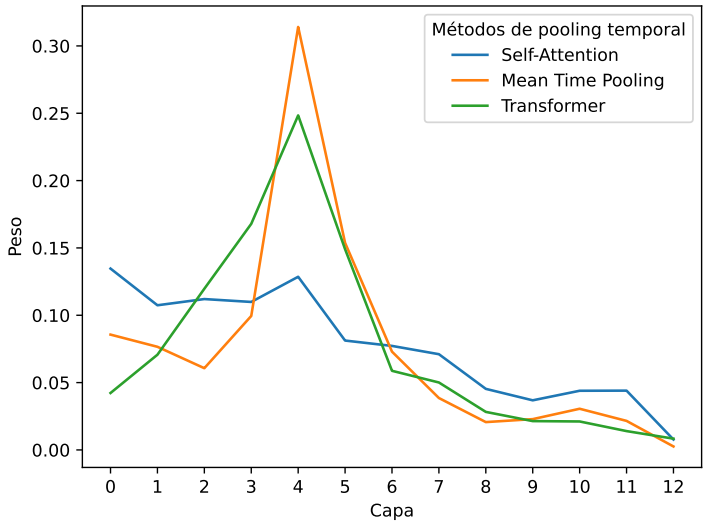


Fig. 4.6: Pesos de la capa Weighted Average Layer Pooling para los modelos comparados en el experimento 4.8.1.

La Figura 4.6 muestra que la capa 4 tiene máximo peso para dos de los tres métodos de pooling temporal. Esta conclusión es consistente con la obtenida en el paper original de WavLM (Chen et al., 2021), donde para WavLM Base+ se observa un gran dominio de la capa 4 en SID en el análisis de pesos de las capas. Cabe destacar, que si bien la capa 4 posee el mayor peso, esto no implica que sea necesariamente la capa con mejores resultados al usar únicamente las representaciones de ésta.

Así experimentamos con diferentes métodos de pooling de capas usando *Transformer* como pooling temporal:

Método	**#Parámetros**	**Acc Top1**	**Acc Top5**	**CE**	**Normalized CE**	**Steps**
Weighted Average	13	0.9435	**0.9875**	**0.2229**	**0.0320**	48000
Última capa	0	0.6752	0.8578	1.5177	0.2176	44000
Mejor capa	0	0.9450	0.9876	0.2253	0.0323	**40000**
Self-Attention	2.4M	0.9139	0.9769	0.4719	0.0677	54000
Transformer	11M	**0.9470**	0.9858	0.2294	0.0329	**40000**

Tab. 4.17: Resultados al usar diferentes mecanismos de poolings de capas usando *Transformer Time Pooling*. La cantidad de parámetros corresponde a la capa de pooling de capas del modelo upstream.

El primer resultado saliente es que la utilización de las representaciones de la última capa por sí solas se traducen en un gran deterioro en el desempeño. Si bien es una técnica adecuada para otros contextos (otros modelos con otras tareas u otras disciplinas como NLP) para este modelo upstream y esta tarea no lo es. Contrariamente, usando la capa 4

se obtiene resultados comparables con los de de *Weighted Average*, por lo que es razonable pensar que esta capa encapsula la información necesaria del upstream para esta tarea. Por su parte, usar *Self-Attention* en el pooling de capas trajo peores resultados, mientras que con *Transformer* se obtuvo una pequeña mejora en Accuracy Top1 manteniendo una buena convergencia. Esto significaría que el mecanismo de atención por sí solo no es suficiente para el pooling de capas, pero sí lo es al ser usado con una MLP y demás incorporaciones de la arquitectura *Transformer*. A su vez, recordemos que la cantidad de steps no es indicativa del tiempo de entrenamiento ya que a mayor cantidad de parámetros más demora cada step, así la convergencia más rápida pertenece a *Mejor capa*. De esta manera, con estos resultados la mejor configuración mantiene *Weighted Average* como mejor pooling de capas, ya que con una cantidad mínima de parámetros obtiene la menor cross entropy y *accuracies* comparables con las mejores.

4.8.3. El estado del arte

En este punto de la experimentación nos detuvimos a observar y entender mejor el estado del arte en esta tarea. Como mencionamos anteriormente, el benchmark que se suele observar es SUPERB (Yang et al., 2021). En la tabla de resultados en SID del benchmark se publica únicamente la accuracy top1 de cada modelo en la partición de test, y con WavLM Base+ se reporta una accuracy de 0,8942. El downstream utilizado para esto utiliza lo que denominamos *Weighted Average Layer Pooling*, seguido de una capa intermedia, y luego *Temporal Mean Pooling*. La capa intermedia permitiría una adaptación del input del pooling temporal. A su vez, es usado para reducir la dimensión de 768 a 256. Esto último sospechamos que se debe a que el resto del modelo downstream es compartido por diferentes modelos con diferentes dimensiones en las representaciones del upstream. Con la accuracy mencionada WavLM Base+ ocupa el tercer lugar, mientras que los primeros dos son WavLM Large (316.62M de parámetros y accuracy 0,9549) y HuBERT Large (300M de parámetros y accuracy 0,9033) (véase la tabla 2.1).

Nuestro modelo con *Temporal Mean Pooling* alcanzaba una accuracy top1 de 0,8638 con cross entropy 0,5540, por lo que buscamos entender esta diferencia. Así, como es de código abierto, decidimos replicar los resultados utilizando el código provisto. Al correr el código como se indica en la documentación, se obtuvo una accuracy máxima de 72,12 en 200.000 steps (este limite de steps es el predeterminado por la biblioteca):

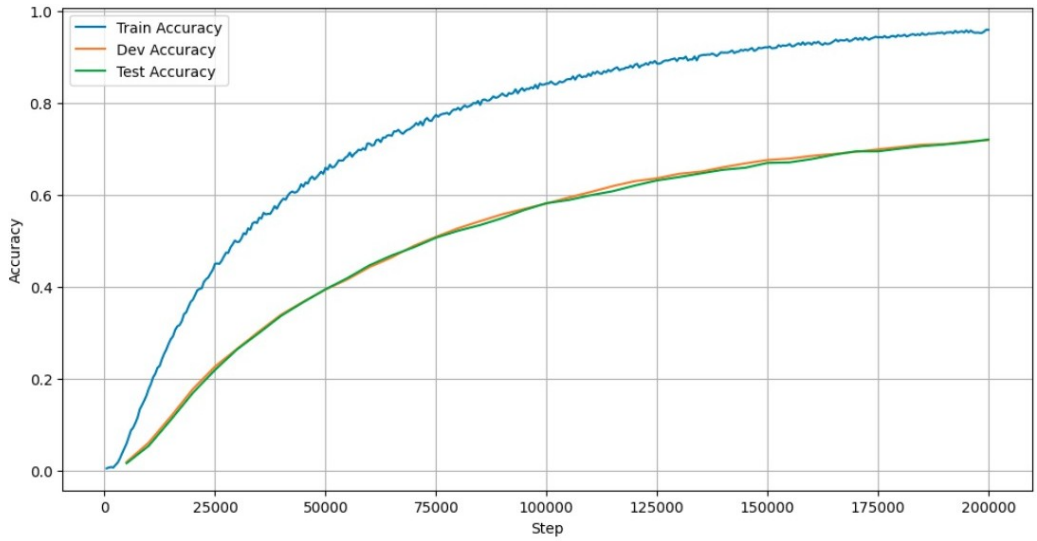


Fig. 4.7: Evolución de la accuracy del modelo predeterminado de S3PRL con WavLM Base+.

Al observar que el modelo no convergía, decidimos probar aumentando el LR. Anteriormente estaba en 0,0001 y lo aumentamos a 0,001:

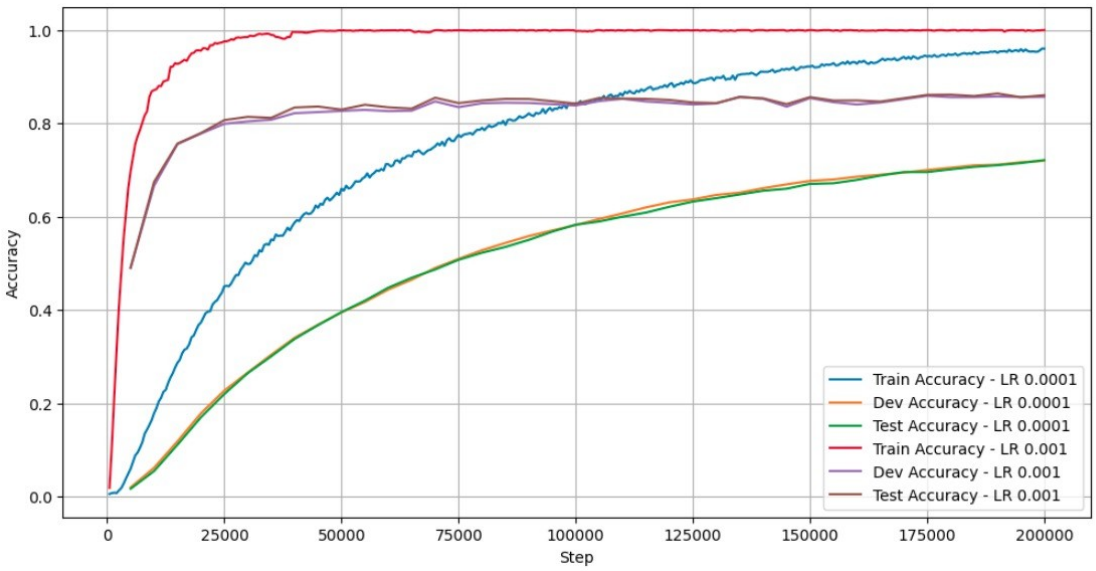


Fig. 4.8: Evolución de la accuracy con diferentes LRs del modelo predeterminado de S3PRL con WavLM Base +.

De esta manera el modelo sí convergió y logró obtener 0,8610 de accuracy top 1 con cross entropy 0,9027, una accuracy similar con una cross entropy mucho mayor a nuestro modelo así como también una convergencia más tardía. La diferencia en la convergencia está en linea con los resultados de la sección 4.7, donde incorporamos el uso de normalización Z global. Aún así, no se alcanzó la accuracy reportada y puede que requiera algunos ajustes más, aunque al no ser el foco de nuestra investigación decidimos no continuar con esta exploración. Mientras que construir el modelo downstream desde cero fue muy enriquecedor, a partir del experimento realizado, hemos aprendido que es fundamental

priorizar, en las etapas iniciales de la experimentación, un entendimiento profundo de los *baselines*, puntos de referencia del estado del arte.

De este experimento nos llevamos algunas ideas interesantes. En primer lugar, el orden de los poolings utilizado es distinto, por lo que en un próximo experimento exploraremos esto. En segundo lugar, se utiliza una capa intermedia entre poolings, si bien puede que su razón se deba simplemente a la adaptación de distintos upstreams, nos parece una buena idea darle al modelo una oportunidad de adaptar los features antes de poolearlos en el tiempo. Por otro lado, en este benchmark no hay reportes de entropía cruzada, y los entrenamientos de estos modelos buscan la mayor accuracy sin monitorear entropía cruzada, que sí se usa como función de pérdida. En nuestro caso, nos concentramos en minimizar la entropía cruzada, y esto puede ocasionar diferencias en los resultados. Finalmente, notamos que nuestros mejores resultados al momento se posicionarían en un segundo lugar con menos de un tercio de la cantidad de parámetros que WavLM Large y potencialmente en mucho menor tiempo de entrenamiento (menos steps y con un modelo más pequeño).

4.8.4. Orden de los poolings

Motivados por el orden inverso en los poolings usado en el downstream del SUPERB benchmark, decidimos explorar el orden de los poolings y sus posibles combinaciones. En nuestro caso y en su momento, el orden era conveniente para poder pre-calcular los embeddings luego de pasar por el pooling temporal. En este momento, al usar poolings con parámetros a aprender, el orden no afecta tanto el tiempo de ejecución de cada step.

De esta manera, probamos distintas combinaciones y ordenes de los poolings estudiados, seleccionando los que mejores resultados habían obtenido anteriormente:

Primer pooling	**Segundo pooling**	**Acc Top1**	**Acc Top5**	**CE**	**NCE**	**Steps**
Weighted Average Layer Pooling	Temporal Mean Pooling	0.8702	0.9601	0.5317	0.0762	20000
Temporal Mean Pooling	Weighted Average Layer Pooling	0.8638	0.9569	0.5540	0.0794	**18000**
Weighted Average Layer Pooling	Self-Attention Time Pooling	0.9206	0.9819	0.3621	0.0519	96000
Self-Attention Time Pooling	Weighted Average Layer Pooling	0.9320	0.9839	0.3227	0.0463	80000
Weighted Average Layer Pooling	Transformer Time Pooling	0.9281	0.9828	0.2736	0.0392	44000
Transformer Time Pooling	Weighted Average Layer Pooling	0.9435	0.9875	**0.2229**	**0.0320**	48000
Mejor capa	Temporal Mean Pooling	0.8662	0.9433	0.5960	0.0855	58000
Mejor capa	Self-Attention Time Pooling	0.9226	0.9796	0.3340	0.0479	36000
Mejor capa	Transformer Time Pooling	0.9450	**0.9876**	0.2253	0.0323	40000
Self-Attention Layer Pooling	Self-Attention Time Pooling	0.9127	0.9781	0.4553	0.0653	48000
Self-Attention Time Pooling	Self-Attention Layer Pooling	0.8921	0.9729	0.5719	0.0820	52000
Transformer Layer Pooling	Transformer Time Pooling	0.9406	0.9860	0.2433	0.0349	70000
Transformer Time Pooling	Transformer Layer Pooling	**0.9470**	0.9858	0.2294	0.0399	40000

Tab. 4.18: Resultados al combinar los distintos poolings vistos en diferentes órdenes.

De los resultados se desprende que no hay un orden entre pooling temporal y de capas que sea siempre mejor; depende de cada modelo. Usando *Weighted Average Layer Pooling* y *Temporal Mean Pooling* se obtiene una pequeña mejora utilizando el orden introducido en S3PRL (primero pooling de capas). A su vez, la configuración *Transformer Time Pooling* con *Weighted Average Layer Pooling* se mantiene con la cross entropy mínima, mientras que Transformer Time Pooling con Transformer Layer Pooling tiene máxima accuracy top1. Por otro lado, cabe mencionar que al emplear *Mejor capa* no tiene sentido intercambiar el orden ya que no hay interacción entre capas en el pooling temporal.

4.8.5. Capa entre poolings

El modelo downstream de SUPERB luego de pasar por el upstream y el pooling de capas hace una pasada por una capa intermedia antes de realizar un pooling temporal. Esta capa permitiría una adaptación del input al pooling temporal. A su vez, es usado para reducir la dimensión de 768 a 256. En nuestro caso no reduciremos la dimensión sino que mantendremos la dimensión en 768 tanto en la entrada como en la salida de esta capa intermedia, así esta capa realiza una transformación lineal que incorpora 590k parámetros adicionales a entrenar.

Experimentaremos con esta capa intermedia con los que consideramos los tres modelos más atractivos para esto, aquellos con los siguientes poolings: Self-Attention Time Pooling + Weighted Average Layer Pooling, Transformer Time Pooling + Weighted Average Layer Pooling, y Transformer Time Pooling + Transformer Layer Pooling. Respectivamente, estas configuraciones poseen 2.4M, 11M y 22M parámetros a entrenar, barriendo así diferentes complejidades. El segundo fue el modelo que menor cross entropy obtuvo en el experimento anterior, mientras que el tercero fue el que llegó a la mejor accuracy. Por su parte, el primero tiene la configuración que alcanza los mejores resultados con su tamaño reducido, siendo así una buena alternativa ante limitaciones de recursos.

Primer pooling	**Capa intermedia**	**Segundo pooling**	**Acc Top1**	**Acc Top5**	**CE**	**NCE**	**Steps**
Self-Attention	✗	Weighted Average	0.9320	0.9839	0.3227	0.0463	80000
Self-Attention	✓	Weighted Average	0.9110	0.9800	0.4075	0.0584	44000
Transformer	✗	Weighted Average	0.9435	**0.9875**	0.2229	0.0320	48000
Transformer	✓	Weighted Average	0.9020	0.9783	0.4332	0.0621	**40000**
Transformer	✗	Transformer	0.9470	0.9858	0.2294	0.0399	**40000**
Transformer	✓	Transformer	**0.9474**	0.9874	**0.2107**	**0.0302**	**40000**

Tab. 4.19: Resultados al incorporar una capa intermedia lineal entre los poolings. En los tres casos, el primer pooling es en el tiempo y el segundo pooling es a nivel capas.

De esta manera, se obtuvo que *Transformer Time Pooling + Transformer Layer Pooling* con una capa intermedia obtiene la menor cross entropy, y a la vez, la mayor accuracy top1.

Notamos que en los otros dos casos, donde el pooling posterior a la capa intermedia es *Weighted Average Layer Pooling*, los resultados son peores aunque la convergencia es más rápida. El efecto en los resultados puede parecer razonable: esta capa intermedia está realizando una proyección del embedding de cada capa (obtenido luego del pooling temporal), que podría verse como una adaptación previa al pooling de capas. En esa línea, se vio en el experimento 4.7 que la normalización Z por capas no mejoró a la normalización Z global, indicando que no hay grandes diferencias entre las representaciones de cada capa. Por lo cual, no habría beneficio en adaptar los embeddings de cada capa antes de poolearlos. Sin embargo, *Transformer Layer Pooling* sí se ve beneficiado por la capa intermedia anterior. Ello podría deberse a que este pooling, a diferencia del promedio pesado, combina la información interna de cada capa, nutriendo al embedding de cada capa del contexto de las otras capas.

Por otro lado, la convergencia más rápida es consistente con los resultados que fuimos obteniendo hasta aquí, donde los modelos más complejos convergen más rápido. En el caso de *Transformer Time Pooling + Transformer Layer Pooling* tiene sentido que no sea observable una mejor convergencia si observamos el aumento relativo de la complejidad al sumar la capa intermedia: para *Self-Attention + Weighted Average Layer Pooling* significa

un aumento del $16,81\,\%$, para *Transformer Time Pooling + Weighted Average Layer Pooling* $4,96\,\%$, mientras que para *Transformer Time Pooling + Transformer Layer Pooling* $2,58\,\%$.

4.9. Interacción entre el cross entropy y la accuracy

A lo largo de la experimentación, fuimos notando que al detener el entrenamiento con *Early Stopping*, si bien la entropía cruzada había convergido, la *accuracy* seguía mejorando. Surgió entonces como hipótesis que al extender el entrenamiento encontraríamos modelos con mejor *accuracy* al costo de peor entropía cruzada. Este fenómeno es relevante porque en la actualidad la publicación de resultados en tareas de clasificación como SID mantiene como práctica reportar la accuracy de los modelos como única métrica. La diferencia entre la entropía cruzada y la accuracy es que la entropía cruzada evalúa la calidad de las posteriors y la accuracy la calidad de las decisiones que se toman con esas posteriors para una función de costo específica donde todos los errores valen lo mismo. Por un lado, la entropía cruzada no depende de cómo se toman las decisiones. Por el otro lado, siempre es posible calibrar el modelo de mejor accuracy manteniendo la discriminación y logrando una mejor calidad de posteriors. En todo caso, al menos, no debemos ignorar a la entropía cruzada.

En este experimento, decidimos entrenar los 3 modelos del experimento anterior, en sus configuraciones con mejor desempeño, por 250000 steps observando sus evoluciones en cross entropy y *accuracy*.

De esta manera, en el modelo con *Self-Attention Time Pooling + Weighted Average Layer Pooling* pudimos observar el fenómeno que entreveíamos y se grafica en la figura 4.9. Si bien la cross entropy alcanza su convergencia a los 74000 steps, la accuracy máxima no se logra hasta los 218000 steps. El modelo que alcanza la mayor accuracy supera por un punto porcentual al modelo con menor cross entropy, aunque presenta aproximadamente un punto más de normalized cross entropy. Esta observación nos lleva a una conclusión importante sobre la metodología de entrenamiento: para optimizar el desempeño en términos de accuracy, es preferible seleccionar el modelo con la mayor accuracy y posteriormente realizar un proceso de calibración. Este paso de calibración es fundamental, ya que la selección basada en accuracy tiende a producir modelos *descalibrados*, es decir, modelos cuyas probabilidades de salida no reflejan con accuracy su verdadera confianza, típicamente sobre-estimando sus predicciones.

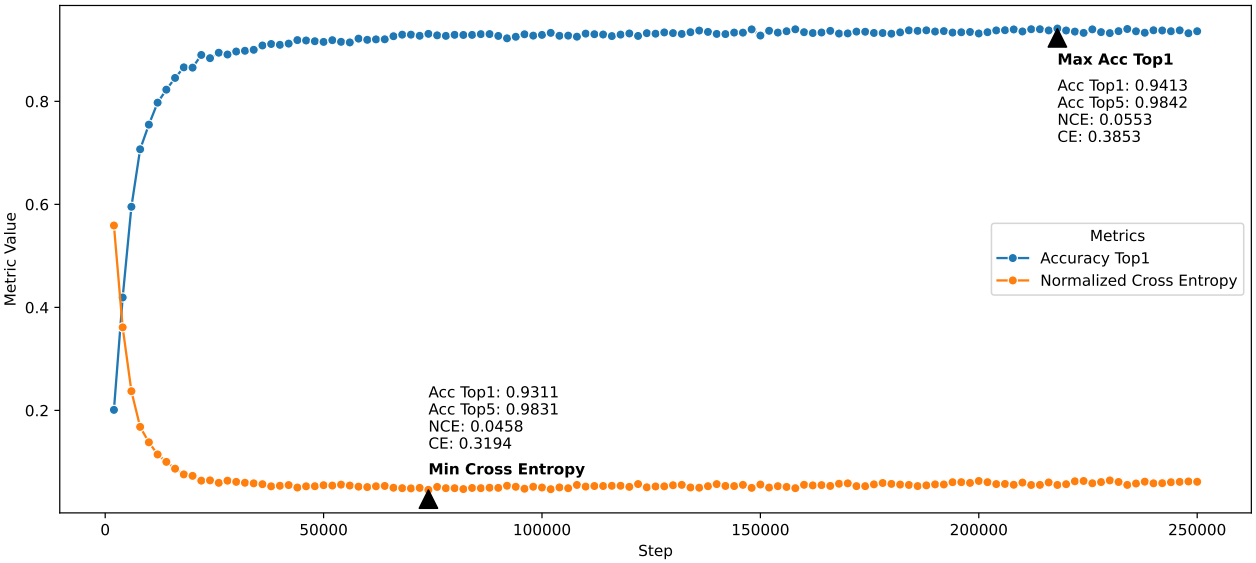


Fig. 4.9: Evolución de las métricas en validación durante el entrenamiento para el modelo con *Self-Attention Time Pooling + Weighted Average Layer Pooling*.

No obstante, este comportamiento no se observó en el modelo con mejores resultados del experimento 4.8.5, como se puede observar en la siguiente figura:

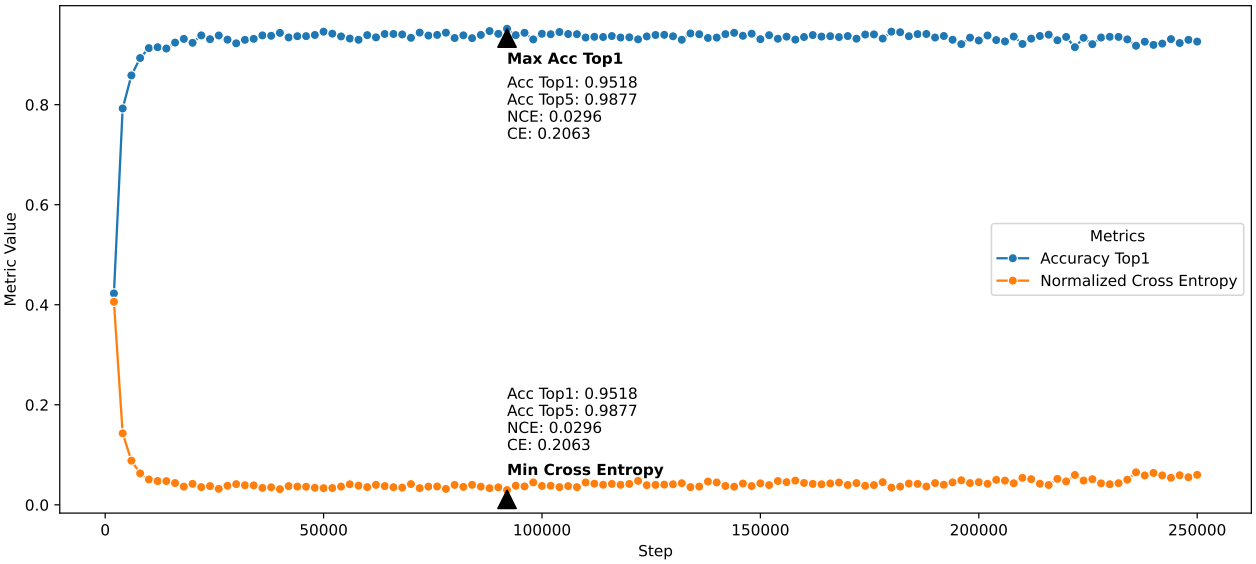


Fig. 4.10: Evolución de las métricas en validación durante el entrenamiento para el modelo con *Transfomer Time Pooling* + Capa intermedia + *Transfomer Layer Pooling*.

Por último, el modelo con *Transformer Time Pooling + Weighted Average Layer Pooling* reveló un comportamiento inesperado. A diferencia del experimento anterior, donde este modelo no había obtenido los mejores resultados, en esta ocasión logró superar a los demás modelos tanto en cross entropy como en accuracy después de 214000 steps:

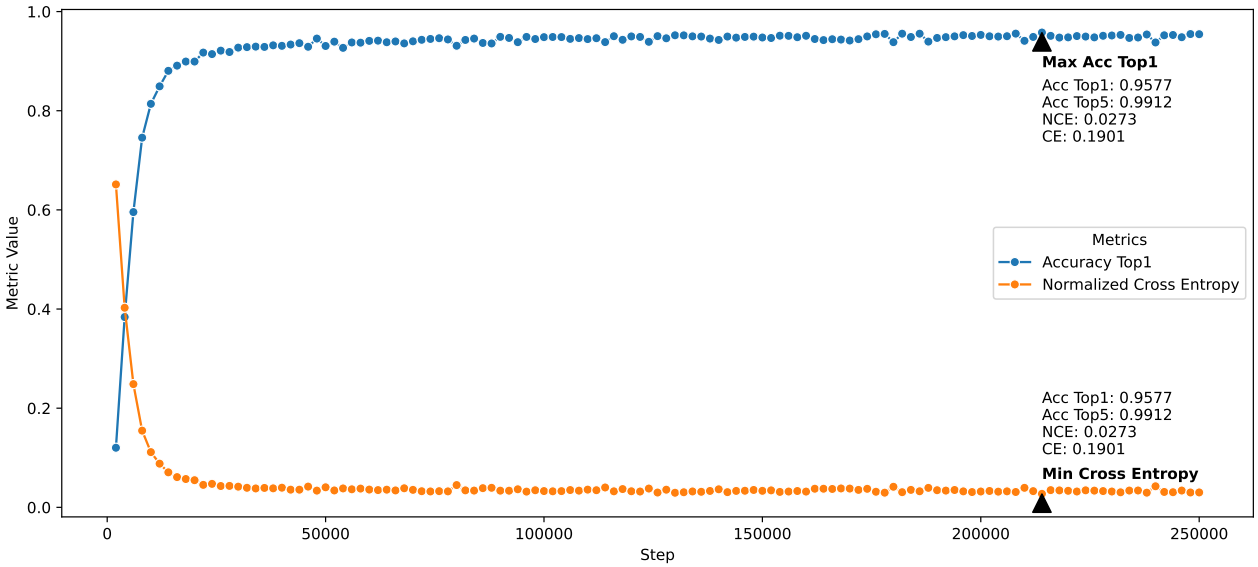


Fig. 4.11: Evolución de las métricas en validación durante el entrenamiento para el modelo con *Transformer Time Pooling + Weighted Average Layer Pooling*.

Esta observación cuestiona la efectividad del *Early Stopping* en algunos casos, ya que en este caso logró mejores resultados sin su uso (al utilizar Early Stopping habíamos obtenido 0.9435 de accuracy y 0.2229 de entropía cruzada en 48000 steps, véase la Tabla 4.19). Al no ser sencillo identificar cuáles serán estos casos donde el Early Stopping no es efectivo, puede llegar a ser conveniente fijar la cantidad de steps a realizarse o monitorear oscilaciones que determinen mejor cuando un modelo convergió. Aún así, en la figura anterior, se observa un caso en el que el *Early Stopping* sí funcionó adecuadamente para encontrar al mejor modelo.

Es importante aclarar que este experimento fue replicado en una etapa final de la investigación, posterior a la secuencia experimental que se presenta a continuación en esta tesis. Si bien los resultados obtenidos cuestionan la eficacia del *Early Stopping* y demuestran un mejor desempeño del modelo *Transformer Time Pooling + Weighted Average Layer Pooling*, los experimentos subsiguientes mantienen el uso de *Early Stopping* por sus ventajas computacionales. Asimismo, se continuará considerando al modelo *Transformer Time Pooling* + Capa intermedia + *Transformer Layer Pooling* como el modelo downstream óptimo obtenido en nuestra investigación.

4.10. Conclusiones de la exploración del modelo downstream

En este capítulo hemos llevado a cabo una exploración exhaustiva de la configuración del modelo downstream con el objetivo de optimizar su desempeño en la tarea de identificación de hablantes. A continuación, sintetizamos los principales hallazgos obtenidos en cada uno de los aspectos evaluados.

Uno de los elementos más analizados fue la tasa de aprendizaje, donde se observó que los valores predeterminados en la literatura (*i.e.*, $10^{-3}, 5 * 10^{-4}, 10^{-4}$) suelen ser los más estables (ver Sección 4.4) . Adicionalmente, se evidenció la clara interacción entre el LR y el tamaño del batch, permitiendo cuantificar el grado de degradación del desempeño al reducir BS. Esto nos proporcionó información clave para entender cómo se modificaría

el desempeño con nuestras limitaciones de recursos. Más adelante, también se notó una relación entre la cantidad de cabezas de atención y el LR, sobre la que no se profundizó, pero fue uno de los factores más influyentes al usar *Self-Attention*.

En los primeros experimentos observamos que el modelo downstream escalaba mejor a lo ancho que a lo largo: no mejoraba al aumentar la cantidad de capas pero sí al aumentar el tamaño oculto. De esta manera, la MLP propuesta inicialmente antes de la capa de salida quedó reducida a una única capa.

Luego, al utilizar el conjunto de datos completo, detectamos que los modelos más grandes eran más propensos al sobre-ajuste. Esto nos llevó a reconsiderar la configuración y optar por un modelo con un tamaño oculto menor. Esta decisión resalta la importancia de evaluar el desempeño del modelo en diferentes escalas del dataset. Aún así, la utilización de una partición reducida de datos fue vital en las etapas iniciales para poder realizar una exploración minuciosa antes de escalar los experimentos al conjunto de datos completo. Otro factor determinante en la factibilidad de nuestra exploración fue la estrategia de precálculo de embeddings del modelo upstream, que redujo el tiempo de entrenamiento en más de un orden de magnitud. Gracias a esta técnica, pudimos realizar experimentaciones más extensas y con mayor granularidad en los hiperparámetros sin que el costo computacional fuera prohibitivo. Posteriormente, la normalización de los embeddings mediante el normalización Z global propuesto mejoró notoriamente la convergencia, reduciendo a un tercio la cantidad de steps en el entrenamiento.

Finalmente, en lo que respecta a los métodos de *pooling*, la exploración progresiva de diferentes técnicas y la incorporación de mecanismos de atención fue clave para optimizar el desempeño final del modelo. El promedio de los embeddings, siendo la estrategia más simple, se sostuvo como una técnica adecuada y la estadística más informativa. Luego, la investigación de los mecanismos de atención abarcó desde configuraciones básicas hasta arquitecturas más sofisticadas como *SummaryMixing* y *Transformers*. En esta investigación la cantidad de cabezas de atención en conjunto con un ajuste del LR demostraron ser los factores más influyentes en el rendimiento, como se mencionó anteriormente. La configuración óptima se logró utilizando *Transformers* tanto para el pooling temporal como para el de capas. En cuanto a la arquitectura general del modelo, los experimentos revelaron que el impacto del orden de los poolings está estrechamente ligado a los distintos métodos de pooling y no hay un orden que sea siempre superior. Análogamente, el efecto de una capa intermedia entre poolings demostró ser altamente dependiente de la configuración específica, sugiriendo que la arquitectura óptima requiere una consideración cuidadosa de estas interacciones.

En resumen, la exploración detallada del modelo downstream nos permitió no solo alcanzar resultados de vanguardia en SID (0,9474 de accuracy top 1 y 0,2107 de cross entropy) con un modelo upstream pequeño, sino también entender los fundamentos detrás de cada decisión de diseño y entrenamiento. Estos hallazgos no solo optimizaron nuestro modelo final, sino que también ofrecen lineamientos valiosos para futuras investigaciones en optimización de modelos de habla en escenarios de recursos limitados.

5. FINE-TUNING

Hasta este punto, hemos buscado el modelo downstream con la arquitectura y configuración de hiperparámetros óptima manteniendo siempre congelado al modelo upstream, es decir, sin modificar sus pesos durante los entrenamientos. En esta sección, entrenaremos también al modelo upstream, analizando sus costos y beneficios. Acuñaremos el término *full-finetuning* para referirnos al entrenamiento con el upstream completamente descongelado. Si bien no analizaremos descongelamientos parciales del modelo upstream, este término explicita que el descongelamiento es total y nos permitirá distinguirlo del uso de técnicas PEFT que se verá más adelante.

Como modelo downstream, seleccionamos el que utiliza *Transformer Time Pooling*, una capa intermedia, y *Transformer Layer Pooling*. Este modelo es el más grande entre los modelos downstream evaluados, con 23.6M de parámetros. Aunque también identificamos modelos más pequeños con resultados similares, al realizar *full-finetuning* las diferencias en el tamaño de los modelos downstream no son tan significativas. En este contexto, los recursos computacionales necesarios entre los diferentes modelos downstream no resultan en diferencias relativas considerables, por lo que tiene sentido en este capítulo evaluar *full-finetuning* utilizando el mejor downstream independientemente de su cantidad de parámetros.

5.1. Programación de la tasa de aprendizaje

El primer y más sencillo experimento es simplemente descongelar el upstream. Sin embargo, al hacer esto nos encontramos con que no se obtuvieron mejores resultados (véase la tabla 5.2) incluso con mucho más parámetros entrenables. No sólo eso, sino que los resultados son significativamente peores. Entonces, al ahondar en las causas, nuestro primer acercamiento fue sospechar del LR. Estos modelos suelen utilizar estrategias más complejas que un LR fijo ya que nos encontramos entrenando un orden de magnitud de parámetros más grande y múltiples capas de *Transformers*. Por consiguiente, indagamos acerca de las estrategias de fine-tuning empleadas en tareas análogas y así arribamos a un reporte [1] en el repositorio oficial del modelo. En este sitio reportan la programación del LR que usaron los autores para Speaker Verification (SV), una tarea íntimamente ligada a SID.

La programación de LR planteada en ese foro incluye tres etapas, donde en la primera solo se entrena al downstream, y en las otras dos, a ambos. En todas las etapas se utiliza descenso por el gradiente estocástico (SGD) con momentum como optimizador (a diferencia de lo que venimos usando nosotros hasta aquí, que es el optimizador Adam) y una programación de LR con algunas épocas de *warm-up* y luego un decrecimiento exponencial. Esta estrategia hace uso de los siguientes conceptos clave:

- *Momentum* (Polyak, 1964): Esta técnica acelera el descenso por el gradiente al acumular un vector de velocidad v en direcciones donde persistentemente se reduce la función de pérdida. Formalmente, en su versión clásica se modifica la ecuación de actualización de parámetros (2.3) de la siguiente manera:

[1] https://github.com/microsoft/unilm/issues/695

$$v^{t+1} = \mu v^t - \eta \nabla L(\theta^t, \mathcal{D}) \tag{5.1}$$

$$\theta^{t+1} = \theta^t + v^{t+1} \tag{5.2}$$

Donde $\mu \in [0, 1]$ es el coeficiente del *momentum*. Para mayor detalle, Sutskever et al. (2013) explica la importancia de la inicialización del LR así como el uso del *momentum*.

- *Weight decay*: También conocida como regularización L2, es una técnica de regularización que modifica la función de pérdida agregando un término de penalización proporcional a la norma euclidiana de los parámetros:

$$L_{nueva}(\theta, \mathcal{D}) = L_{anterior}(\theta^t, \mathcal{D}) + \lambda \left\|\theta\right\|_2^2 \tag{5.3}$$

Donde λ es un valor que determina la magnitud de la penalización, haciendo que valores más altos promuevan parámetros de menor magnitud.

- *Warm-up*: Consiste en aumentar gradualmente desde cero hasta un valor objetivo al LR durante la etapa inicial del entrenamiento. La etapa de warm-up permite que el modelo se adapte gradualmente a los patrones en los datos sin sobre-ajustarse prematuramente a características específicas o incidentales que podrían aparecer al inicio del entrenamiento. De esta manera, el *warm-up* actúa como un mecanismo de regularización inicial. Kalra and Barkeshli (2024) muestra que el *warm-up* permite utilizar mayores LRs así como también previene la divergencia del modelo durante el entrenamiento.

- *Exponential Decay*: Empezar con un LR grande y luego decrementarlo múltiples veces es una técnica *de facto* en el entrenamiento de redes neuronales modernas que ha mostrado resultados empíricos favorables. Si bien hay discusión acerca de las causas de esto (You et al., 2019), mientras un LR inicial grande evita mínimos locales, el decrecimiento ayuda al modelo a converger e incluso aprender patrones más complejos (que requieran cambios sutiles en los pesos).

Antes de definir las configuraciones concretas de cada etapa de la programación de LR, una cuestión muy importante a tener en cuenta es que en la primera etapa reportan haber usado 1024 como tamaño de batch, en la segunda 512 y en la tercera 192. Nosotros hemos estado restringidos a un batch de 32, y como concluimos del experimento 4.4, este parámetro tiene interacción con el LR, por lo que debemos ajustar los valores dados para el $BS = 32$. Una manera de solucionar esto es ajustar siguiendo una proporción lineal (Goyal et al., 2017):

$$nuevo\_lr = anterior\_lr * \frac{nuevo\_batch\_size}{anterior\_batch\_size} \tag{5.4}$$

La siguiente tabla resume la configuración usada en cada una de las tres etapas de la estrategia planteada con el escalamiento mencionado:

Etapa	Modelo a optimizar	Weight Decay	Warm-up Epochs	LR after warm-up	LR Final
1	Downstream	$4 * 10^{-4}$	1	$2 * 10^{-2}$	$1{,}25 * 10^{-5}$
2	Upstream + Downstream	10^{-4}	3	$5 * 10^{-4}$	$2{,}75 * 10^{-4}$
3	Upstream + Downstream	10^{-4}	2	$1{,}33 * 10^{-4}$	$3{,}33 * 10^{-5}$

Tab. 5.1: Valores de los distintos hiperparámetros para las distintas etapas de la programación de LR propuesta.

En todas las etapas se utiliza $momentum = 0{,}9$.

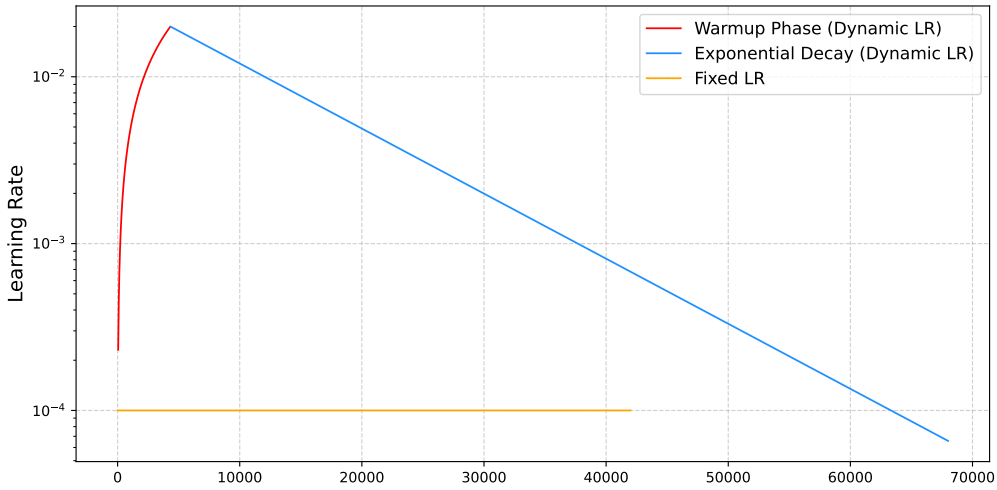
Como primer acercamiento al *fine-tuning* experimentamos con *full fine-tuning*, entrenando conjuntamente todos los parámetros de los modelos upstream y downstream. En este primer experimento exploramos *full fine-tuning* sin la etapa inicial en la que se entrena solo el downstream. Para ello primero consideramos usar la configuración de la etapa 2 de la programación de LR, que está pensada para usarse con el upstream descongelado. Sin embargo, al usar esta configuración para entrenar a todo el modelo sin haber entrenado al downstream antes se alcanzaron resultados peores que utilizando un LR fijo. Creemos que esto se debe a que la etapa 2 está pensada como refinamiento partiendo de un downstream entrenado, a su vez se tiene un especial cuidado en no sobre-ajustar al realizar *full fine-tuning*, por lo que es más conservador en la modificación de pesos. Luego, probamos utilizando la configuración del LR de la etapa 1 pero optimizando ambos modelos, cuyos resultados son mostrados en la Tabla 5.2 junto con los obtenidos al usar un LR fijo. Para una obtener una comparación completa también entrenamos al modelo downstream con el upstream congelado utilizando la configuración de la etapa 1 de la programación de LR o un LR fijo. Los valores de la tabla con LR 'dinámico' corresponden al uso de la configuración de la etapa 1.

LR	Upstream Descongelado	Acc Top1	Acc Top5	CE	Normalized CE	Steps
Fijo	✗	0.9474	0.9874	**0.2107**	**0.0302**	40000
Fijo	✓	0.8241	0.9377	0.7553	0.1083	**22000**
Dinámico	✗	**0.9617**	**0.9899**	0.2186	0.0314	82000
Dinámico	✓	0.9485	0.9879	0.2563	0.0368	62000

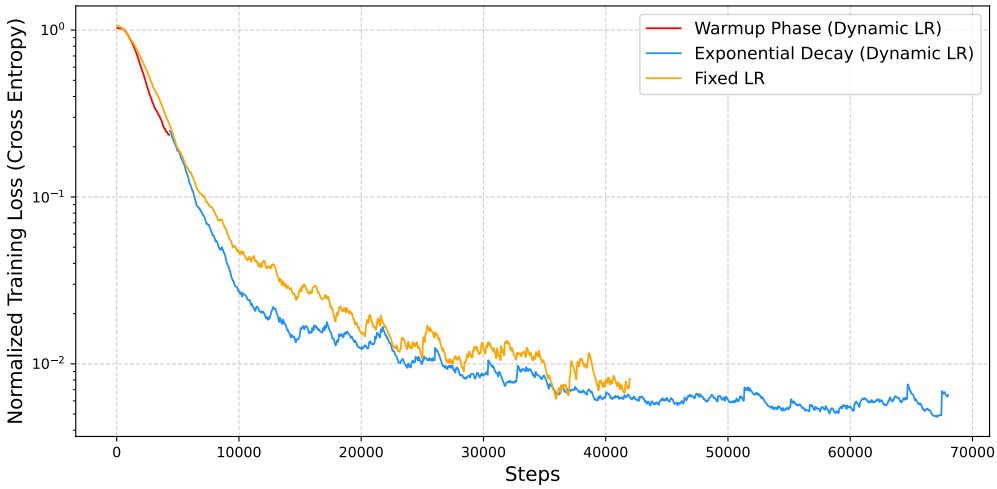
Tab. 5.2: Comparación de resultados al entrenar en una etapa con LR fijo en 10^{-4} y con la programación de LR (Dinámico) de la etapa 1 de la Tabla 5.1 con el upstream congelado y descongelado (*full fine-tuning*).

En cuanto a los resultados, observamos que la programación de LR logra mejoras notables para hacer *full fine-tuning*. Al entrenar solamente el downstream la programación mejora la accuracy en detrimento leve de la cross entropy y una convergencia en el doble de tiempo. En cuanto a la convergencia del *full fine-tuning* los modelos convergen en una menor cantidad de pasos aunque cada paso es más lento (aproximadamente $1{,}88it/s$ contra $2{,}77it/s$) ya que al descongelar el upstream se ajustan más parámetros durante el entrenamiento.

En el siguiente gráfico se puede comparar las distintas evoluciones de la cross entropy en entrenamiento con el LR fijo y con la programación presentada para *full fine-tuning*.



(a) Evolución del LR según los steps.



(b) Función de pérdida suavizada a lo largo del entrenamiento.

Fig. 5.1: Evolución del LR y la cross entropy en batches de la partición de entrenamiento durante el entrenamiento con *full fine-tuning*, correspondiente a los valores de la tabla 5.2.

De estos gráficos se desprende que la programación de LR logra una convergencia más suave con el decrecimiento exponencial. A su vez, con la programación de LR se consigue una función de pérdida consistentemente menor a lo largo del entrenamiento. Consideramos que la anterior visualización es fundamental para ajustar las diferentes estrategias de LR a diferentes contextos y reconocer si hay que ajustar ciertos hiperparámetros. Por ejemplo, sería razonable mantener el *warm-up* hasta notar un incremento en la varianza de la función de pérdida. También, el decrecimiento exponencial del LR fuerza la convergencia al tender a cero a los ajustes en los pesos, por lo que podría considerarse sostener un LR constante (podría ser alrededor del 10^{-4}, que hemos utilizado hasta aquí) a partir de cierto punto antes de continuar su decrecimiento a cero (una tercera parte de la programación con la técnica *Reduce LR On Plateau*). No obstante, nos atendremos a la programación antes descrita y mantenemos como posible trabajo futuro la exploración de técnicas *ad-hoc* y alternativas.

El siguiente paso de la experimentación de la programación de LR es probar un entrenamiento en etapas. La primera etapa corresponde en entrenar el downstream, y las siguientes etapas harán *full fine-tuning* tomando como base al modelo entrenado en la etapa anterior. En cada etapa, se utilizará la configuración correspondiente de la programación de LR presentada anteriormente.

Etapa	Accuracy Top1	Accuracy Top5	Cross Entropy	Normalized Cross Entropy	Steps
1	0.9617	0.9899	0.2186	0.0314	82000
2	**0.9697**	**0.9926**	**0.1775**	**0.0255**	50000
3	0.9697	0.9926	0.1775	0.0255	22000

Tab. 5.3: Resultados del entrenamiento de cada etapa con las programaciones de LR presentadas anteriormente, con los valores de hiperparámetros detallados en la tabla 5.1.

Al analizar los resultados presentados en la tabla, se observa que la ejecución de la etapa 3 no produjo mejoras en el rendimiento del modelo, lo cual sugiere que esta etapa adicional de entrenamiento no aportó valor al proceso de optimización. Las razones de esto pueden ser múltiples: la programación propuesta era para SV (una tarea más compleja que SID), sobre otro dataset más grande (VoxCeleb2), y utilizando otros tamaños de *chunk* al entrenar (nosotros usamos siempre 5s mientras que en el esquema propuesto han variado entre etapas). Por otro lado, vemos que la segunda etapa, en la que se entrena el upstream, se consigue pequeñas ganancias en todas las métricas por el costo de 50000 steps de *full fine-tuning*. De todas maneras, toda ganancia es valiosa y más aún en estos niveles de performance, donde cada vez es más difícil conseguirlas. Nos quedaremos entonces con las primeras dos etapas como el mejor esquema encontrado para realizar fine-tuning en nuestro contexto.

Finalmente, podemos notar que la programación de LR presentada en esta sección es un ejemplo claro de estas configuraciones axiomáticas que se suelen presentar en algunos papers, y mencionamos como motivación en la introducción de este trabajo. Como no tenemos la capacidad de explorar en profundidad todos los valores de hiperparámetros que se presentan, solo nos vemos en capacidad de implementar lo expuesto y probar con los valores dados ajustándolos a nuestro contexto. Los sustentos de estos valores suelen ser que son la adaptación de otro paper en un contexto similar, o bien los sustentos simplemente residen en la intuición de los autores. Más aún, en este caso, el acceso a esta información se encuentra únicamente en el foro del repositorio oficial.

5.2. LoRA

En este experimento, evaluamos reemplazar el full fine-tuning por técnicas PEFT, en particular, LoRA. Específicamente, los módulos seleccionados para la adaptación son las capas de la feed-forward en los bloques de *Transformers*, como fue hecho en Lin et al. (2024) para reconocimiento de emociones en el habla (Speaker Emotion Recognition, SER).

Así, en primer lugar, realizamos una comparación entrenando los modelos con estas técnicas sin haber entrenado al downstream antes, de la misma forma que comenzamos el experimento anterior. En cuanto al LR, utilizaremos la configuración etapa 1 del cronograma expuesto en el anterior experimento. Para la configuración de los métodos PEFT, siguiendo los valores predeterminados en modelos similares adoptamos $r = 16$ y $\alpha = 32$, esta proporción entre ambos ($2r = \alpha$) ha mostrado obtener el mejor desempeño (Biderman

et al., 2024). También, por defecto, se suele utilizar Dropout con probabilidad 0,05 sobre la entrada de las matrices de LoRA. Con rango 16, en WavLM Base+ se está aproximando matrices de pesos de $768x3072$ con matrices de $768X16$ y $16x3072$, es decir, ajustes de 2.36M de parámetros aproximados con 61440 parámetros. Con esta dimensionalidad es posible que no se logre capturar la suficiente complejidad de los ajustes, es por eso que también experimentamos con otra configuración con la que se han visto buenos resultados (Biderman et al., 2024; Raschka, 2023): $r = 256$ y $\alpha = 512$. También, sumamos $r = 64, \alpha = 128$ como configuración intermedia ya que $r = 256$ aumenta la cantidad de parámetros considerablemente.

PEFT	Parámetros entrenados	Acc Top1	Acc Top5	CE	NCE	Steps
✗	23.6M (downstream)	0.9617	0.9899	0.2186	0.0314	82000
✗	118M (upstream + downstream)	0.9485	0.9879	0.2563	0.0368	62000
LoRA ($r = 16, \alpha = 32$)	25.1 M	0.9598	0.9910	0.2265	0.0325	60000
LoRA ($r = 64, \alpha = 128$)	29.5 M	0.9674	0.9931	0.2011	0.0288	**56000**
LoRA ($r = 256, \alpha = 512$)	47.2 M	**0.9690**	**0.9944**	**0.1914**	**0.0274**	68000

Tab. 5.4: Resultados al usar PEFT para entrenar en una única etapa. La cruz en la columna PEFT hace referencia a la no utilización de técnicas PEFT.

Los resultados muestran que la implementación de LoRA en estos módulos mantiene un desempeño consistentemente por encima del full fine-tuning. Más aún, con $r = 64$ y $r = 256$ la utilización de LoRA logra superar al rendimiento del modelo en que solo se entrenó al downstream. En términos de eficiencia computacional, la cantidad de parámetros entrenados impacta directamente en el consumo de recursos de GPU durante el entrenamiento. Mientras el full-finetuning incorpora 94.4M de parámetros entrenables, LoRA con $r = 16$ incorpora 1.5M y con $r = 256$, 22.6M. Con $r = 64$ se incorporan 5.9M de parámetros y se obtienen resultados intermedios, por lo que también son una opción a considerar. Esta reducción significativa en el uso de recursos computacionales representa una ventaja importante de los métodos PEFT.

Por lo tanto, hemos visto que una configuración de LoRA logra mejorar la primera etapa de entrenamiento, por lo que cabe la duda de cuál es la mejor estrategia de entrenamiento al usar esta técnica. El siguiente paso, siguiendo los resultados del experimento 5.1, será entrenar partiendo del downstream entrenado reemplazando el full fine-tuning por el método PEFT.

PEFT	Parámetros entrenados	Acc Top1	Acc Top5	CE	NCE	Steps
✗	118M	**0.9697**	0.9926	0.1775	0.0255	50000
LoRA ($r = 16, \alpha = 32$)	25.1 M	0.9667	0.9924	0.1680	0.0241	68000
LoRA ($r = 64, \alpha = 128$)	29.5 M	0.9676	**0.9930**	0.1632	0.0234	74000
LoRA ($r = 256, \alpha = 512$)	47.2 M	0.9678	**0.9930**	**0.1608**	**0.0231**	**44000**

Tab. 5.5: Comparación de resultados al utilizar métodos PEFT en la segunda etapa de entrenamiento, es decir, partiendo del mejor downstream entrenado.

En este caso, notamos que los métodos PEFT en sus diferentes configuraciones no mejoran el desempeño en accuracy del full fine-tuning pero sí logran una menor cross entropy, que es la métrica que buscamos optimizar con nuestra configuración de entrenamiento. Además, con $r = 16$, en este contexto sí se consigue también mejorar el desempeño con respecto a solo entrenar el modelo downstream. Por lo tanto, esta configuración se

presenta como una alternativa atractiva para disminuir la cross entropy y mejorar ligeramente la accuracy con una cantidad mínima de parámetros adicionales. Por su parte, con $r = 256$ entrenando en esta segunda etapa se alcanza una mejora notable en cross entropy, pero una accuracy menor a entrenar en una única etapa. A su vez, con $r = 64$ se alcanza una accuracy equivalente, por lo que la elección de 256 como rango solo se sostiene en búsqueda de la menor cross entropy.

En conclusión, consideramos que la utilización de LoRA en este contexto no solo es una alternativa atractiva por su menor cantidad de parámetros sino que es preferible para tener una menor cross entropy. La eficiencia computacional de este método no solo posibilita un entrenamiento más veloz sino que además permite utilizar batches más grandes al ocupar menos memoria.

6. CONCLUSIONES Y TRABAJO FUTURO

En este trabajo, exploramos distintas técnicas, hiperparámetros y arquitecturas para realizar Transfer Learning en SID y llegamos a resultados estados del arte. Los resultados compilados en la experimentación son el reflejo de más de 3233 horas de tiempo de ejecución, más de cuatro meses y medio. Además, si bien ha quedado implícito por no ser el foco del trabajo, este proceso nos dejó aprendizajes y un gran desarrollo técnico en la implementación del código necesario para realizar cada experimento.

En el capítulo 4 nos enfocamos en optimizar el modelo downstream, logrando con un modelo upstream pequeño (WavLM Base+) alcanzar mejores resultados que modelos con una cantidad de parámetros tres veces superior. Comenzamos la experimentación con los hiperparámetros más esenciales de una red neuronal, con los que pudimos profundizar en los fundamentos detrás de cada decisión de diseño. Identificamos como factores determinantes para el desempeño del modelo al LR y la utilización de atención en los mecanismos de pooling. Este último aspecto requirió una exploración detallada de hiperparámetros, dado que los resultados presentaron alta varianza. Aunque inicialmente la atención como método de pooling produjo resultados inferiores al promedio, una afinada exploración permitió obtener mejoras significativas en el desempeño. En particular, el factor que más marcó la diferencia fue la cantidad de cabezas de atención en combinación con el LR.

En cuanto a la convergencia, en reiteradas ocasiones observamos que modelos más complejos convergen más tempranamente, y si bien muchas veces alcanzan un mejor desempeño, también son más propensos a sobre-ajustar a los datos de entrenamiento. A su vez, logramos reducir hasta la mitad el tiempo de entrenamiento con la utilización de técnicas de normalización Z. Otro factor fundamental en los tiempos de ejecución y que nos permitió poder explorar más hiperparámetros fue pre-calcular los embeddings del upstream.

Por otro lado, entre los elementos que no mostraron buenos resultados se encuentran la utilización de diferentes estadísticas concatenadas en reemplazo del promedio como pooling temporal, y el uso del *Positional Encoding* convencional en el pooling temporal con atención. Las razones por las que obtuvimos estos resultados en la concatenación de distintas estadísticas creemos que podrían radicar en que la distribución resultante presenta valores atípicos que distorsionan la representación, o simplemente, estas estadísticas no aportan información adicional usando los embeddings de WavLM Base+. En cuanto al *Positional Encoding*, una posible explicación para su bajo rendimiento podría estar relacionada con la escala de los valores generados. Es posible que el *encoding* introduzca magnitudes que dominen los valores originales de los *embeddings*, provocando una pérdida de información relevante para la tarea. Otra explicación posible es que la tarea de identificación de hablante no se beneficia de un entendimiento del orden temporal, ya que las características que determinan la identidad podrían estar presentes en toda la señal de audio y ser locales.

Tras explorar los distintos hiperparámetros y aprender qué combinaciones llevaban a un mejor desempeño, arribamos a un modelo downstream que utiliza transformers tanto para pooling temporal como de capas y que alcanzó los mejores resultados: 0,9474 de accuracy y 0,2107 de cross entropy.

Luego, en el capítulo 5 incursionamos en el fine-tuning del modelo upstream. Así, en-

contramos que entrenar a todo el modelo no es una buena estrategia por sí sola, y que requiere un mayor cuidado en el ajuste de los pesos. Obtuvimos ligeras mejoras en el desempeño con un entrenamiento por etapas con diferentes programaciones dinámicas de LR. A su vez, nos adentramos en métodos PEFT experimentando con LoRA y entendiendo su funcionamiento e hiperparámetros. De esta manera, arribamos a una cross entropy mínima de 0,1608 entrenando primero el downstream y luego entrenando también el upstream con LoRA. La accuracy máxima (0,9697) fue alcanzada haciendo full fine-tuning en la segunda etapa. Como conclusión de la sección, notamos que no siempre modelos con mayor cantidad de parámetros llegan a mejores resultados sino que a mayor cantidad de parámetros se requiere un mayor cuidado para ajustar el LR e integrar el modelo upstream, el cual ya se encuentra pre-entrenado, con el modelo downstream, que posee pesos aleatorios. Esta diferencia en el conocimiento adquirido por las distintas partes de la red, puede llevar a que el upstream se modifique demasiado durante el finetuning, compensando la falta de entrenamiento del downstream, y olvidando su conocimiento previo llevando a un sobreajuste de los datos.

Por último, a lo largo de la investigación se fueron sembrando múltiples interrogantes que quedaron fuera del alcance de este trabajo, pero que representan direcciones prometedoras para futuras investigaciones. Entre las principales cuestiones destacamos las siguientes:

- **Exploración en otras tareas y datos**: Experimentar con el modelo downstream hallado en distintas tareas y conjuntos de datos. ¿Son consistentes nuestros hallazgos en otros contextos y dominios?

- **Evaluación de múltiples modelos upstream**: Analizar si modelos más complejos, como WavLM Large, pueden aprovechar mejor las ventajas del *full fine-tuning* y explorar si estas capacidades se traducen en un mejor desempeño en tareas específicas.

- **Técnicas de *full fine-tuning***: Experimentar con diversas técnicas de finetuning y full-finetuning, entre ellas: (1) entrenar únicamente la última capa, (2) la técnica de *gradual unfreezing*, (3) diferentes programaciones de LR para cada capa .

- **Profundización en la programación del LR**: Si bien hay trabajo precedente (Wu et al., 2019b), todavía queda un vasto espacio para comprender a fondo las programaciones de LR y cómo determinar estrategias óptimas. Una pregunta específica en la misma línea es cómo interactúan las cabezas de atención con el LR.

- **Relación entre la complejidad del modelo downstream y el *domain-shift***: Al haber incorporado mayor complejidad en el downstream y alcanzado resultados que han sido mejorados solo levemente por el finetuning nos surge la siguiente pregunta: ¿Un mejor downstream hace que el upstream tenga menor *domain-shift*? ¿Cómo se compara con técnicas PEFT?

- **Medición de la importancia de los hiperparámetros**: Diseñar una métrica para evaluar la relevancia de cada hiperparámetro. Esto permitiría asignar los recursos de manera más eficiente al focalizar la exploración de hiperparámetros en los más influyentes para el desempeño final.

- **Adapters vs Downstream**: ¿Es mejor incorporar nuevos pesos dentro del modelo upstream, como hace LoRA, o más pesos en el downstream? En nuestro downstream final utilizamos distintos mecanismos de atención, ¿Cómo se compara adaptar módulos de atención con agregar mecanismos de atención en el downstream?

A pesar de las múltiples direcciones de investigación que quedan abiertas, consideramos que este trabajo nos ha proporcionado una comprensión más profunda de los mecanismos de Transfer Learning aplicados a SID. Hemos logrado establecer una metodología sistemática para la optimización de modelos downstream y hemos demostrado que modelos relativamente pequeños, cuidadosamente optimizados, pueden competir e incluso superar a arquitecturas mucho más complejas. Los hallazgos sobre la interacción entre mecanismos de atención, estrategias de pooling y programaciones de LR no solo contribuyen al avance del estado del arte en identificación de hablantes, sino que también ofrecen valiosas perspectivas para el diseño de soluciones más eficientes en tareas de procesamiento de audio en general.

Más allá de los resultados numéricos, este proyecto fue una oportunidad invaluable para profundizar en los fundamentos del aprendizaje profundo y adquirir experiencia práctica en la implementación y optimización de modelos complejos en PyTorch. En definitiva, este trabajo demuestra que el éxito en Transfer Learning no siempre radica en utilizar modelos más grandes, sino de comprender a fondo cómo aprovechar y adaptar eficientemente las representaciones pre-entrenadas para tareas específicas.

Bibliografía

Ossama Abdel-Hamid, Li Deng, and Dong Yu. 2013. Exploring convolutional neural network structures and optimization techniques for speech recognition. In *Interspeech 2013*, pages 3366–3370.

Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. 2016. Layer normalization. *ArXiv*, abs/1607.06450.

Alexei Baevski, Steffen Schneider, and Michael Auli. 2019. vq-wav2vec: Self-supervised learning of discrete speech representations. *ArXiv*, abs/1910.05453.

Alexei Baevski, Henry Zhou, Abdelrahman Mohamed, and Michael Auli. 2020. wav2vec 2.0: a framework for self-supervised learning of speech representations. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA. Curran Associates Inc.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Dan Biderman, Jacob Portes, Jose Javier Gonzalez Ortiz, Mansheej Paul, Philip Greengard, Connor Jennings, Daniel King, Sam Havens, Vitaliy Chiley, Jonathan Frankle, Cody Blakeney, and John Patrick Cunningham. 2024. LoRA learns less and forgets less. *Transactions on Machine Learning Research*. Featured Certification.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Sanyuan Chen, Chengyi Wang, Zhengyang Chen, Yu Wu, Shujie Liu, Zhuo Chen, Jinyu Li, Naoyuki Kanda, Takuya Yoshioka, Xiong Xiao, Jian Wu, Long Zhou, Shuo Ren, Yanmin Qian, Yao Qian, Micheal Zeng, and Furu Wei. 2021. Wavlm: Large-scale self-supervised pre-training for full stack speech processing. *IEEE Journal of Selected Topics in Signal Processing*, 16:1505–1518.

Zewen Chi, Shaohan Huang, Li Dong, Shuming Ma, Bo Zheng, Saksham Singhal, Payal Bajaj, Xia Song, Xian-Ling Mao, Heyan Huang, and Furu Wei. 2022. XLM-E: Cross-lingual language model pre-training via ELECTRA. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6170–6182, Dublin, Ireland. Association for Computational Linguistics.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar. Association for Computational Linguistics.

Kevin Clark, Urvashi Khandelwal, Omer Levy, and Christopher D. Manning. 2019. What does BERT look at? an analysis of BERT's attention. In *Proceedings of the 2019 ACL Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 276–286, Florence, Italy. Association for Computational Linguistics.

George V. Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:303–314.

Andrew M. Dai and Quoc V. Le. 2015. Semi-supervised sequence learning. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, page 3079–3087, Cambridge, MA, USA. MIT Press.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In *North American Chapter of the Association for Computational Linguistics*.

Luciana Ferrer. 2022. Analysis and comparison of classification metrics. *ArXiv*, abs/2209.05355.

Daniel Garcia-Romero and Carol Y. Espy-Wilson. 2011. Analysis of i-vector length normalization in speaker recognition systems. In *Interspeech 2011*, pages 249–252.

Varun Godbole, George E. Dahl, Justin Gilmer, Christopher J. Shallue, and Zachary Nado. 2023. Deep learning tuning playbook. Version 1.0.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *CoRR*, abs/1706.02677.

Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech recognition with deep recurrent neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6645–6649.

Geoffrey E. Hinton, Simon Osindero, and Yee Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural Computation*, 18:1527–1554.

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *CoRR*, abs/1207.0580. Cite arxiv:1207.0580.

Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. Parameter-efficient transfer learning for NLP. In *Proceedings of the 36th International Conference on Machine Learning*.

Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. In *Annual Meeting of the Association for Computational Linguistics*.

Wei-Ning Hsu, Benjamin Bolte, Yao-Hung Hubert Tsai, Kushal Lakhotia, Ruslan Salakhutdinov, and Abdel rahman Mohamed. 2021. Hubert: Self-supervised speech representation learning by masked prediction of hidden units. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 29:3451–3460.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani, and Jonathan Taylor. 2023. *An Introduction to Statistical Learning with Applications in Python*. Springer Texts in Statistics. Springer, Cham.

Dayal Singh Kalra and Maissam Barkeshli. 2024. Why warmup the learning rate? underlying mechanisms and improvements. *arXiv preprint arXiv:2406.09405*.

Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Ananya Kumar, Aditi Raghunathan, Robbie Matthew Jones, Tengyu Ma, and Percy Liang. 2022. Fine-tuning can distort pretrained features and underperform out-of-distribution. In *International Conference on Learning Representations*.

Yann Lecun and Francoise Soulie Fogelman. 1987. Modeles connexionnistes de l'apprentissage. *Intellectica, special issue apprentissage et machine*, 2.

Tzu-Han Lin, How-Shing Wang, Hao-Yung Weng, Kuang-Chen Peng, Zih-Ching Chen, and Hung-yi Lee. 2024. Peft for speech: Unveiling optimal placement, merging strategies, and ensemble techniques. *arXiv preprint arXiv:2401.02122*.

Tomas Mikolov, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. In *International Conference on Learning Representations*.

Seyedmahdad Mirsamadi, Emad Barsoum, and Cha Zhang. 2017. Automatic speech emotion recognition using recurrent neural networks with local attention. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2227–2231.

Tom Mitchell. 1997. *Machine Learning*. McGraw-Hill Education.

Abdelrahman Mohamed, Hung-yi Lee, Lasse Borgholt, Jakob D. Havtorn, Joakim Edin, Christian Igel, Katrin Kirchhoff, Shang-Wen Li, Karen Livescu, Lars Maaløe, Tara N. Sainath, and Shinji Watanabe. 2022. Self-supervised speech representation learning: A review. *IEEE Journal of Selected Topics in Signal Processing*, 16(6):1179–1210.

Arsha Nagrani, Joon Son Chung, and Andrew Zisserman. 2017. Voxceleb: A large-scale speaker identification dataset. In *Interspeech*.

Titouan Parcollet, Rogier van Dalen, Shucong Zhang, and Sourav Bhattacharya. 2024. Summarymixing: A linear-complexity alternative to self-attention for speech recognition and understanding. In *Interspeech 2024*, pages 3460–3464.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing*.

Boris Polyak. 1964. Some methods of speeding up the convergence of iteration methods. *Ussr Computational Mathematics and Mathematical Physics*, 4:1–17.

Alec Radford and Karthik Narasimhan. 2018. Improving language understanding by generative pre-training.

Sebastian Raschka. 2023. Practical tips for finetuning llms using lora (low-rank adaptation).

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. 1986. Learning representations by back-propagating errors. *Nature*, 323:533–536.

Steffen Schneider, Alexei Baevski, Ronan Collobert, and Michael Auli. 2019. wav2vec: Unsupervised pre-training for speech recognition. In *Interspeech*.

David Snyder, Daniel Garcia-Romero, and Daniel Povey. 2015. Time delay deep neural network-based universal background models for speaker recognition. In *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pages 92–97.

David Snyder, Daniel Garcia-Romero, Gregory Sell, Daniel Povey, and Sanjeev Khudanpur. 2018. X-vectors: Robust dnn embeddings for speaker recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5329–5333.

Ilya Sutskever, James Martens, George E. Dahl, and Geoffrey E. Hinton. 2013. On the importance of initialization and momentum in deep learning. In *International Conference on Machine Learning*.

Wilson L. Taylor. 1953. "cloze procedure": A new tool for measuring readability. *Journalism & Mass Communication Quarterly*, 30:415 – 433.

Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Neural Information Processing Systems*.

A. Waibel, T. Hanazawa, G. Hinton, K. Shikano, and K.J. Lang. 1989. Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(3):328–339.

Yanzhao Wu, Ling Liu, Juhyun Bae, Ka-Ho Chow, Arun Iyengar, Calton Pu, Wenqi Wei, Lei Yu, and Qi Zhang. 2019a. Demystifying learning rate policies for high accuracy training of deep neural networks. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1971–1980.

Yanzhao Wu, Ling Liu, Juhyun Bae, Ka-Ho Chow, Arun Iyengar, Calton Pu, Wenqi Wei, Lei Yu, and Qi Zhang. 2019b. Demystifying learning rate policies for high accuracy training of deep neural networks. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 1971–1980.

Shu-wen Yang, Po-Han Chi, Yung-Sung Chuang, Cheng-I Jeff Lai, Kushal Lakhotia, Yist Y Lin, Andy T Liu, Jiatong Shi, Xuankai Chang, Guan-Ting Lin, et al. 2021. Superb: Speech processing universal performance benchmark. *arXiv preprint arXiv:2105.01051*.

Kaichao You, Mingsheng Long, Jianmin Wang, and Michael I. Jordan. 2019. How does learning rate decay help modern neural networks. *arXiv: Learning*.