Proyecto final:

Generación de LaTeX a partir de imágenes de fórmulas

Curso 2020/21

Visión por computador

José Antonio Álvarez Ocete,
Daniel Pozo Escalona

joseantonioao32@correo.ugr.es,
danipozo@correo.ugr.es

Índice

1. Introducción

Nuestro objetivo en este trabajo será desarrollar un modelo que, dada una imagen producida al compilar código LaTeX asociado a una fórmula, prediga el código LaTeX que fue utilizado para compilarla. La aplicación práctica de un modelo de estas características permitiría agilizar procesos de digitalización de documentos matemáticos, o la redacción de los mismos a partir de fuentes de cuyo código original no se dispone.

Para ello utilizaremos un esquema codificador-decodificador basado en *transformers* [9].

2. Bases de datos

Las bases de datos para el problema son:

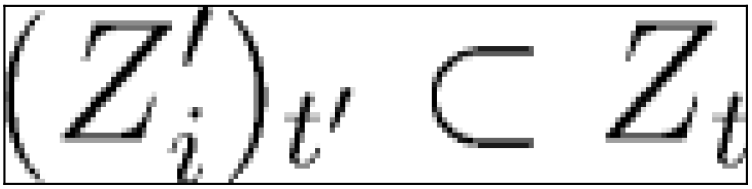
- IM2LATEX-100K [3], y
- IMLATEX-170K [1]: contiene 65000 ejemplos, que se añaden a los 100000 de im2latex-100k.

Una muestra de IMLATEX-170K se puede ver en la figura 1.

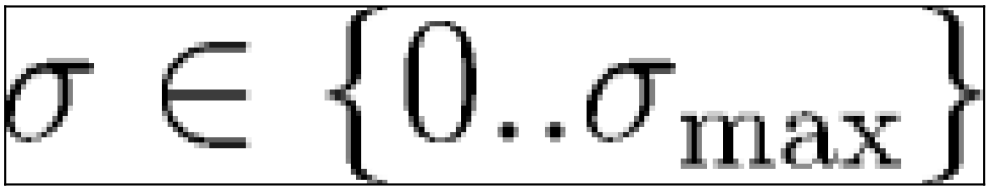
Ambas bases de datos contienen ambigüedad en los ejemplos, en forma de fórmulas o segmentos de fórmulas que, escritos de distinta forma, producen la misma imagen. Dedicaremos parte del tiempo del proyecto a diseñar formas de normalizar estos datos, para mejorar los resultados. Entraremos en detalle más adelante.

Debido a esta ambigüedad, las evaluaciones iniciales con modelos de referencia y las finales pueden no ser directamente comparables. Esto lo paliaremos de dos formas: con la base de datos sintética que introducimos a continuación, y evaluando los modelos de referencia en la base de datos normalizada. Además, hay que tener en cuenta que lo relevante en este problema es dar expresiones LaTeX que produzcan las imágenes que se proporcionen al modelo, más que estas coincidan con unas prefijadas.

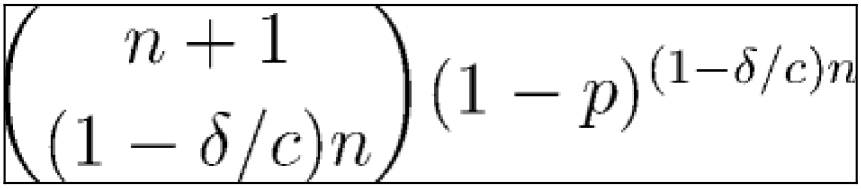
En la práctica, utilizaremos únicamente IM2LATEX-170K, ya que contiene a la anterior.



(Z _ { i } ^ { \prime }) _ { t ^ { \prime } } \subset Z _ { t }



\sigma \in \{ 0 . . \sigma _ { \operatorname* { m a x } } \}



\binom { n + 1 } { (1 - \delta / c) n } (1 - p) ^ { (1 - \delta / c) n }

Figura 1: Tres muestras de im2latex-170k.

2.1. Base de datos sintética: Toy

Lo primero que hemos hecho ha sido crear una base de datos similar a las que pretendemos tratar, en la que las fórmulas han sido generadas a partir de una gramática relativamente sencilla, que contiene una cantidad pequeña de símbolos. A esta base de datos la hemos denominado **Toy**.

Hemos generado 50.000 ejemplos distintos. Los guiones Python que hemos escrito a tal efecto permiten cambiar la gramática y generar conjuntos de datos con cantidades arbitrarias de ejemplos únicos.

Además, esta base de datos puede ser útil para evaluar cambios a la arquitectura o hiperparámetros de forma menos costosa.

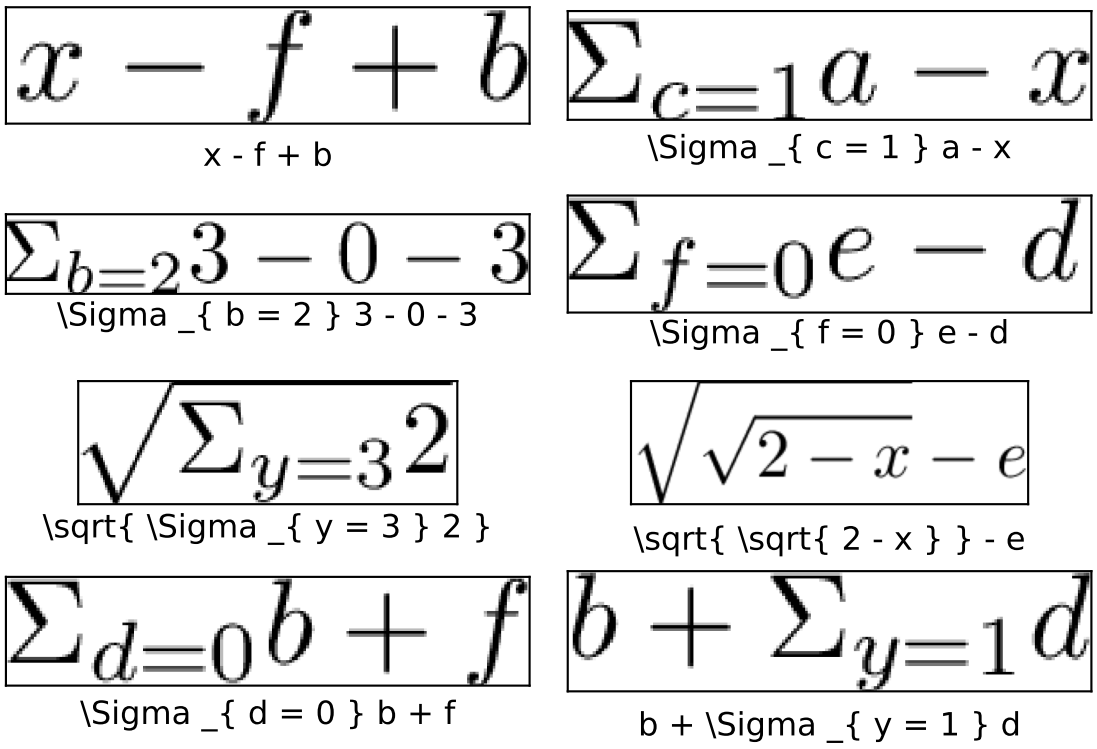


Figura 2: Muestras de la base de datos sintética.

3. Modelo propuesto

El modelo que proponemos es el siguiente:

- Un codificador, formado por una red convolucional, seguido por un modelo *transformer*.
- Un decodificador *transformer*, que emplea atención multi-cabezal sobre la salida del codificador y sobre una secuencia potencialmente incompleta, para predecir el siguiente token de la fórmula. Es decir, es un modelo de lenguaje condicional que modela $p(x_t|V, x_1, ..., x_{t-1})$, donde V es la salida del codificador.

Utilizaremos una red convolucional para obtener las características de las imágenes de entrada. Este vector de características será la entrada para el modelo codificador-decodificador.

Aunque posteriormente el modelo de referencia que se presenta a continuación sufrirá ciertas variaciones, el esquema general no variará. Este modelo posee un codificador convolucional con tres capas Conv-BN-ELU-MaxPooling y una sola capa de atención en el codificador y en el decodificador.

Codificador convolucional

Capa/bloque
Conv2D-BN-ELU(64 filtros)
MaxPooling2D(2×2)
Conv2D-BN-ELU(64 filtros)
MaxPooling2D(2×2)
Conv2D-BN-ELU(64 filtros)
MaxPooling2D(2×2)

Codificador *transformer*

Capa/bloque
Codif. conv.
Codificación de posición
Dropout
Capa de codificador

Decodificador *transformer*

Capa/bloque
Embedding
Codificación de posición
Dropout
Capa de decodificador

Figura 3: Bloques del modelo de referencia.

4. Detalles técnicos y de implementación

En esta sección detallamos algunas cuestiones que pueden tener relevancia de cara a reproducir el trabajo realizado y entender las dificultades técnicas que hemos afrontado.

4.1. *Hardware* empleado

Para los experimentos, hemos empleado máquinas con las siguientes características:

- 30 GB RAM,
- procesadores de 8 núcleos, modelo no especificado,
- GPU Nvidia Quadro P5000 o P6000.

4.2. Arquitectura y configuración de experimentos

El código de los experimentos ha sido concebido de la siguiente forma: como un guion Python principal, que se encarga de leer los datos, configurar el modelo, entrenarlo y guardar los resultados. La configuración del modelo depende de una serie de banderas de línea de órdenes que el programa interpreta. Esto nos ha permitido lanzar múltiples experimentos en paralelo, sin necesidad de modificar el código. El código se encuentra disponible en `https://github.com/Ocete/LaTeXrec`.

4.3. Preprocesamiento de datos

Para las imágenes, el procesamiento que realizamos es redimensionarlas a una altura común, manteniendo la relación de aspecto. Tras esto, eliminamos los ejemplos con imágenes demasiado grandes para ser procesadas por el modelo, debido al coste cuadrático del mecanismo de atención.

Para las fórmulas, utilizamos el *tokenizer* de TensorFlow [8] para obtener el alfabeto de entrada. Por lo tanto, este no es fijo sino que depende del conjunto de datos. Esto nos permite codificar las fórmulas numéricamente.

4.4. Guardado de modelos

TensorFlow proporciona algunas utilidades para guardar modelos y cargarlos posteriormente, lo que permite entrenarlos en varias fases, guardar los modelos en varios puntos del entrenamiento o usarlos para obtener predicciones después de entrenados.

Sin embargo, no hemos conseguido que funcionen con nuestro modelo, que es relativamente complejo, en el sentido de que ha sido creado mediante *model subclassing* y toma varios argumentos que no son entradas (las máscaras) al ser llamado.

Esto es un obstáculo que habría de ser solventado de cara a una posible aplicación del modelo.

4.5. Lectura eficiente

Una vez comenzamos a trabajar con la base de datos IM2LATEX-170K descubrimos que esta no cabía por completo en memoria. De hecho, no cabía ni una cuarta parte. De cara a implementar una solución escalable y eficiente a este problema creamos una clase, `LaTeXrecDataset`, que hereda de `tf.data.Dataset` [7], de TensorFlow. Esta clase nos permite leer las imágenes conforme el modelo las necesita y, al mismo tiempo, eliminarlas de memoria cuando dejan de hacerlo. Todo esto se procesa de forma automática y muy cómodamente, además de permitirnos implementar *prefetching*. Esto es, lectura anticipada de los datos antes de que hagan falta para acelerar el proceso.

4.6. *Early Stopping*

Hemos implementado *early stopping* con el objetivo de reducir el tiempo de cómputo de algunos experimentos cuando se estanca su evolución. Esta implementación ha sido manual ya que el entrenamiento de los modelos se realiza paso a paso manualmente y no podemos utilizar el *callbacks* de Tensorflow. Dicho *early stopping* esta basado la precisión en validación, no en la pérdida.

4.7. Registros (*logs*)

Añadimos un sistema de *logging* a distintos archivos y a la salida estándar, para facilitar la depuración de errores y el seguimiento del progreso del programa.

5. Características añadidas al modelo

En esta sección describimos las distintas características implementadas sobre el modelo inicial descrito con anterioridad. Los resultados de las mismas se describen en la sección de experimentos.

5.1. Eliminación de ambigüedades

Como ya discutimos en la sección 2, en este problema encontramos ambigüedad en la salida de los datos. Distintas expresiones LaTeX generan la misma salida o una casi indistinguible en la imagen. Por ejemplo, $\backslash sin$ y $\backslash operatorname\{sin\}$: $\sin(x)$ frente a $\sin(x)$.

Esto hace que el aprendizaje sea más difícil. El primer cambio que hicimos tras tener un modelo funcional fue eliminar este tipo de ambigüedades en los datos. Cabe aclarar que aunque estemos alterando los datos, la imagen LaTeX obtenida es exactamente la misma en todos estos casos.

Utilizamos el *tokenizer* de TensorFlow [8] para generar una lista de tokens de las primeras 60.000 imágenes en la que basarnos. Estudiando esta lista hemos decidido modificar las siguientes ambigüedades, buscando un balance entre implementación sencilla y que merezca la pena porque sean lo suficientemente relevantes:

- Los operadores con nombre más famosos (como seno y máximo), tienen un comando particular en LaTeX: $\backslash sin$, y $\backslash max$. Utilizaremos estos comandos en vez de los respectivo $\backslash operatorname\{sin\}$ y

\operatorname{max}, siempre que se pueda. Aplicaremos esta misma sustitución para los comandos con un asterisco: \operatorname* {sin}. La lista completa de operadores a la que le aplicamos essta sustitución es la siguiente: '\sin, \cos, \tan, \arcsin, \arccos, \arctan, \sinh, \cosh, \tanh, \max, \min, \exp, \log, \ln, \sup, \inf, \lim, \dim, \deg, \ker, \cot, \Pr, \lg, \arg, \det, \vol.

- El comando \prime se utiliza en LaTeX para mostrar una comilla grande. En caso de utilizarse como exponente (\^{ \prime }) tiene la misma representación gráfica que una comilla simple: '. Sustituiremos la expresión: ' \^{ \prime }' por '.
- Los símbolos de llaves '{' y '}' se pueden escribir también como ' \brace ' y ' \rbrace ' respectivamente. Los sustituiremos por la versión más corta que es, además, más general ya que se puede utilizar tanto en modo texto como en modo matemáticas.
- El símbolo de la daga se puede escribir en *LaTex* utilizando tanto ' \dagger ' como ' \dag '. Aunque este símbolo apenas aparece en nuestras fórmulas, añadir esta sustitución es una línea extra que no añade complejidad ninguna. Utilizaremos su versión más corta.
- Finalmente, la tipografía aplicada al utilizar '\cal' y '\mathcal' es exactamente la misma aunque su sintaxis es distinta. '\cal' se utiliza para palabras o letras sueltas: '\prime A'; mientras que '\mathcal' se utiliza para expresiones más complejas: '\mathcal { sin ^{ 2 } (x) }'. Puesto que el uso de '\cal' está obsoleto y '\mathcal' es más general, utilizaremos este último, reajustando la sintaxis conforme sea necesario.

5.2. Atención eficiente

Uno de los principales escollos que hemos encontrado ya desde el trabajo preliminar en este proyecto es el enorme consumo de memoria del modelo *transformer*. Esto se debe a que, en cada capa de atención, se ha de calcular la matriz de atención, de tamaño cuadrático en la longitud de la secuencia procesada. En el caso de las características provenientes de la imagen, esta secuencia puede alcanzar longitud 500.

Recientemente, se han propuesto múltiples alternativas para aproximar el mecanismo de atención de forma más eficiente [6]. En la tabla 1 de este artículo de revisión se encuentran listadas todas las alternativas, junto con algunas características de las mismas.

Hemos seleccionado, de entre las que permiten emplear un decodificador, la atención rápida mediante el mecanismo FAVOR+ [2]. Hemos adaptado la implementación de los autores (https://github.com/google-research/google-research/tree/master/performer/fast_attention/tensorflow) y la hemos probado en algunos experimentos.

Esta técnica se basa en emplear características aleatorias de Fourier [4] para aproximar la función Softmax, de forma que en la ecuación de la atención

$$\text{Attn}(Q, K, V) = \text{Softmax}(QK^T)V$$

se pueden reemplazar las matrices Q y K por matrices Q' y K' con una dimensión independiente del tamaño de la secuencia a procesar, y tales que $Q'K'^T$ aproxima a $\text{Softmax}(QK^T)$. Así se pueden reordenar los cálculos, evitando la complejidad cuadrática.

5.3. Codificación posicional 2-dimensional

La capa de *self attention*, en la que se basan los bloques *transformer*, sigue un modelo de conexión "todos con todos", por lo que pierde la información posicional de los elementos que recibe. Es por ello que se añade codificación posicional, para añadir artificialmente dicha información de la posición de cada palabra.

Sin embargo, la codificación posicional usual está pensada para dar información acerca de las posiciones relativas de distintos elementos en una secuencia plana, como puede ser un texto. En el caso de imágenes, esto supone una pérdida de información.

Para paliar esta pérdida de información hemos implementado codificación posicional 2-dimensional, utilizando así la información relativa de las características al completo.

En nuestro caso particular, añadimos esa codificación posicional a la salida de nuestra red convolucional con las características de la imagen obtenidas.

La implementación de la codificación posicional 2-dimensional ha sido traducida a partir de la disponible en este fichero.

5.4. Métrica BLEU

Hacia el final del desarrollo hemos implementado la métrica BLEU para evaluar de forma más intuitiva los resultados obtenidos. Esta métrica nos da una estimación de cuanto se parecen dos cadenas de tokens. Suele utilizarse para comparar cuán buena es una traducción de lenguaje a otro, y la utilizaremos en el último experimento.

6. Parámetros del modelo

El modelo consta de los siguientes parámetros:

- Profundidad: se refiere a la dimensión del *embedding* que emplea el modelo para trabajar con las secuencias de entrada.
- Número de capas del codificador y el decodificador.
- Número de unidades de la red aplicada punto a punto, un componente interno de la arquitectura *transformer*.
- Número de cabezales de atención: la atención implementada es atención multicabezal *estrecha*, por lo que hay que especificar el número de cabezales, que ha de dividir a la profundidad del modelo.
- Tasa de *dropout*: la arquitectura emplea *dropout*, lo que obliga a especificar la tasa con que se aplica el mismo.
- Número de filtros de la red convolucional del codificador.

7. Experimentos

7.1. Metodología

Realizaremos experimentos sobre las dos bases de datos de las que disponemos: la sintética e IM2LATEX-170K. Para la sintética, los experimentos tienen el objetivo principal de comprobar si algunas de las características implementadas tienen verdadera utilidad o no. Para comprobar su efectividad nos bastará con dividir el conjunto de datos en entrenamiento y validación con una proporción del 90 % / 10 %.

Sin embargo, con la base de datos IM2LATEX-170K buscamos también poner a prueba el modelo. Para ello dividiremos el modelo en entrenamiento frente a test con proporción 90 % / 10 %. Este subconjunto de test es apartado antes de realizar ningún experimento.

A continuación para cada experimento el conjunto de entrenamiento se baraja y se vuelve a dividir en entrenamiento y validación con proporción 90 % / 10 %. De esta forma, el subconjunto de validación varía aleatoriamente entre experimentos.

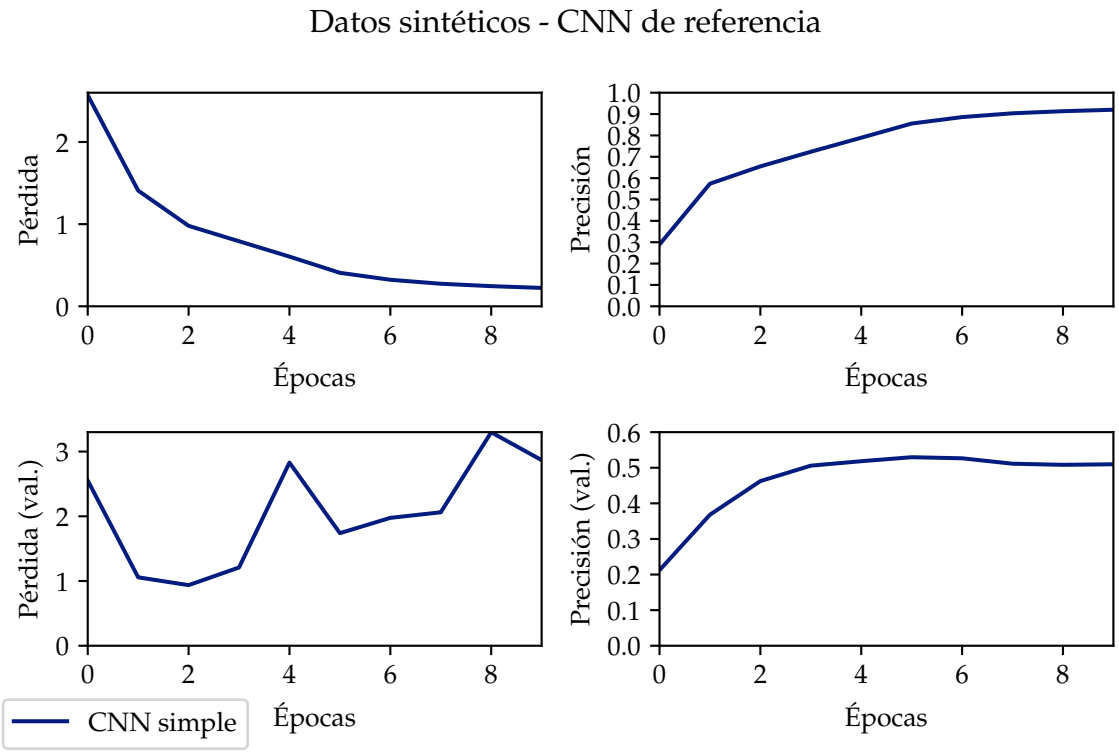
7.2. Experimentos sobre los datos sintéticos

En esta primera sección detallamos algunos de los experimentos más interesantes realizados sobre la base de datos sintética 2.1. Vamos a destacar 3: uno inicial sobre el modelo sin alteraciones que servirá como referencia para los demás, y otros dos que prueban algunas de las modificaciones implementadas.

7.2.a. Experimento inicial sobre los datos sintéticos

Para el primer experimento nos ceñimos al modelo propuesto 3 junto con los parámetros especificados **??** sin ninguna modificación. Los resultados obtenidos han sido los siguientes:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN simple	0.9201	0.5098	0.2242	2.8680	1.27



Obtenemos una precisión en entrenamiento del 92.01 % frente a 50.98 % en validación, produciéndose un tremendo sobreajuste sobre los datos de entrenamiento. En las gráficas vemos como 4 o 5 épocas eran suficientes para ajustar el modelo a los datos. A partir de ese punto, la precisión en validación baja ligeramente mientras que en entrenamiento continúa subiendo. De la misma forma, la pérdida acaba subiendo conforme aumenta el número de épocas.

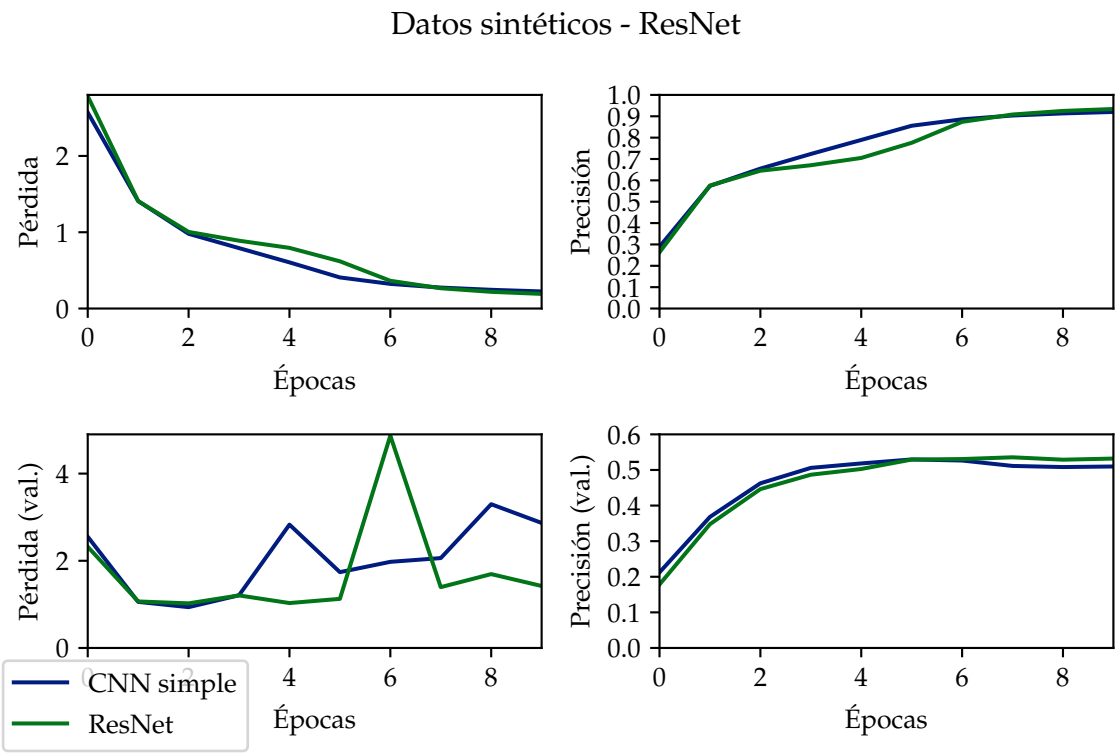
El tiempo de ejecución completo ha sido de 1.27 horas. No es particularmente alto, pero si consideramos que aún no hemos añadido ningún extra al modelo, y que la bases de datos utilizada es la sintética, esto nos hace pensar que el tiempo de ejecuciones de los posteriores experimentos será notablemente alto.

Ese 50.98 % de precisión en validación nos indica que el modelo acierta el siguiente símbolo LaTeX esperado en el 50.98 %, en media, de los casos.

7.2.b. ResNet

En este segundo experimento cambiamos la red convolucional utilizada para extraer las características de las imágenes por una ResNet. Mantenemos el resto de parámetros invariantes para comparar unicamente el resultado de ResNet. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Simple	0.9201	0.5098	0.2242	2.8680	1.27
ResNet	**0.9340**	**0.5320**	**0.1922**	**1.4217**	1.40



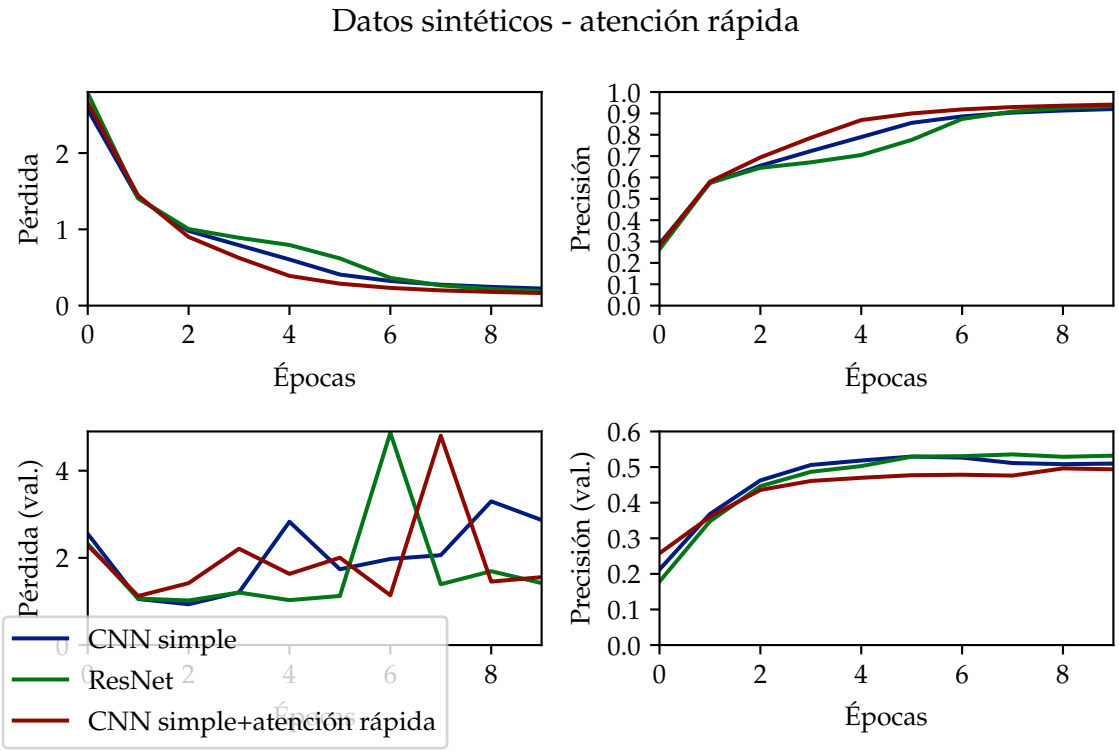
Vemos como el modelo mejora en todas las métricas estudiadas. Si bien el modelo con ResNet tarda un poco más en ajustarse a los datos, como se aprecia en la gráfica de precisión en validación, también se sobreajusta menos a los mismos - o al menos tarda más en hacerlo. A pesar de que la mejora en un 3 % de precisión no demasiado remarcable, si que nos indica cierta mejora así que nos incita a probar esta misma comparación en la base de datos IM2LATEX-170K, como se realiza en el experimento 7.3.d.

En cuanto al tiempo, este aumenta ligeramente, lo que se traducirá en una posterior sobrecarga temporal en su aplicación sobre IM2LATEX-170K.

7.2.c. Atención rápida

En este tercer y último experimento sobre los datos sintéticos probamos la atención eficiente explicada en la sección 5.2. Volvemos a utilizar la red convolucional simple y mantenemos los mismos parámetros. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Simple	0.9201	0.5098	0.2242	2.8680	1.27
ResNet	0.9340	**0.5320**	0.1922	**1.4217**	1.40
Atención eficiente	**0.9411**	0.4936	**0.1652**	1.5576	1.29



Vemos en la tabla como el modelo empeora en precisión en validación respecto al experimento original. Como ya explicamos en la sección 5.2 donde se explica la atención eficiente, el objetivo de esta es reducir el consumo de memoria, uno de los principales cuellos de botella del modelo. Si bien esto no se percibe particularmente ahora ya que el tiempo es practicamente el mismo, nos permitirá aumentar el tamaño del batch estudiado para IM2LATEX-170K, como veremos en el experimento 7.3.c.

7.3. Experimentos sobre IM2LATEX-170K

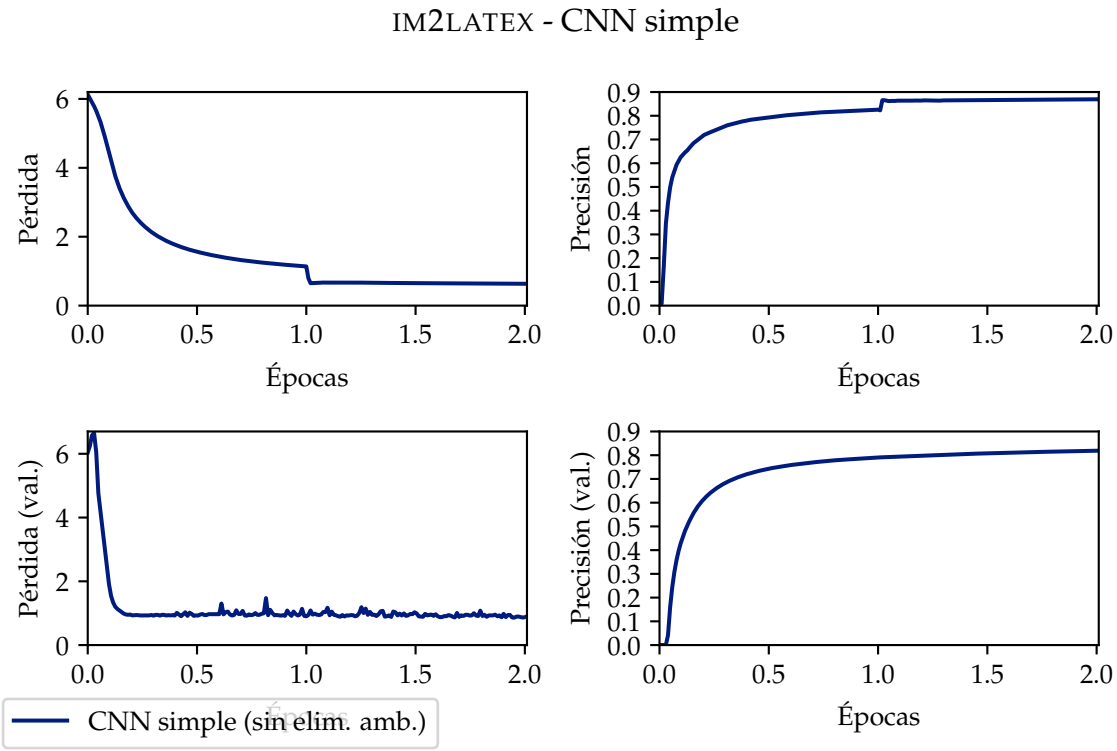
Para esta tanda de experimentos ponemos en uso el *early stopping* 4.6 implementado anteriormente. Este se dará si no se ha mejorado la precisión en validación en un 1 % durante 10 evaluaciones consecutivas. Realizamos una evaluación cada 50 batches de 32 imágenes cada uno.

Cabe destacar que estamos utilizando la precisión en validación. Esto es porque nuestro *early stopping* no busca evitar el sobreajuste sino simplemente reducir el monumental tiempo de ejecución sobre IM2LATEX-170K. De hecho, como veremos, el sobreajuste en IM2LATEX-170K no es ni mucho menos tan relevante como lo es sobre Toy.

7.3.a. Experimento inicial sobre IM2LATEX-170K

Para este primer experimento sobre la base de datos IM2LATEX-170K volvemos al modelo propuesto originalmente 3 junto con los parámetros comentados **??**, sin modificaciones adicionales. Los resultados obtenidos han sido los siguientes:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Simple	0.8695	0.8190	0.6340	0.9014	8.33



Lo primero que observamos tras este experimento fue sin duda el alto valor de precisión en validación: 81.90 %. Este valor tiene aún más relevancia si lo comparamos con el valor obtenido en el experimento 7.2.a de un 50.98 %. Cabría esperar que ante el aumento de complejidad de la gramática estudiada y las imágenes obtenidas el rendimiento del modelo fuese considerablemente inferior. Sin embargo, el resultado es exactamente el contrario. Podríamos pensar que puede deberse a que ante una gramática tan sencilla como la de Toy se produzca un tremendo sobreajuste. Aunque esto ocurre, no es la causa de su pobre valor de precisión, como ya se vió en su correspondiente análisis 7.2.a.

Nuestra teoría actual se basa en que el modelo aprende mucho mejor ante tal cantidad y variedad de imágenes como presenta IM2LATEX-170K. Esto nos sugiere probar técnicas para aumentar el conjunto de datos. Sin embargo, no nos valdría con simplemente tomar subimágenes de los datos existentes ya las imágenes generados por LaTeX se ajustan al texto en ellas, y el tomar subimágenes invalidaría el resultado esperado.

Volvamos a los resultados del experimento. Como podemos ver en las gráficas, en este experimento hemos entrenado únicamente durante dos épocas. Esto ya nos lleva a un tiempo de 8.33 horas, por lo que no era posible realizar más. Hemos decidido evaluar el modelo entre épocas para tener datos más explicativos, ya que unicamente dos medidas no nos permitían obtener suficiente información.

Fijándonos en la tabla podemos apreciar como vuelve a producirse sobreajuste: 86.95 % de precisión en entrenamiento frente a 81.90 % en validación. Sin embargo, este margen en considerablemente menor que el del 40 % que se producía sobre la base de datos sintética.

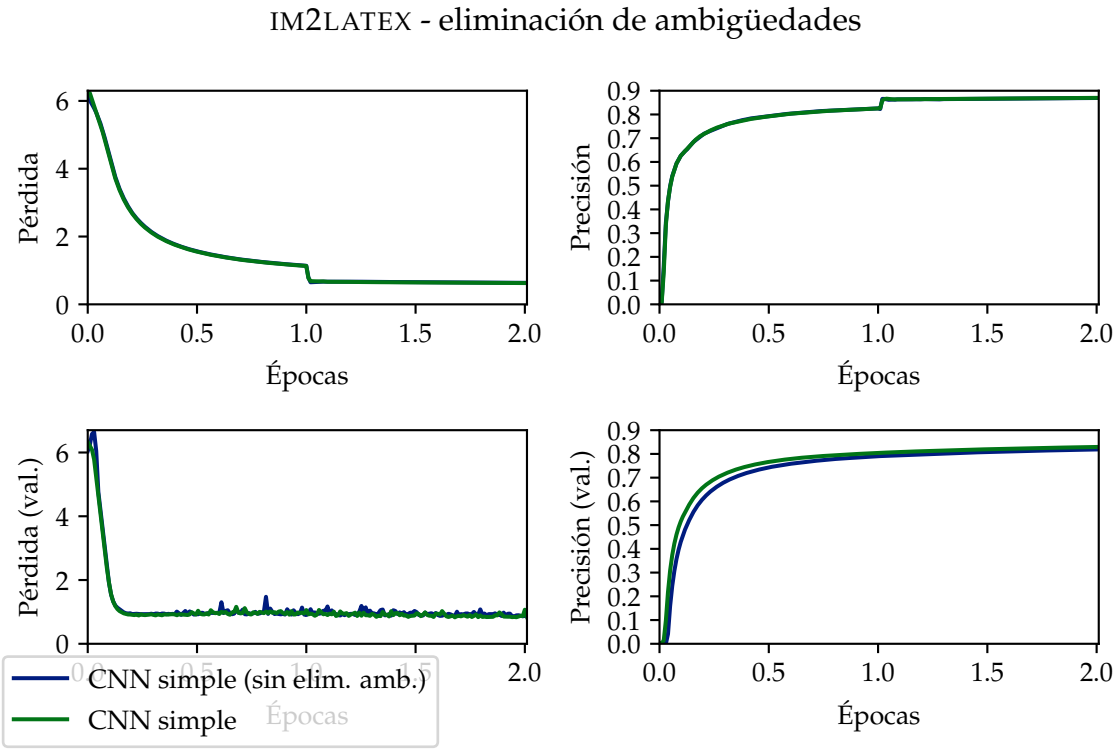
Cabe destacar la repentina mejora en precisión durante el entrenamiento al terminar la primera época. Esto nos muestra como el modelo recuerda especialmente bien los datos proporcionados y puede indicar que entrenar sobre los mismo datos durante demasiadas épocas podría ser contraproducente. Como podemos ver en precisión en validación, 2 épocas no son suficientes para sobreajustarnos a los datos ya que el modelo sigue mejorando hasta el final.

7.3.b. Eliminar ambigüedades

En este segundo experimento añadimos la eliminación de ambigüedades explicada en la sección 5.1 a los datos de IM2LATEX-170K. Este experimento no se realizó sobre Toy porque la gramática que definimos para generar dichos datos carece de ambigüedades.

Mantenemos el resto del modelo y parámetros invariantes. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Simple	0.8695	0.8190	0.6340	0.9014	8.33
CNN Elim. amb.	0.8701	0.8295	0.6272	0.8406	8.77



Ya que el modelo apenas varía, los resultados son muy parecidos. Mejoramos la precisión tanto en validación como en entrenamiento, por lo que el preprocesamiento ha merecido la pena. Adicionalmente, la convergencia del modelo un poco más rápida, como se aprecia en la gráfica de precisión en validación.

Respecto al tiempo, si bien este aumenta debido al preprocesamiento de los datos, almacenamos la formulas ya procesadas en un JSON para mitigar dicha sobrecarga de cara a proximos experimentos. Bastará con leer dicho JSON en vez de realizar de nuevo el preprocesamiento.

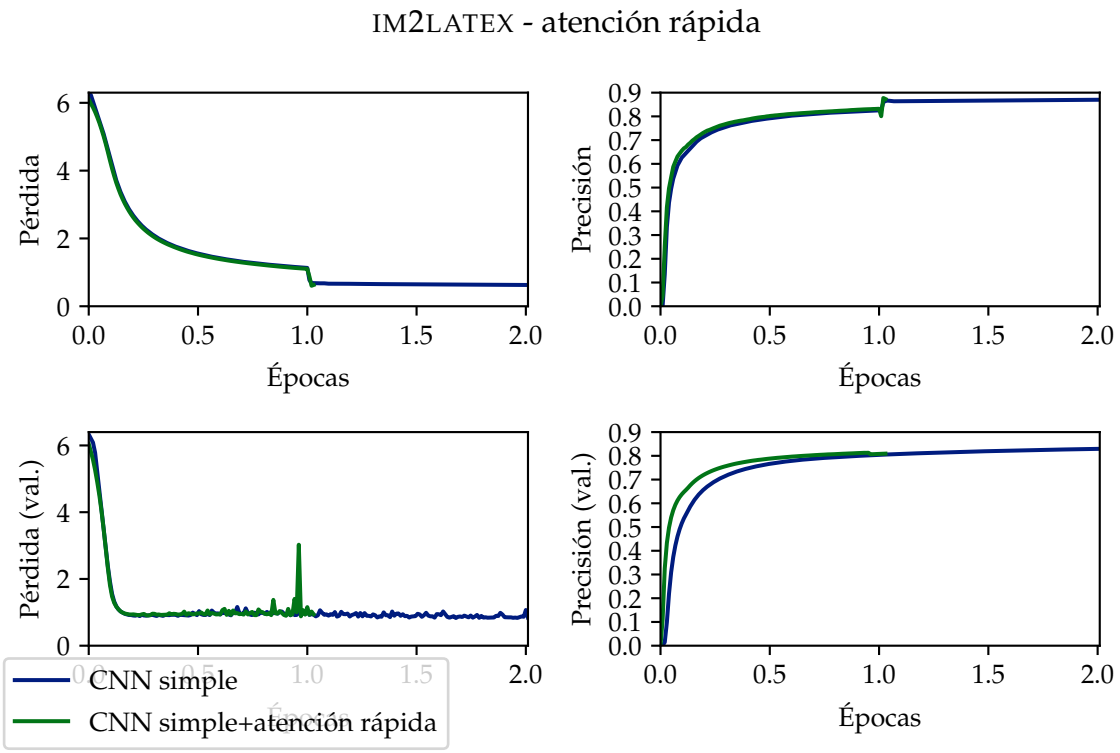
Finalmente, ya que este preprocesamiento aumenta la precisión sin añadir tiempo de cómputo al modelo, merece la pena en todos lo Vemos en la tabla como el modelo empeora en precisiñón en validación respecto al experimento original. Como ya explicamos en la sección 5.2 donde se explica la atención eficiente, el objetivo de ésta es reducir el consumo de memoria, uno de los principales cuellos de botella del modelo. Si bien esto no se percibe particularme ahora ya que el tiempo ess casos. Es por ello que a partir de ahora todos los experimentos se realizarán sobre los datos sin ambigüedades.

7.3.c. Atención eficiente

Comprobamos ahora la eficacia de la mejora de atención eficiente 5.2 aplicada sobre la base de datos IM2LATEX-170K. Como vimos en el experimento 7.2.a, la atención eficiente no tiene como objetivo mejorar la precisión del modelo, sino reducir el uso de memoria y, potencialmente, reducir el tiempo de ejecución del mismo.

Utilizamos para este el modelo anterior, con eliminación de ambigüedades, añadiendole el uso de atención eficiente. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Elim. Amb.	0.8701	**0.8295**	**0.6272**	**0.8406**	8.77
CNN Aten. Efic.	**0.8731**	0.8092	0.6281	0.9400	**4.68**



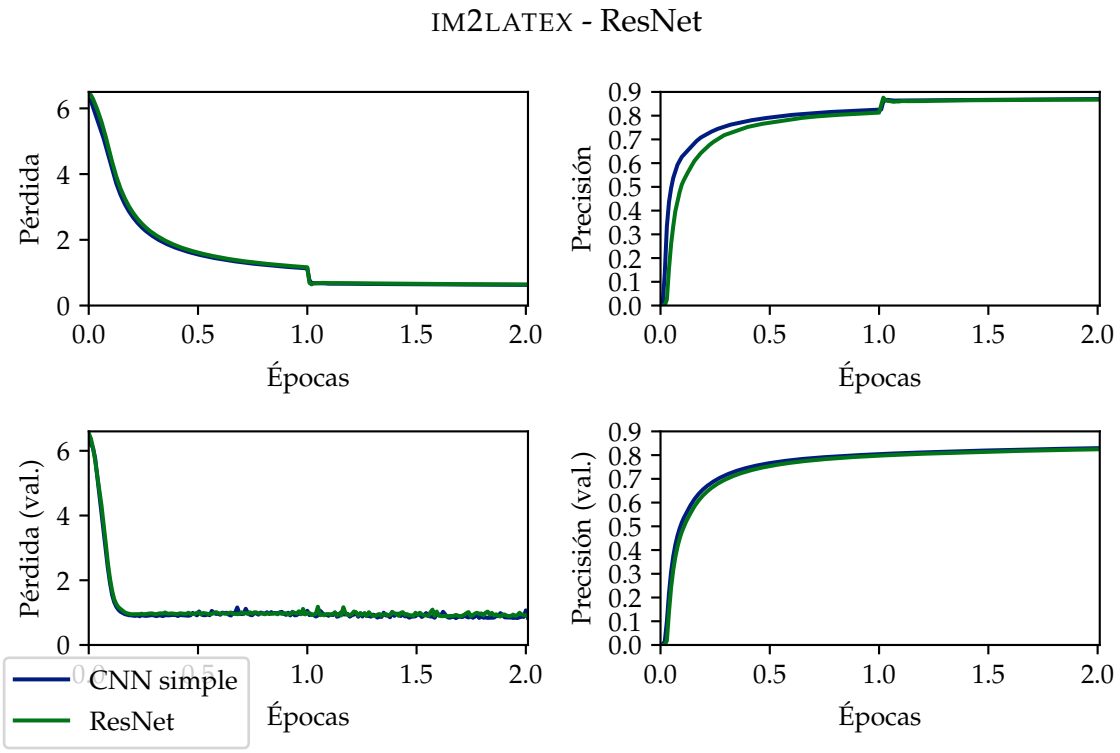
En este experimento se activo el *early stopping* tras dos evaluacines (100 batches de 32 imágenes cada uno) de la segunda época. Observando la tabla vemos como este modelo empeora la precisión en validación en un 2 %. Sin embargo, aumenta considerablemente la velocidad de convergencia. Es por ello que se reduce el tiempo de ejecución total en más de 4 horas.

Si bien se podría estar dando convergencia prematura, evitando que el modelo siga mejorando a partir de la segunda época, veremos en futuros experimentos que este efecto no se produce.

7.3.d. ResNet

Procedemos a estudiar el uso de ResNet en vez de la CNN simple utilizada hasta ahora, de igual manera que se hizo en el experimento 7.2.b. Desactivamos para ello la atención eficiente del experimento anterior pero mantenemos la eliminación de ambigüedades. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Elim. Amb.	**0.8701**	**0.8295**	**0.6272**	**0.8406**	8.77
ResNet Elim. Amb.	0.8674	0.8246	0.6412	0.8908	**8.57**



Observamos como el resultado de este modelo utilizando ResNet es ligeramente peor en precisión que el modelo anterior. Sin embargo, la diferencia es realmente pequeña, y es mayor en entrenamiento que en validación. De hecho, observando las gráficas de precisión vemos como se reduce la precisión entrenamiento es considerablemente peor durante la primera mitad de la primera época, mientra que la precisión en validación es practicamente idéntica. Esto muestra como el modelo con ResNet se sobreajusta menos manteniendo la misma precisión en validación.

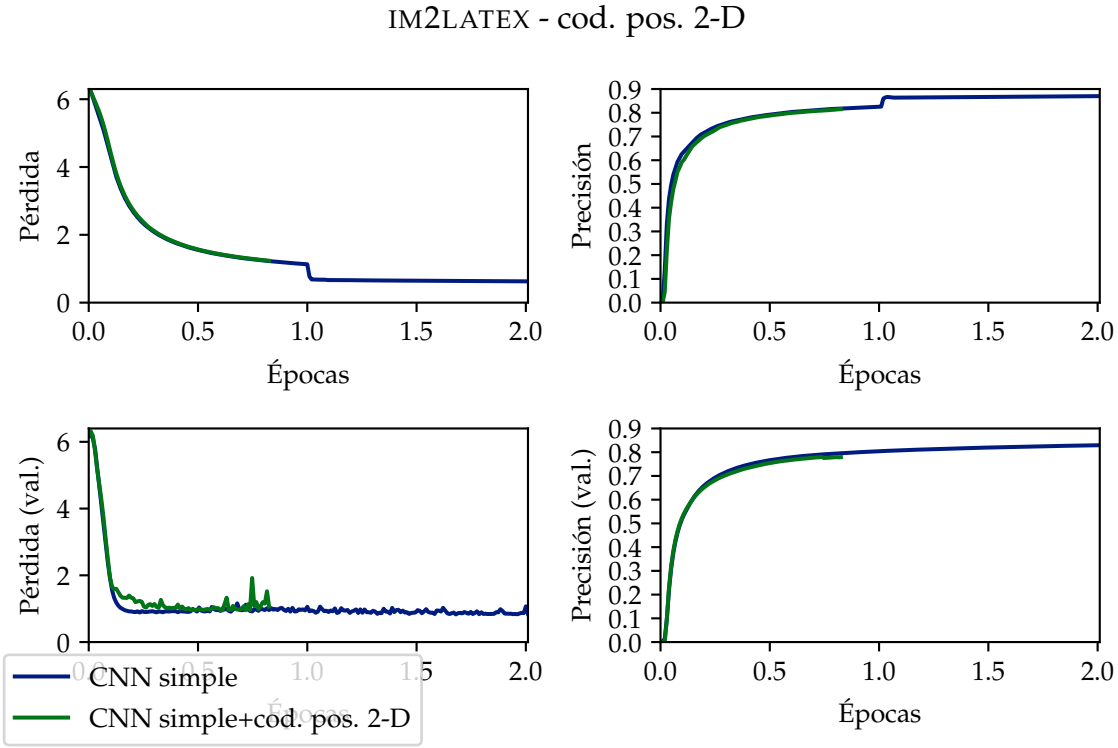
Estos resultados distan mucho de lo que esperábamos. Sustituir la CNN utilizada para extraer las características de la imagen por una mejor debería repercutir positivamente en los resultados del modelo.

Por otro lado, el tiempo de ejecución se reduce ligeramente. Es por ello que utilizaremos ResNet en los experimentos finales.

7.3.e. Codificación posicional 2-dimensional

A continuación ponemos a prueba la codificación posicional 2-dimensional explicada en esta sección 5.3. Para ello volvemos al modelo con una CNN simple quitando ambigüedades, y le añadimos la codificación posicional 2-dimensional. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Elim. Amb.	**0.8701**	**0.8295**	**0.6272**	**0.8406**	8.77
CNN Cod. Pos. 2d	0.8151	0.7791	1.2356	1.0639	**3.60**



Como podemos ver, el *early stopping* ha vuelto a activarse ante la falta de convergencia. Aunque los resultados en precisión apenas varían, la evolución se ha estancado antes de terminar la primera época. Los valores de pérdida en validación si tardan más en converger y oscilan, aunque ya hemos observado oscilaciones parecidas en experimentos anteriores.

En principio, este experimento no revela nada particularmente esclarecedor sobre la codificación posicional 2-dimensional, solamente un estancamiento prematuro. Lo utilizaremos en experimentos posteriores debido a esta inconclusión que debería ser estudiada con mayor profundidad.

7.3.f. Learning rate modificado

Este experimento es un poco distinto. No es de los experimentos que inicialmente enfocamos como para poner en esta memoria, sino como uno utilizado para probar distintas alternativas o ideas que quizás podrían mejorar el modelo pero no han acabado funcionando. Es por ello que no hemos realizado más experimentos en esta dirección. Sin embargo, creemos que por la idea merece la pena reportarlo aquí.

El eje central de este experimento es la evolución del *learning rate* del modelo, originalmente es la siguiente, proveniente de [9]:

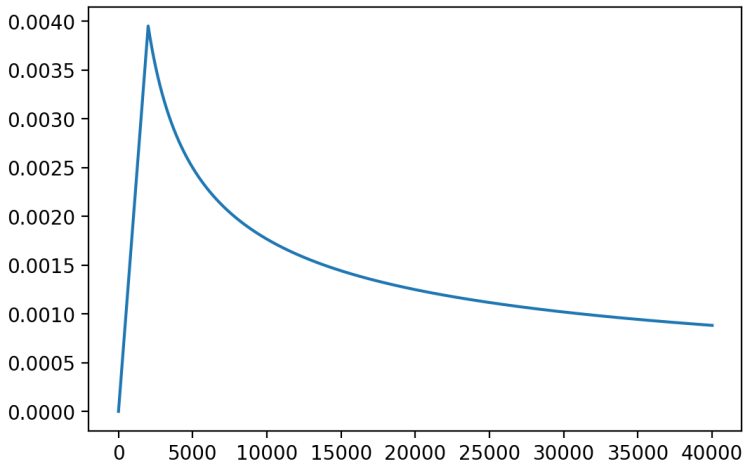


Figura 4: Evolución del learning rate original

Donde vemos el valor del *learning rate* respecto al número de imágenes que estudia el modelo. El *learning rate* se inicializa a 0.0040 y se reduce poco a poco conforme evoluciona el modelo.

Observamos que, al principio del entrenamiento, la precisión aumentaba rápidamente, y después mucho más lentamente. Nuestra idea fue permitir, en algunos momentos del entrenamiento, valores altos de learning rate en ciertos puntos del modelo más desarollado. Variantes de esta idea han sido probadas con gran éxito [5]. Propusimos la siguiente curva alterada:

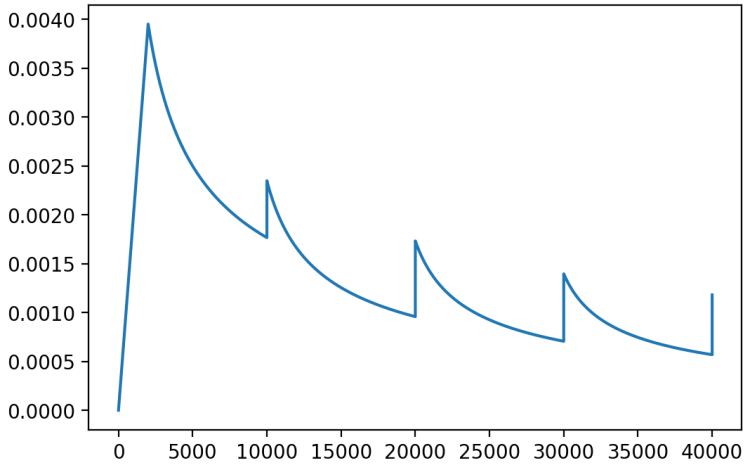
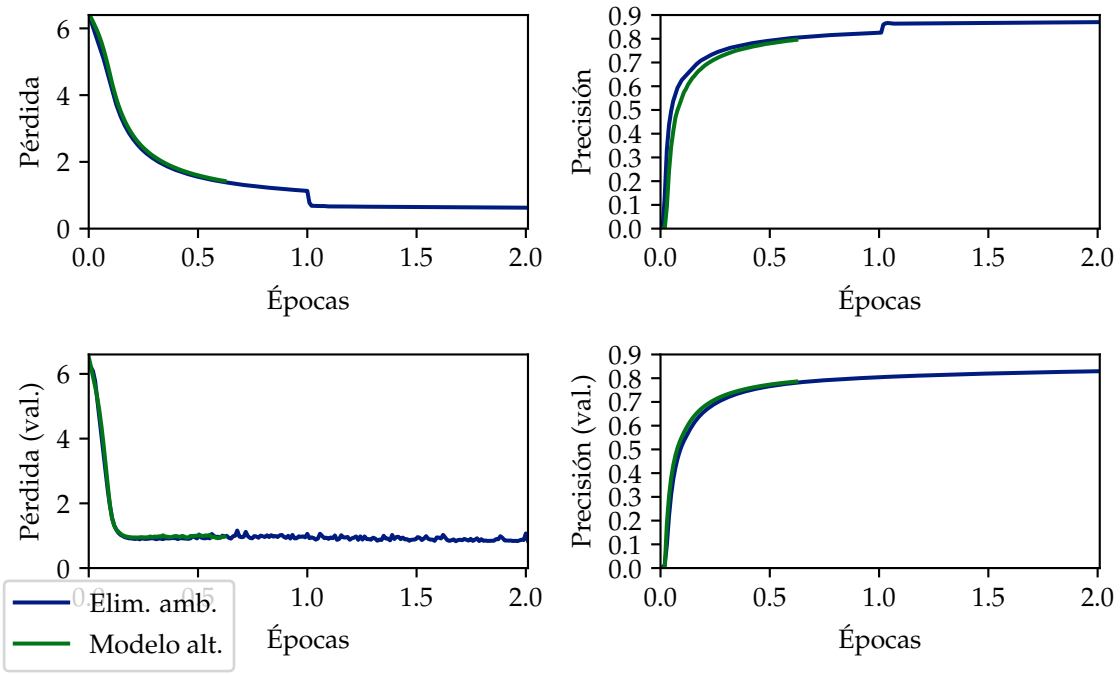


Figura 5: Evolución del learning rate propuesto

Utilizando esta nueva evolución para el *learning rate* lanzamos un nuevo experimento. El modelo de este experimento es una combinación de los experimentos 7.3.c y 7.3.d. Es decir, utiliza ResNet en vez de la CNN simple utilizada en el modelo inicial junto con atención eficiente. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
CNN Elim. Amb.	**0.8701**	**0.8295**	**0.6272**	**0.8406**	8.77
Modelo Alternativo	0.7956	0.7860	1.4274	0.9771	2.70

IM2LATEX modelo alternativo



Como podemos ver, el modelo activó el *early stopping* y se estancó mucho antes que el resto de modelos estudiados hasta ahora. Es por ello que descartamos esta línea de estudio.

7.4. Experimentos finales

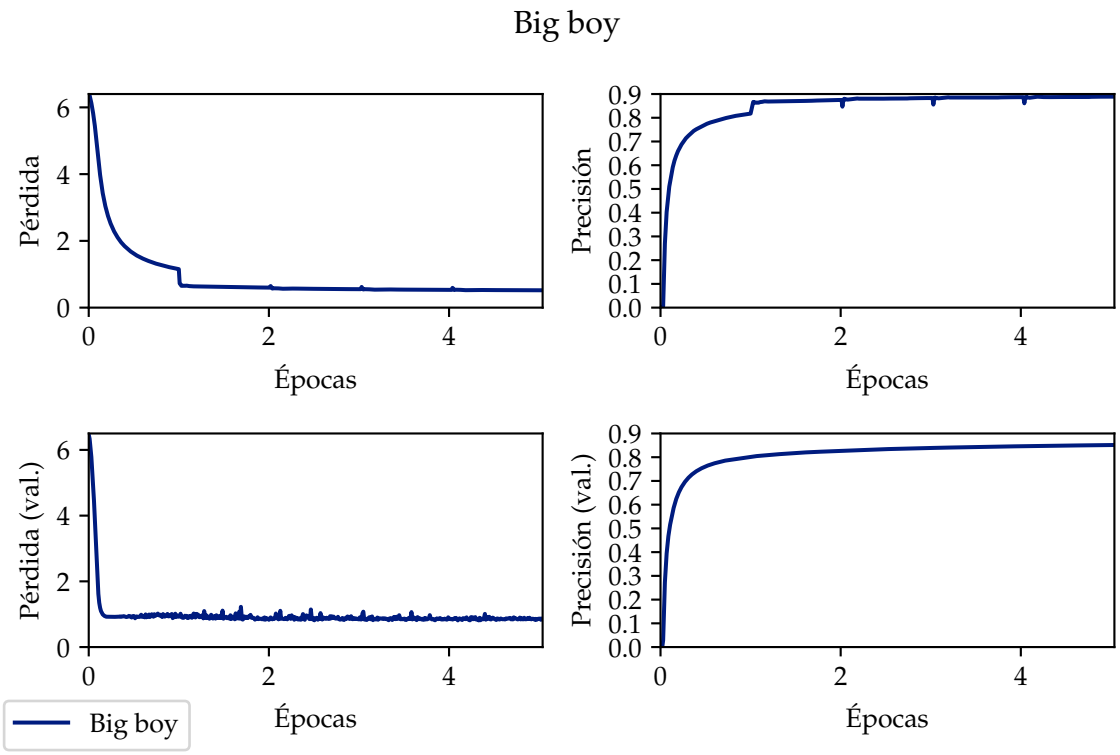
Para estos ultimos experimentos buscamos obtener el mejor modelo posible en términos de precisión en validación. Finalmente, evaluaremos el modelo correspondiente sobre el conjunto de test.

7.4.a. Combinación de los experimentos anteriores: Big boy

En primer lugar combinaremos todas las características implementadas en todos los experimentos anteriores y dejaremos que el modelo entrene todo lo posible sin alterar nada más. En particular, Utilizaremos eliminación de ambigüedades (7.3.b), atención eficiente (7.3.c), ResNet en lugar de la red convolucional simple (7.3.d) y codificación posicional 2-dimensional (7.3.e).

Reconfiguramos el *early stopping* para que sea active tras 10 evaluaciones sin una mejora del 0.01 % de precisión en validación y entrenamos el modelo durante 5 épocas. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
Big Boy	0.8889	0.8514	0.5196	0.8602	20.23



Hemos obtenido una precisión del 85.14 % en validación, la mayor hasta ahora. Vemos como la evolución de la misma se produce sobretodo durante la primera época, como ya ocurría en los experimentos anteriores. Sin embargo, el modelo nunca llega a estancarse ya que no ha activado el *early stopping* y ejecuta las 5 épocas al completo. Esto sugiere que podría mejorarse aún más, aunque sea ínfimamente, entrenándolo aún más.

Esto, claramente, no es una opción después de las 20.23 horas de entrenamiento que ha supuesto este experimento. Procedemos ahora a jugar con los parámetros del modelo para intentar mejorarlo todavía más.

7.4.b. Ajuste de parámetros: Bigger boy

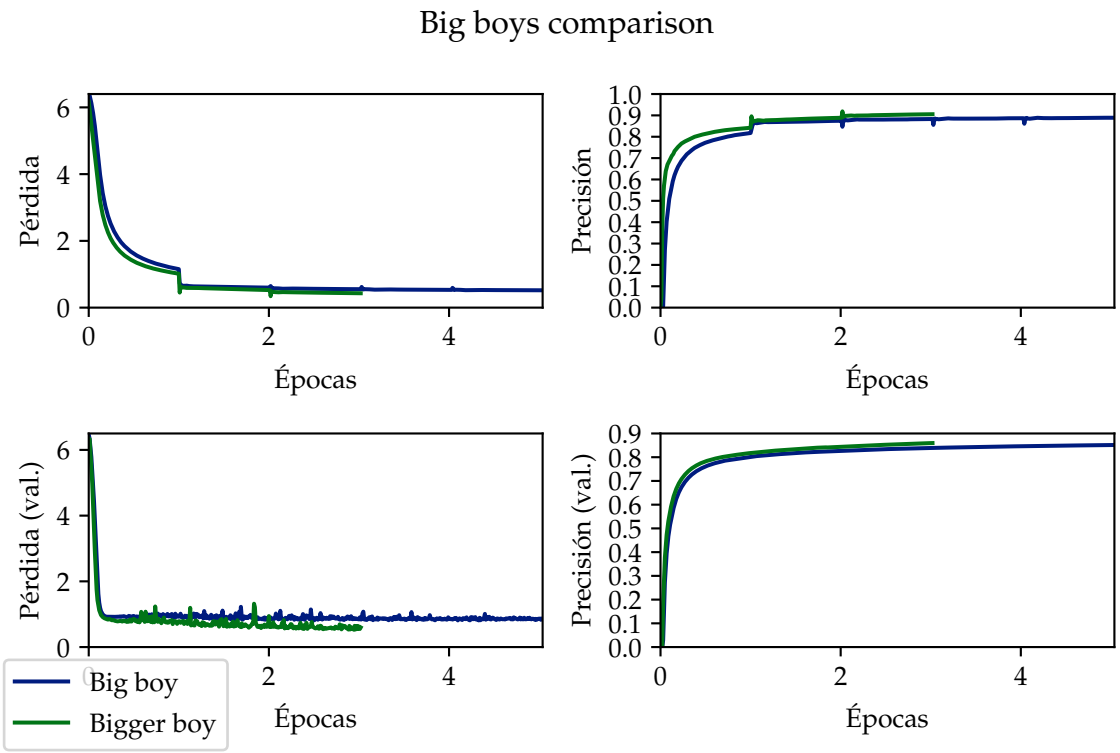
Para este último experimento antes de la evaluación final mantendremos el modelo anterior y reajustaremos algunos de sus parámetros. En particular:

- Aumentamos la profundidad del modelo de 20 a 32.
- Aumentamos el número de capas de 1 a 4.
- Aumentamos el número de cabezas de atención de 1 a 4, utilizando así atención multidimensional.

Este aumento de parámetros viene motivado por el hecho de que, en la bibliografía que emplea *transformers*, el empleo de modelos más grandes suele redundar en mejoras.

Adicionalmente, mantenemos la configuración de *early stopping* pero reducimos el número de épocas a 3 para reducir el tiempo de cómputo total. Veamos los resultados obtenidos:

	Precisión	Precisión en validación	Pérdida	Pérdida en validación	Tiempo (h)
Big Boy	0.8889	0.8514	0.5196	0.8602	20.23
Bigger Boy	**0.9054**	**0.8596**	**0.4309**	**0.5976**	18.40



Obteniendo en unicamente 3 épocas una precisión del 85.96 % en validación. Además, como se puede apreciar en las gráficas, las precisión en validación sigue creciendo cuando paramos el entrenamiento por encima de la del experimento anterior, volviendo a sugerir la necesidad de un entrenamiento prolongado.

Sin embargo, esta modificación en los parámetros nos ha salido cara: el tiempo de cómputo de 3 épocas ha aumentado a 18.40 horas. Esto nos indica que a pesar de que el tiempo sea cercano al de Big Boy, hemos conseguido mejor resultado con unicamente 3 épocas. Esto vuelve a reforzar la hipótesis de que con más datos el modelo sería aún mejor.

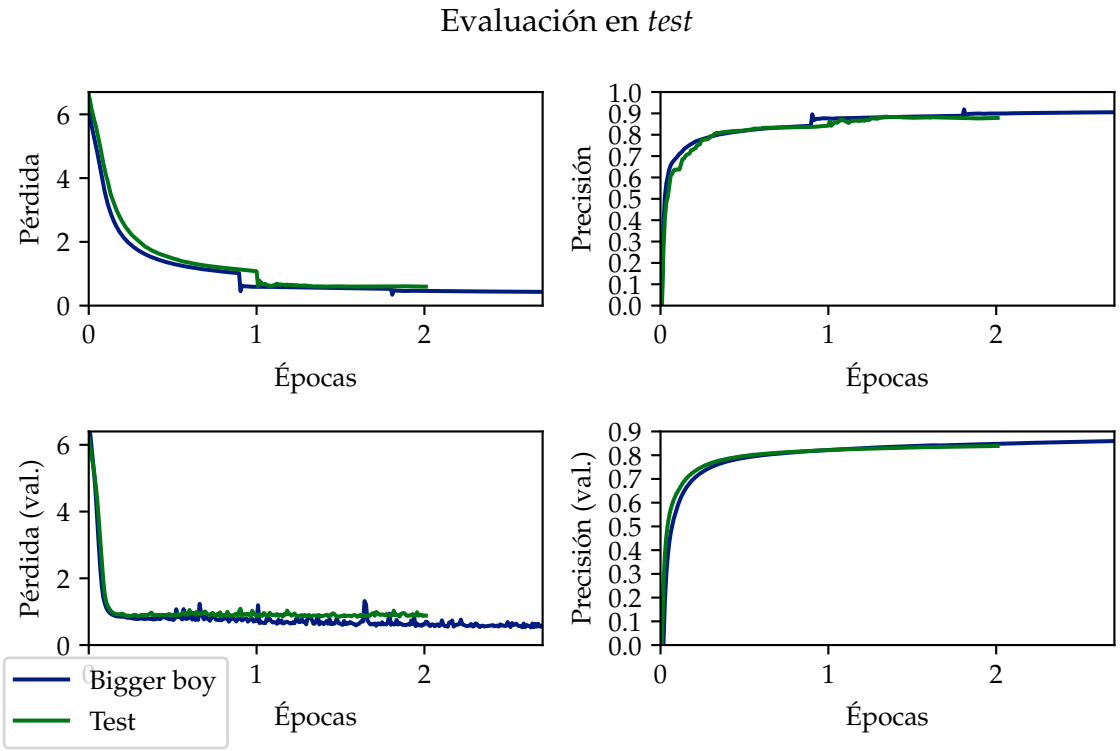
Cabe destacar finalmente la mejora en pérdida en validación. Esta es la primera comparación en la que puede observarse una mejora notable en este valor.

7.4.c. Evaluación sobre test

En este último experimento evaluamos el modelo anterior en el conjunto de test. Para ello entrenaremos con todo el conjunto de entrenamiento, que hasta ahora se había utilizado dividido entre entrenamiento y validación. El único cambio ha sido reducir la profundidad del modelo de 32 a 20 para reducir el tiempo de ejecución.

Adicionalmente, no hemos podido ejecutar este modelo sobre 3 epochs como hicimos con el anterior, sino unicamente sobre 2. Esto se debe a que el conjunto de entrenamiento es aún más grande (al incluir también a validación). Veamos los resultados obtenidos:

	Prec.	Prec. en val/test	Pérd.	Pérd. en val/test	BLEU	BLEU en test	Tiempo (h)
Bigger Boy	**0.9054**	**0.8596**	**0.4309**	**0.5976**	-	-	**18.40**
Test	0.8788	0.8389	0.5942	0.8817	0.8675	0.8098	19.43



Vemos como los resultados obtenidos no difieren particularmente de los estudios realizados en validación. Hemos alcanzado una precisión final en test del 83.89 % con tan solo dos epochs, lo que naturalmente es mejorable con más entrenamiento. Sin embargo, dicho tiempo ha ascendido a 19.43 horas al aumentar el tamaño de conjunto de entrenamiento.

Cabe destacar que este el primer experimento en el que hemos puesto en práctica la métrica BLEU, que fue implementada hacía el final del desarrollo. Esta nos da una aproximación de lo buena que es la traducción realizada. Los valores son consistentes respecto a los de precisión.

8. Conclusiones

En este trabajo hemos afrontado el problema de la obtención de código LaTeX de una serie de fórmulas a partir de las imágenes que generan. Para ello hemos utilizado un modelo codificador-decodificador con *transformers* junto con una red ResNet para extraer las características de las imágenes.

El modelo final entrena en 20 horas obteniendo una precisión del83.89 % y un valor de métrica BLEU del 80.98 % en test. A pesar de que este modelo es mejorable en muchos aspectos, consideramos los resultados obtenidos un completo éxito.

8.1. Trabajo futuro

En esta sección exponemos las líneas de trabajo que no hemos podido explorar por falta de tiempo:

- Aumento de datos: hemos visto en distintos experimentos como no son necesarias muchas épocas sobre los mismos datos, sino más datos distintos entre sí. De cara a generar nuevos datos se puede generar una gramática más compleja que la creada para la base de datos sintética, o bien aplicar técnicas de visión por computador a las imágenes, aunque con muchas restricciones para no invalidar los resultados asociados.

- En el experimento 7.3.d vimos como sustituir la CNN simple utilizada por una ResNet no mejoraba particularmente los resultados. La extracción de características es un proceso fundamental de nuestro modelo y deberíamos estudiar cómo mejorar el modelo en ese sentido. Quizás una red CNN simple sea suficiente para obtener toda la información de imágenes relativamente sencillas como las que utilizamos. Sería necesario estudiar un equilibrio entre buena extracción de características y minimizar el tiempo de ejecución en esta sección del modelo.
- En el experimento 7.3.e estudiamos la codificación posicional 2-dimensional sin llegar a ninguna conclusión específica. Actualmente no estamos seguros de si mejora o empeora el modelo, sólo de que las variaciones en los resultados obtenidas al introducir esta característica en el modelo no son especialmente significativas. Sería necesario estudiar en profundidad esta modificación.

Referencias

[1] Alex Taradachuk Blake Vente. *im2latex-170k*. 2020. URL: https://www.kaggle.com/rvente/im2latex170k/metadata.

[2] Krzysztof Choromanski y col. "Rethinking Attention with Performers". En: *arXiv:2009.14794 [cs, stat]* (30 de sep. de 2020). arXiv: 2009.14794. URL: http://arxiv.org/abs/2009.14794 (Accedido 13-01-2021).

[3] Anssi Kanervisto. *im2latex-100k , arXiv:1609.04938*. Zenodo, jun. de 2016. DOI: 10.5281/zenodo.56198. URL: https://doi.org/10.5281/zenodo.56198.

[4] Ali Rahimi y Benjamin Recht. "Random Features for Large-Scale Kernel Machines". En: *Proceedings of the 20th International Conference on Neural Information Processing Systems*. NIPS'07. Vancouver, British Columbia, Canada: Curran Associates Inc., 2007, págs. 1177-1184. ISBN: 9781605603520.

[5] Leslie N. Smith y Nicholay Topin. *Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates*. 2018. arXiv: 1708.07120 [cs.LG].

[6] Yi Tay y col. *Efficient Transformers: A Survey*. 2020. arXiv: 2009.06732 [cs.LG].

[7] Tensorflow team. *Tensorflow `Dataset` class*. TensorFlow. 2020. URL: https://www.tensorflow.org/api_docs/python/tf/data/Dataset (Accedido 28-01-2021).

[8] Tensorflow team. *Tensorflow Keras Tokenizer*. TensorFlow. 2020. URL: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/text/Tokenizer (Accedido 21-12-2020).

[9] Ashish Vaswani y col. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].