75.26 - Simulación

Facultad de Ingeniería
Universidad de Buenos Aires

Mermet, Ignacio Javier
98153

Julio 2023

Índice

1 Introducción

En el presente trabajo se apunta a explicar el paper *Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations*[12] de *Lu et. al.* Se introduce la motivación de los autores, una breve reseña de la literatura para explicar el estado del arte en el momento de publicación, se explican los puntos principales del paper y se muestran los resultados obtenidos al intentar replicar los resultados obtenidos en el paper.

1.1 Entrenamiento de redes neuronales muy profundas

Las redes neuronales convolucionales empezaron a dominar el campo de vision por computadora luego de la introducción de AlexNet [10] en 2012. Una gran ventaja de esta familia de modelos es que aprenden mapas de *features* junto con un clasificador en un proceso punta a punta. De este modo, no es necesario que se definan a priori las *features* relevantes de la imagen o del dominio bajo estudio.

Desde entonces, se ha acelerado el estudio de estas familias de modelos, con aplicaciones varias en imágenes, lenguaje natural, sonido, procesamiento de señales, sistemas de recomendaciones, entre otras áreas. Distintos papers elaboran sobre la importancia de la profundidad de dichas redes para lograr mejores resultados. Es decir, la performance, bajo alguna medida de performance relevante al problema, aumenta al agregar capas de parámetros libres o bloques con más neuronas. Sin embargo, el entrenamiento de modelos cada vez más grandes ha expuesto varios problemas relacionados a la convergencia del entrenamiento.

Dos problemas comunes a la hora de entrenar redes neuronales muy profundas son la explosión de gradientes y el desvanecimiento de gradientes. El primero se refiere a que los valores de los gradientes a medida que se propagan por la red toman valores exponencialmente más grandes, hasta tomar el valor `NaN`. De este valor, no se pueden recuperar, dado que las operaciones de matmul no se encuentran definidas. El segundo se refiere al caso contrario, donde los gradientes se vuelven progresivamente más pequeños y tienden a cero, frenando por completo el entrenamiento. En ambos casos, la convergencia de la red se ve afectada. Ambos problemas se hacen más aparentes en redes neuronales muy profundas. Estos problemas han sido, a grandes rasgos, resueltos mediante el uso de distintos métodos de regularización entre capas así como mediante inicialización normalizada de los parámetros de la red.

Sin embargo, aún convergiendo, una consecuencia de aumentar la profundidad de las redes, es que se evidencia una degradación en el error de aprendizaje: la precisión de la red se satura y luego empeora al intentar agregar más capas. Sería natural pensar que se trata de un caso de sobreajuste u *overfitting*, pero no es el caso.

1.2 Redes residuales

Dentro de la literatura que trata de resolver este problema, [5] propone lo que denomina aprendizaje de residuos para resolver el problema de degradación. En esta sección hago un breve resumen de §1 del paper de ResNet a fin de proveer el contexto necesario para el resto de esta monografía.

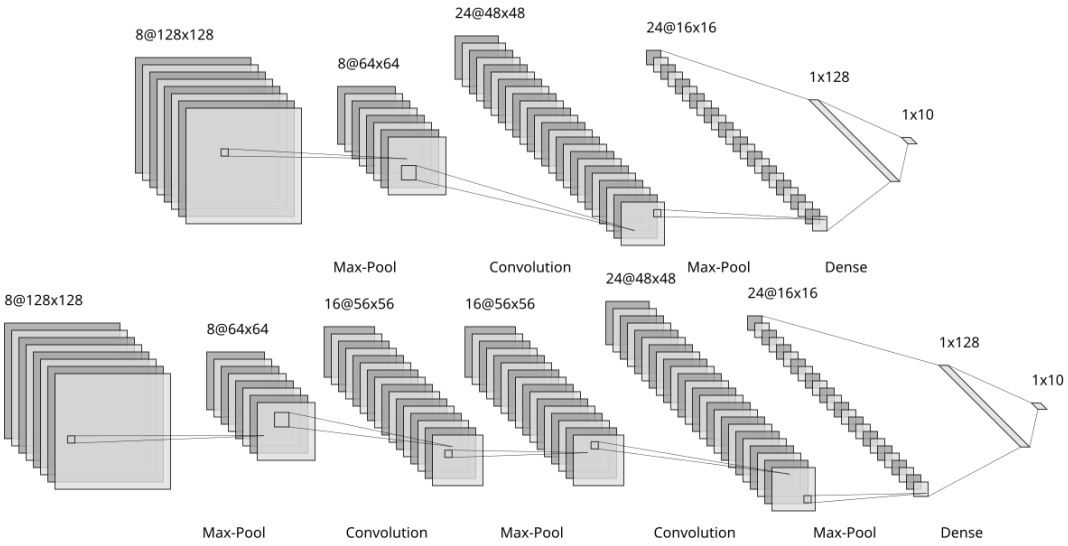

Figura 1: Ejemplo de CNNs

La segunda CNN en 1 posee una capa más de conv-maxpooling que la primera. Asumamos que las demás capas están copiadas de la primer CNN de la figura, con sus mismos pesos. La capa extra podría aprender un mapeo de identidad y por tanto tener *al menos* la misma precisión que la primer CNN de la figura. Esta solución por construcción de una red efectivamente más profunda indicaría que una red más profunda debería tener un error de entrenamiento más chico que una red más chica, pero esto se contradice con distintos resultados experimentales.

En lugar de esperar que algún conjunto de capas aprendan un mapeo deseado, se les hace aprender un mapeo residual: deseando aprender el mapeo $\mathscr{H}(x)$, dejan que ese conjunto no lineal de capas aprenda $\mathscr{F}(x) = \mathscr{H}(x) - x$, de modo de obtener la función original como $\mathscr{F}(x) + x$. Los autores conjeturan que es más fácil optimizar el mapeo residual que el mapeo original. Volviendo a nuestro argumento de construcción, si el mapeo identidad fuera óptimo, sería más facil llevar el residual a cero que aprender $\mathscr{H}(x)$ como un mapeo de identidad.

Dicha formulación puede ser expresada como conexiones entre capas no contiguas de una red neuronal [14]. ResNet utiliza mapeos identidad como conexiones de "atajo" entre capas (2), dado que no introducen costo computacional extra ni agregan parámetros libres a la red. La salida de estas conexiones simplemente se suma a la salida de la capa. De este modo, se puede optimizar por backpropagation. En 1.2 se ilustra su implementación utilizando `pytorch`.

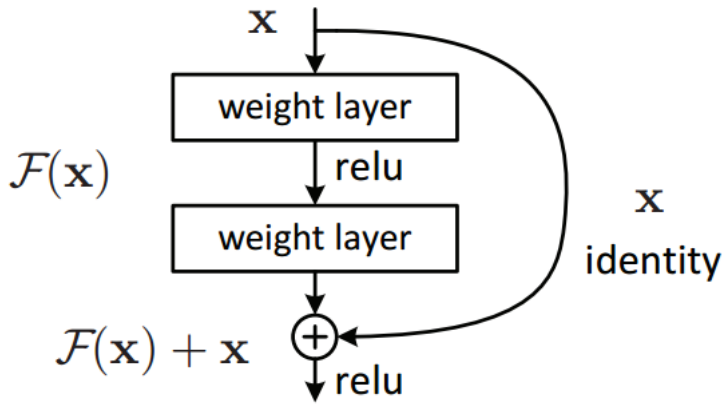

Figura 2: Bloque de ResNet

```python
class ResidualBlock(nn.Module):
    def __init__(
        self,
        in_channels: int,
        out_channels: int | None = None,
        stride: int = 1,
        kernel_size: int = 3,
        padding: int = 1,
        downsampling: bool = False,
    ):
        super().__init__()

        if out_channels is None:
            out_channels = in_channels

        self.downsampling = downsampling

        self.main_block = nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size=kernel_size,
                stride=stride,
                padding=padding,
            ),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(),
            nn.Conv2d(
                out_channels,
                out_channels,
                kernel_size=kernel_size,
                # stride=stride,
                padding=padding,
            ),
            nn.BatchNorm2d(out_channels),
        )

        if not downsampling:
            self.res_f = nn.Identity()
        else:
            self.res_f = nn.Sequential(
                nn.Conv2d(
                    in_channels,
                    out_channels,
                    kernel_size=1,
                    stride=stride,
                    # padding=padding,
                ),
                nn.BatchNorm2d(out_channels),
            )

    def forward(self, x):
        residual = x
        out = self.main_block(x)
        residual = self.res_f(residual)
        out += residual
        out = F.relu(out)
        return out
```

Figura 3: Ejemplo de implementación con `pytorch` de un bloque residual.

Al implementar esta arquitectura, se muestran resultados experimentales que indican que:

- Arquitecturas de redes residuales muy profundas son fácilmente optimizables
- El error de sus contrapartes no residuales aumenta conforme aumenta su profundidad
- Las redes residuales pueden aumentar su precisión simplemente agregando más bloques

2 Arquitecturas residuales como sistemas dinámicos

Observemos, brevemente, que cada bloque residual de ResNet puede ser descrito por el siguiente sistema dinamico discreto [2]:

$$y_l = h(z_l) + \mathscr{F}(z_l, W_l) \tag{1}$$
$$y_{l+1} = g(y_l) \tag{2}$$

Donde z_l es la entrada del l-ésima capa, z_{l+1} es la salida de la l-ésima capa, y_l es una variable auxiliar de la l-ésima capa, h, g son mapeos que pueden ser no-lineales. El entrenamiento se determina experimentalemente [6] que es más fácil si tanto g como h son el mapa identidad.

Consideremos G como la inversa de g, entonces el sistema dinámico anterior puede ser descrito como

$$z_{l+1} = G(h(z_l) + \mathscr{F}(z_l, W_l)) \tag{3}$$

Para que el sistema sea estable (no haya desvanecimiento o explosión de gradientes), el gradiente del lado derecho debe ser cercano a la identidad. Asumiendo $\mathscr{F}$ como una pequeña perturbación, entonces se requiere que $\nabla G \nabla h \approx I$, lo cual se cumple si g y h son mapas identidad. En tal caso, 3 se convierte en:

$$z_{l+1} = z_l + \mathscr{F}(z_l, W_l) \tag{4}$$

Lo cual se puede ver como una discretización del sistema dinámico

$$\frac{dz}{dt} = \mathscr{F}(z, W(t)) \tag{5}$$

Cuya discretización más sencilla toma la forma

$$z_{l+1} = z_l + \Delta t_l \mathscr{F}(z_l, W_l) \tag{6}$$

Donde Δt_l es el tamaño del l-ésimo paso. Notemos que esto es el primer paso del método de Euler.

2.1 PolyNet

En [15], se propone una arquitectura llamada “Inception” cuyos bloques se muestran en 4.

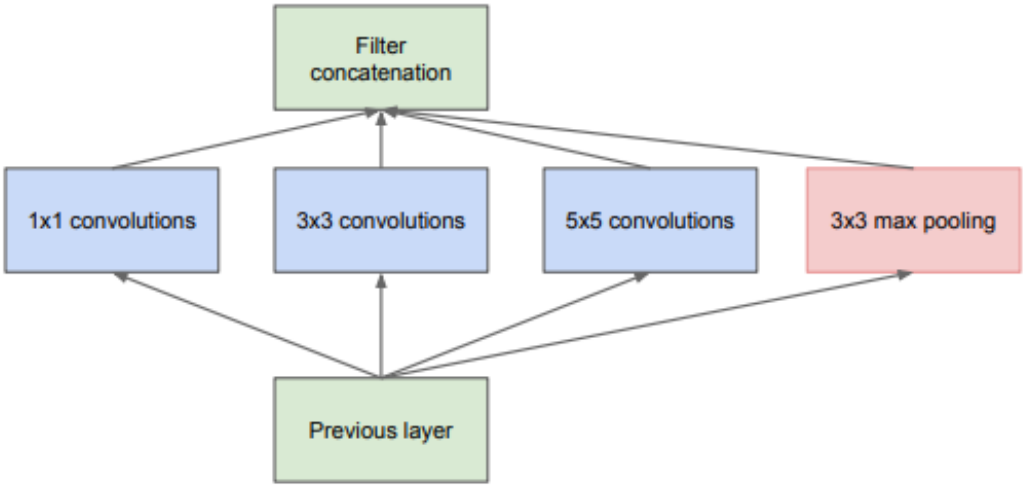

Figura 4: Bloque de Inception

Zhang et al proponen en [17] utilizar bloques de Inception como bloques residuales. En 5 se muestra el diagrama original.

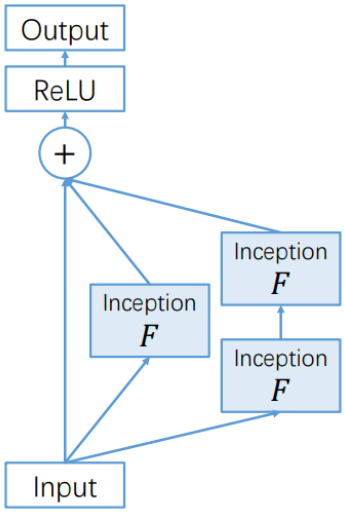

Figura 5: Bloque de PolyNet

Es necesario aclarar que todos los bloques etiquetados “Inception F” comparten sus parámetros. Configuraciones donde distintos bloques no comparten parámetros son también estudiadas. La motivación principal es estudiar composiciones polinomiales de operadores para explorar la diversidad estructural que permite combinar aditivamente distintos bloques.

Notar que esto resulta en un término de segundo orden respecto de ResNet.

$$(I + F + F^2) \cdot x = x + F(x) + F(F(x)) \tag{7}$$

Cuyo lado izquierdo puede reescribirse como

$$I + (I + F)F = I + F + F^2 \tag{8}$$

De modo tal que su computo sea mas eficiente.

Esta formulación puede verse como un paso del método implícito de Euler

$$u_{n+1} = (I - \Delta t f)^{-1} u_n \tag{9}$$

Para la ecuación diferencial $u_t = f(u)$. El método implícito permite pasos mas grandes, de modo que son necesarios menos bloques encadenados para lograr clasificación de estado del arte.

2.2 FractalNet

FractalNet [11] muestra que el aprendizaje explicito de residuales no es un imperativo para entrenar redes muy profundas.

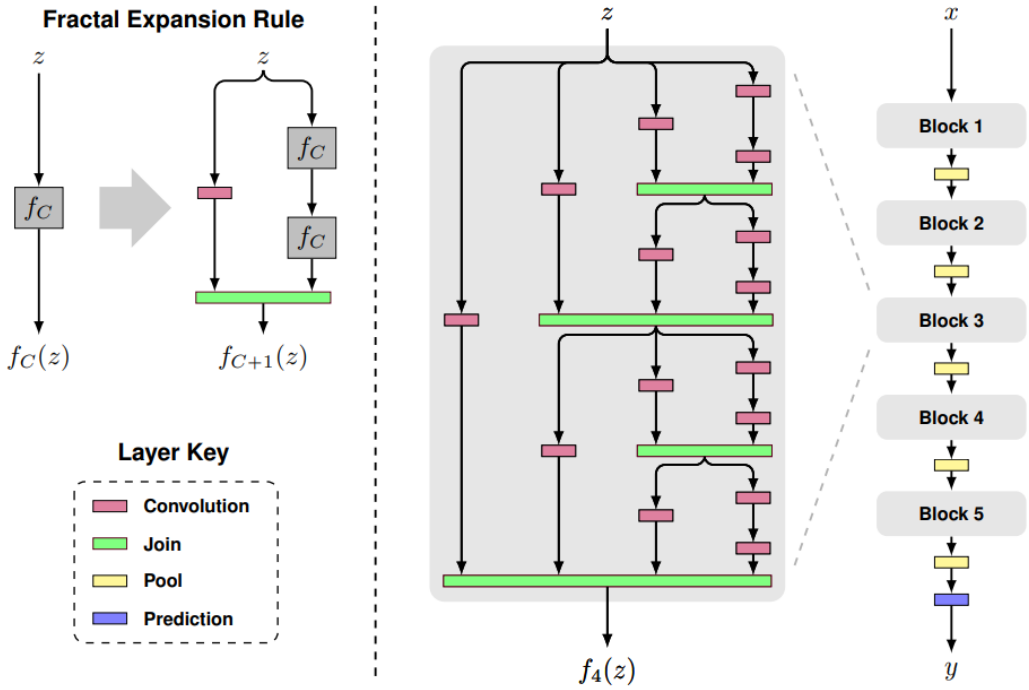

Figura 6: Regla de expansión de FractalNet

6 muestra la regla de expansión de FractalNet. Siendo C el indice de un fractal truncado $f_C(\cdot)$, la estructura de FractalNet esta definida por $f_C(\cdot)$. El caso base se define como una red con una sola capa convolucional: $f_1(z) = \text{conv}(z)$.

Los fractales sucesivos se definen como

$$f_{C+1}(z) = [(f_C \circ f_C)(z)] \oplus [\text{conv}(z)] \tag{10}$$

Donde $\circ$ denota composición de funciones y $\oplus$ un operador de unión. Este operador de unión puede ser, por ejemplo, concatenación o suma elemento a elemento. Los autores utilizan el promedio elemento a elemento.

Entonces, 10 puede ser reescrito como:

$$f_{c+1} = \frac{1}{2} f_c \circ f_c + \frac{1}{2} k_c \tag{11}$$

Entonces, cada bloque de FractalNet de orden dos puede ser reescrito como:

$$x_{n+1} = k_1 \times x_n + k_2 \times (k_3 \times x_n + f_1(x_n)) + f_2(k_3 \times x_n + f_1(x_n)) \tag{12}$$

Lo cual puede ser considerado como un caso de un esquema de Runge-Kutta de segundo orden para la EDO $u_t = f(u, t)$.

2.3 RevNet

Recordemos que un sistema dinamico es reversible si su función de transición de estados es biyectiva. Es decir, para cada estado existe un unico estado anterior desde el cual podría haberse llegado.

En [4] se propone RevNet, una arquitectura compuesta de una serie de bloques reversibles. Los parámetros de cada capa se particionan en dos grupos x_1, x_2. Cada bloque reversible

toma dos entradas (x_1, x_2) y produce dos salidas (y_1, y_2) acorde a las reglas en 13 y funciones residuales $\mathscr{F}$ y $\mathscr{G}$ análogas a ResNet.

$$y_1 = x_1 + \mathscr{F}(x_2) \tag{13}$$
$$y_2 = x_2 + \mathscr{G}(y_1) \tag{14}$$

Por tanto, las activaciones de una capa pueden ser reconstruidas desde las activaciones de la próxima capa:

$$x_2 = y_2 - \mathscr{G}(y_1) \tag{15}$$
$$x_1 = y_1 - \mathscr{F}(x_2) \tag{16}$$

De este modo, se ejecuta backpropagation sin guardar las activaciones en memoria.[1] De este modo, su costo en memoria es independiente de su profundidad, a excepción de algunas pocas capas no reversibles.

Reformulemos 13 como el siguiente sistema dinámico discreto:

$$X_{n+1} = X_n + \mathscr{F}(Y_n) \tag{17}$$
$$Y_{n+1} = Y_n + \mathscr{G}(X_{n+1}) \tag{18}$$

Con lo cual podemos apreciar su interpretación como un paso del método de Euler del sistema dinámico:

$$\dot{X} = f_1(Y, t) \tag{19}$$
$$\dot{Y} = f_2(X, t) \tag{20}$$

2.4 LM-ResNet

Notemos que todas las formulaciones anteriores se corresponden con métodos de un solo paso para aproximar sistemas dinámicos. En [12] proponen una arquitectura basada en el método lineal de múltiples pasos.

2.4.1 Métodos multipasos

Los métodos de Euler y Runge Kutta mencionados previamente requieren solamente conocer el valor de y_i para poder calcular y_{i+1}. En un método multi pasos, se emplean dos o más puntos anteriores para calcular el siguiente punto.

Dado un problema a valores iniciales

$$\begin{cases} y\prime = f(t, y), \ \ a \leqslant t \leqslant b \\ y(a) = \alpha \end{cases} \tag{21}$$

Un método multi pasos de orden $m > 1$ es aquel cuya ecuación en diferencias para aproximar y_{i+1} en t_{i+1} se corresponde con la ecuación:

$$y_{i+1} = a_{m-1}y_i + a_{m-2}y_{i-1} + \ldots + a_0 y_{i+1-m} \tag{22}$$
$$+ h\left[b_m f(t_{i+1}, y_{i+1}) + b_{m-1}f(t_i, y_i) + \ldots + b_0 f(t_{i+1-m}, y_{i+1-m})\right] \tag{23}$$

Con $i = m-1, m, \cdots, N-1$ y donde $a_0, a_1, \cdots, a_{m-1}$ y $b_0, b_1, \cdots, b_m$ son constantes y los valores iniciales $y_0 = \alpha, y_1 = \alpha_1, y_2 = \alpha_2, \cdots, y_{m-1} = \alpha_{m-1}$ son especificados.

Si $b_m = 0$, el método es **explícito o abierto** ya que el valor de w_{i+1} se da explícitamente en términos de los valores previamente calculados. Cuando $b_m \neq 0$, el método es **implícito o cerrado**, ya que el valor de w_{i+1} se encuentra en ambos miembros de la ecuación y se especifica sólo implícitamente.

[1] La VRAM en GPUs suele ser un limitante en el entrenamiento. Si la arquitectura tiene muchos parámetros, es común utilizar un batch size más chico, lo cual es subóptimo para SGD con mini batches.

2.4.2 Formulación de LM-ResNet

La arquitectura LM puede ser formulada como

$$u_{n+1} = (1 - k_n)u_n + k_n u_{n-1} + f_n(u_n) \tag{24}$$

Donde $k_n \in \mathbb{R}^n$ es un parámetro entrenable de cada capa que actúa como factor de escalado del residuo de la capa actual y la anterior. La figura 7 muestra el bloque de LM-ResNet *acorde al código original*. Notar que 8, el bloque según el paper, presenta diferencias. En primer instancia, el código original multiplica el residuo de la capa previa por $1 - k_s$, pero acorde a la ecuación 24, y el diagrama 8, se multiplica por k_s. El diagrama original es poco claro también, pareciendo indicar que se multiplica u_n tanto por k_s como por $1 - k_s$.

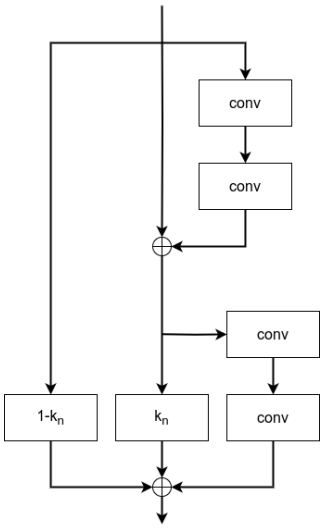

Figura 7: Bloque de LM-ResNet, acorde al código original.

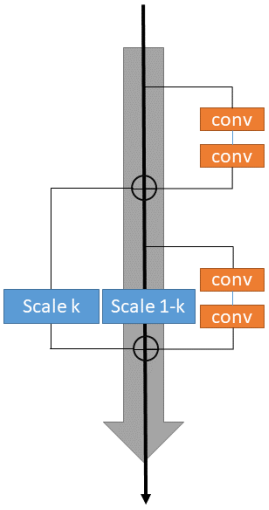

Figura 8: Bloque de LM-ResNet, acorde al paper original.

Para los resultados experimentales, se toma como verdad la ecuación 24. De otro modo, deberíamos modificar 24 como:

$$u_{n+1} = f_n(u_n) + k_n \times u_n + (1 - k_n) \times u_{n-1} \tag{25}$$

3 Resultados experimentales

En esta sección se detallan los resultados de distintos experimentos desarrollados a fin de replicar los resultados expuestos en el paper. A estos efectos, se utiliza el código original[2] y se arma un pipeline de entrenamiento acorde a lo relatado en el paper.

El código utilizado para obtener estos resultados está disponible en https://github.com/CrossNox/beyond. Los aportes hechos en este repositorio son:

- Corrección del bug antes mencionado
- Mejoras varias al código, incluyendo aquellas para reducir el impacto en memoria de entrenamiento
- Modularización del código
- Armado de notebooks para entrenar modelos
- Armado de notebooks para realizar los gráficos resultantes del entrenamiento

3.1 CIFAR10

Los experimentos aquí detallados se realizan utilizando el dataset CIFAR10. Este dataset se compone de 60000 imágenes de 32x32 píxeles con tres canales de color (RGB). Se divide en 50000 imágenes de entrenamiento y 10000 para el set de pruebas.

Siguiendo lo especificado en §2.2, se aplican técnicas de image augmentation al dataset de entrenamiento. En particular, se agrega un padding de 4 píxeles a cada lado, de modo tal que la imagen pasa a tener 40x40 píxeles. Con una probabilidad del 50 % se espeja la imagen horizontalmente. Finalmente, se toma un recorte aleatorio de 32x32 píxeles.

En etapa de pruebas, se toma una vista simple de la imagen original de 32x32. Es importante notar esto dado que el paper de ResNet utiliza otra técnica de evaluación, tomando múltiples vistas de la imagen original.

3.2 Hiperparámetros relevantes

A fin de seguir los lineamientos planteados por el paper, se modificó el código original y se implementó un pipeline de entrenamiento con los hiperparámetros especificados. El batch size se configuró como 128 imágenes por batch. Se utiliza SGD como optimizador, con weight decay configurado como 0.0001 y momentum como 0.9. En 9 se especifica la curva del parametro de learning rate.

Figura 9: Curva de learning rate

El entrenamiento se detiene a los 160 epochs.

Todos los experimentos fueron llevados a cabo con CUDA 12.1 en una GPU Nvidia 2070 Super con 8GB de memoria de video, la cual posee un CUDA compute capability de 7.5.

[2] Tomado de https://github.com/2prime/LM-ResNet

3.3 Resultados

En 1 se resumen los principales resultados de los experimentos realizados. Se indica la precisión obtenida (y su error como contraparte) junto con aquella reportada en el paper. Para que los números reportados sean más correctos, se podría haber tomado el promedio y desviación estándar de varias ejecuciones, pero por motivos de tiempo disponible, no fue posible. Se indica la diferencia entre los errores obtenidos y los reportados en el paper.

Vale aclarar que se usan "MResNet" y "LMResNet" intercambiablemente, el primero con base en el nombre dado en el código. Por otro lado, vemos la nomenclatura `LMResNet-X`, donde X indica la cantidad de capas convolucionales que tiene esa versión de la red.

	Replicación		**Paper**				
	Acc.	**Error (%)**	**Acc.**	**Error (%)**	**Diferencia de error**	**Parametros entrenables**	**Mult-Add totales (G)**
LMResNet-20	0.9200	8.00	0.9167	8.33	+0.33	273475	5.22
LMResNet-32	0.9204	7.96	0.9282	7.18	-0.78	467913	8.85
LMResNet-44	0.9279	7.21	0.9334	6.66	-0.55	662351	12.47
LMResNet-56	0.9315	6.85	0.9369	6.31	-0.54	856789	16.10
LMResNet-110	0.9357	6.43	0.9384	6.16	-0.27	1731760	32.40
LMResNet-164	0.9439	5.61	0.9473	5.27	-0.34	1703760	31.70

Tabla 1: Tabla de resumen de resultados obtenidos.

Un detalle que resulta confuso es que la cantidad de parámetros entrenables de ResNet110 no se condice con lo reportado en el paper (en este experimento dió 1.73M, vs 1.14M en el paper). La cantidad de Mult-Add da una idea del costo computacional de inferencia de cada modelo.

Los errores reportados dan *razonablemente* dentro de lo esperado, sin notar diferencias que sean irreconciliables con lo reportado en el paper. A finales de 2017, fecha de publicación del paper bajo estudio, el SOTA para CIFAR10 era[3] de 97.88 % de precisión [7], por lo que la diferencia en ese momento aún era de alrededor de 3.5 % de precisión.

En 10 se muestra la curva de precisión sobre el set de test para cada uno de los modelos entrenados. Notemos que al cambiar el learning rate, los valores se estabilizan y se observa lo que los autores proclaman: agregar más capas en efecto aumenta la precisión sin notarse cambios grandes de comportamiento en 11. En dicho gráfico se ven las curvas de pérdida para cada modelo durante entrenamiento, tanto para el set de pruebas como de entrenamiento. En 12 se ven las curvas en escala logarítmica para apreciar mejor la diferencia entre ambas curvas. Si bien se puede observar que la curva de pérdida de prueba aumenta luego del cambio de learning rate, su precisión mejora.

[3] https://paperswithcode.com/sota/image-classification-on-cifar-10

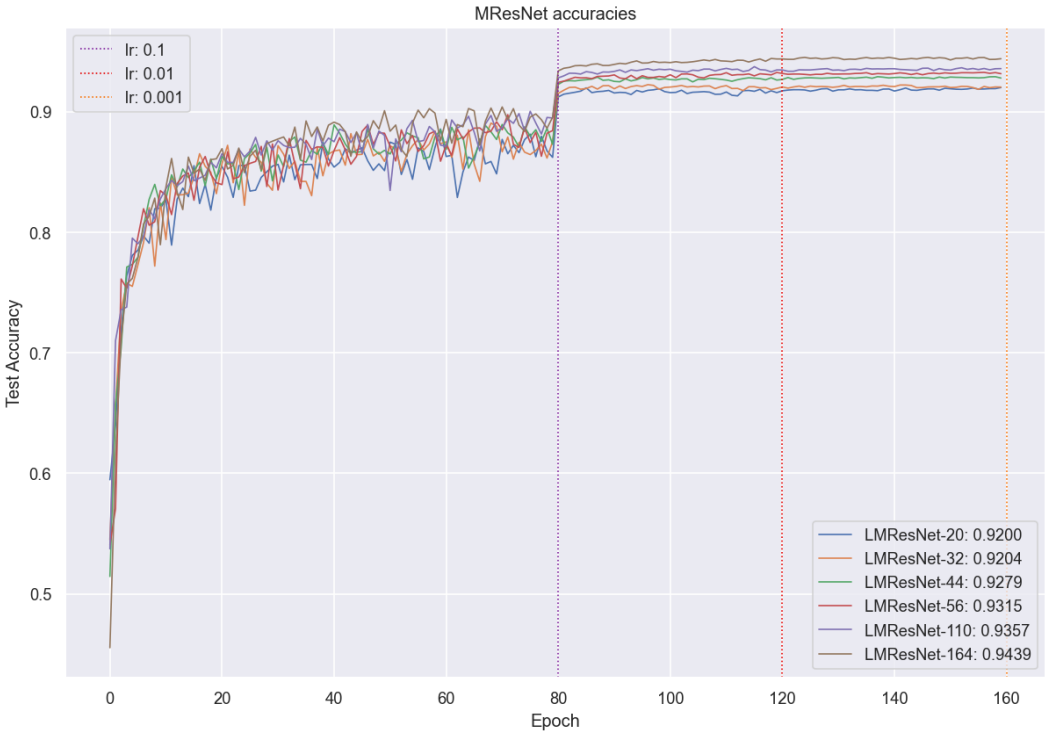

Figura 10: Accuracy sobre el set de test

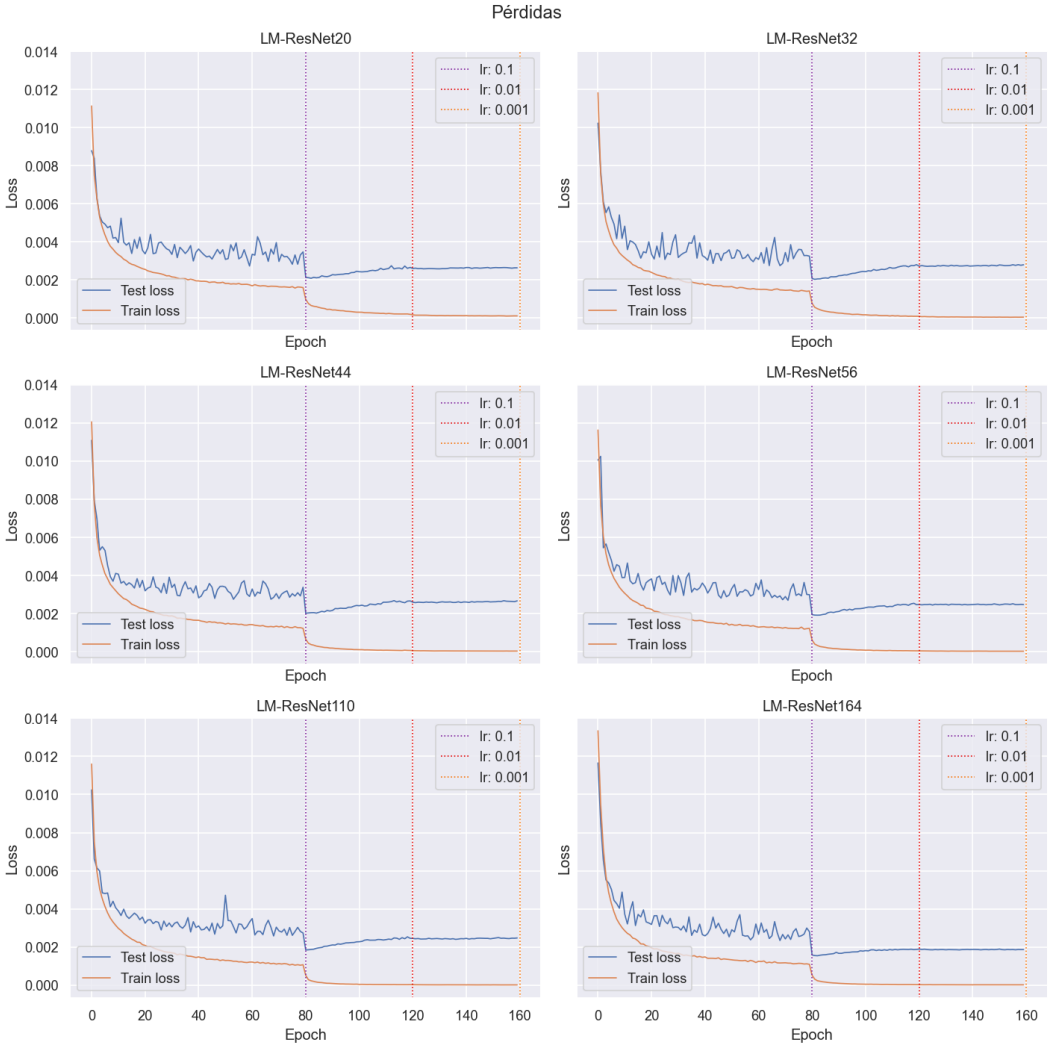

Figura 11: Pérdida (cross-entropy) de cada modelo durante entrenamiento

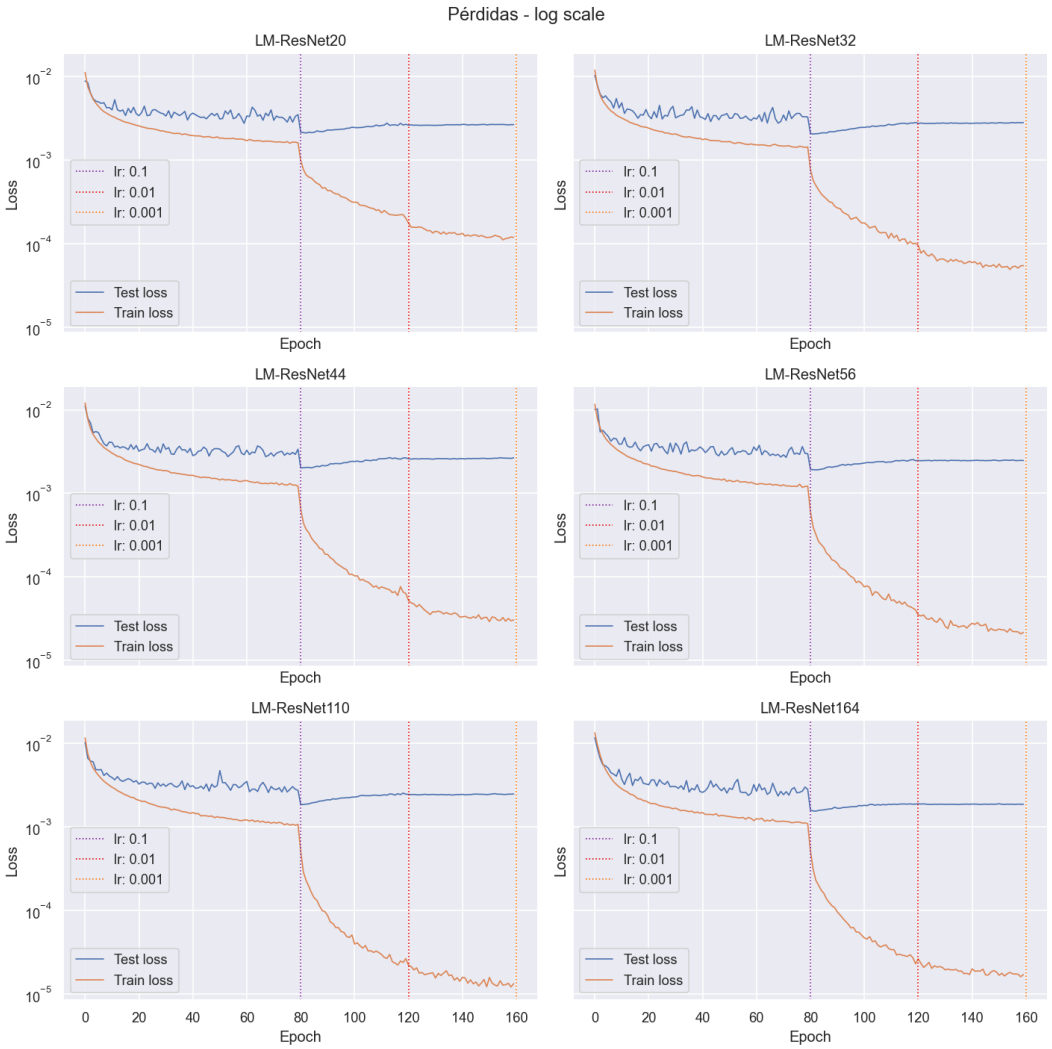

Figura 12: Pérdida (cross-entropy) de cada modelo durante entrenamiento. Escala logarítmica.

En 13 y 14 se ven los valores aprendidos por los parámetros k_n, los momentos de cada una de las capas residuales. Este punto es donde mayor discrepancias se encuentran respecto del paper original. En primer instancia, el paper proclama que estos valores se inicializan $\sim \mathscr{U}(-0.1, 0)$, sin embargo en el código original [4] los valores se inicializan en $\mathscr{U}(1.0, 1.1)$. Tal como se hizo antes, se toma lo explicado en el paper como verdad, y se modifica el código acorde. Notemos que esto, sin embargo, hace que ambas discrepancias tomen sentido: $1 - \mathscr{U}(1.0, 1.1) \sim \mathscr{U}(-0.1, 0.0)$. Sin embargo, por claridad, se prefiere dejar el código corregido para cumplir con lo descrito explícitamente. Luego, se observa algo que los autores notan: que los momentos aprendidos decaen rápidamente hacia el final de la red. La figura obtenida no tiene una caída tan suave como la reportada en el paper, sin embargo se cumple tanto para las redes más chicas como las redes más grandes. Los momentos reportados en el paper se distribuyen en $[-2.0, 1.0]$. En los experimentos de replicación, vemos que los valores se mantienen en $[-2.5, 1.0]$, lo cual parece bastante cercano.

Pensemos en qué implica un valor negativo para k_n. Revisando 24, vemos que se le está dando un valor positivo alto al residual actual y se le está dando un valor negativo al residual anterior. Podríamos justificarlo pensando que la información provista por el último punto da mucha más información que un punto anterior.

[4] https://github.com/2prime/LM-ResNet/blob/ff49654a50abd76f9fd40342a6d7b82e3ff11c31/MResNet.py#L139

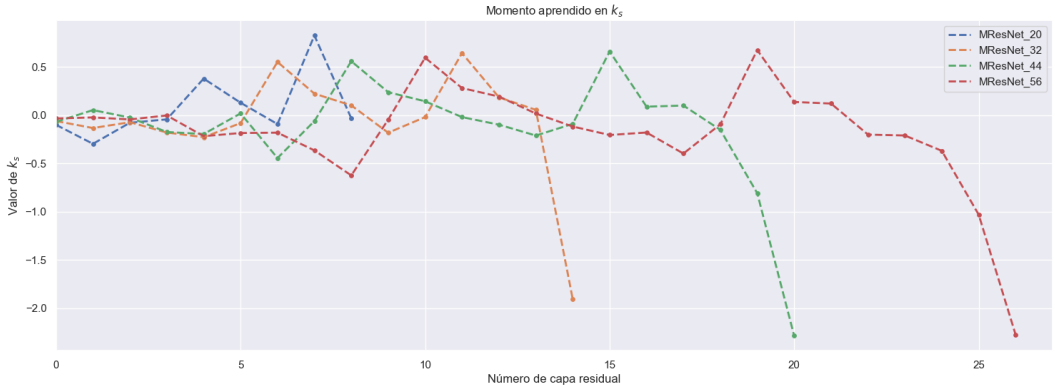

Figura 13: Momento aprendido en cada capa.

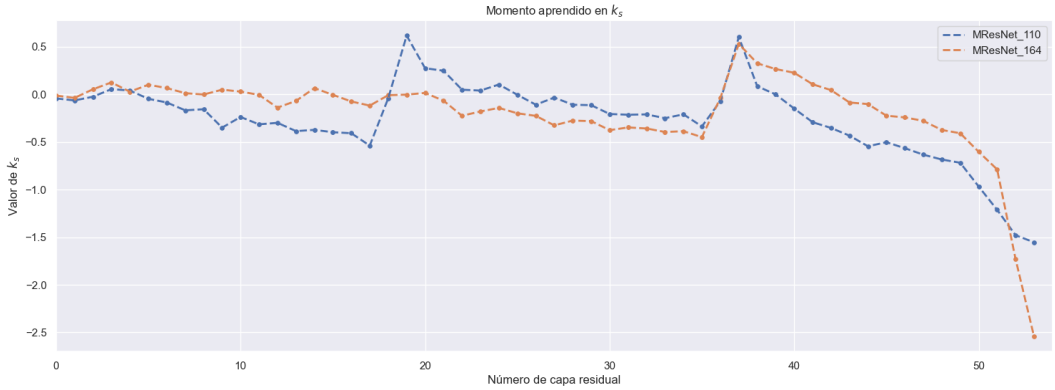

Figura 14: Momento aprendido en cada capa (modelos grandes).

4 Sistemas dinámicos estocásticos

En [8], se propone, aleatoriamente eliminar un subset de capas y reemplazarlas por la función identidad. Se inicia con una red extremadamente profunda (incluso más de 1200 capas) para luego ir eliminando algunas durante cada epoch de entrenamiento. Por otro lado, [3] propone utilizar regularización shake-shake para mejorar los resultados de una red con múltiples ramas (como LM-Resnet), reemplazando la operación de suma como unión de ramas por una operación afín estocástica.

4.1 Sistema dinámico de una regularización Shake-Shake

Tal como se adelantó, en [3] se propone la siguiente combinación afín estocástica:

$$X_{n+1} = X_n + \eta f_1(X_n) + (1-\eta) f_2(X_n) \tag{26}$$

Como operador de unión de múltiples componentes de un bloque residual. $\eta \sim \mathscr{U}(0,1)$. Introduciendo un paso temporal Δt encontramos su sistema dinámico estocástico asociado:

$$X_{n+1} = X_n + \left(\frac{\Delta t}{2} + \sqrt{\Delta t}(\eta - \frac{1}{2})\right) f_1(X_n) + \left(\frac{\Delta t}{2} + \sqrt{\Delta t}(\frac{1}{2} - \eta)\right) f_2(X_n) \tag{27}$$

Cuando $\Delta t \equiv 1$, la ecuación anterior se reduce a la regularización shake-shake, pero cuando se elige $\Delta t \neq 1$, entonces se tienen alternativas a la misma. En el paper se deriva, desde 27, que la red correspondiente a la regularización shake-shake es una aproximación débil del siguiente sistema dinámico estocástico:

$$dX = \frac{1}{2}(f_1(X) + f_2(X))dt \tag{28}$$

$$+ \frac{1}{\sqrt{12}}(f_1(X) - f_2(X)) \odot [\mathbb{1}_{N\times 1}, 0_{N,N-1}]\, dB_t \tag{29}$$

Donde dB_t es un vector de N dimensiones de movimiento Browniano.

4.2 Profundidad estocástica

En [8], como se adelantó, se propone eliminar bloques residuales aleatoriamente durante entrenamiento. Esto obedece a varios objetivos: en primer lugar, reducir el tiempo de entrenamiento. En segundo lugar, mejorar la robustez de la red. Es decir, actúa como método de regularización.

El paso hacia adelante de propagación de gradientes se vería como:

$$X_{n+1} = X_n + \eta_n f(X_n) \tag{30}$$

Donde $\mathbb{P}(\eta_n = 1) = p_n, \mathbb{P}(\eta_n = 0) = 1 - p_n$. Al introducir un paso temporal $\Delta t \equiv 1$, se deriva que la red con dropout estocástico se reduce a una aproximación débil del sistema estocástico dinámico

$$dX = p(t)f(X)dt + \sqrt{p(t)(1-p(t))}f(X) \odot [\mathbb{1}_{N\times 1}, 0_{N,N-1}]dB_t \tag{31}$$

4.3 Entrenamiento estocástico para LM-ResNet

En el paper se deriva la siguiente estrategia estocástica de entrenamiento para la arquitectura LM:

$$X_{n+1} = (2 + g_n)X_n - (1 + g_n)x_{n-1} + \eta_n f(X_n) \tag{32}$$

Donde $\mathbb{P}(\eta_n = 1) = p_n, \mathbb{P}(\eta_n = 0) = 1 - p_n$. Lo cual indicaría que se puede implementar simplemente eliminando el bloque residual con probabilidad p.

4.4 Resultados experimentales

4.4.1 Hiperparámetros

El dataset se trata del mismo modo que para experimentos anteriores. En este caso, la probabilidad de eliminar un bloque residual en cada capa es un función lineal de la capa. Siendo l la capa bajo consideración, la probabilidad de eliminar un bloque residual es $\frac{l}{L}(1 - p_L)$, donde L refiere a la profundidad de la red y p_L es la probabilidad asociada a la capa anterior a l.

Para LM-ResNet56 se selecciona $p_L = 0.8$ y para LM-ResNet110 se selecciona $p_L = 0.5$. Se entrena utilizando descenso estocástico por gradiente con decaimiento de peso de 0.0001 y momento de 0.9. El factor de aprendizaje arranca en 0.1, pasa a ser 0.01 luego del epoch 250 y 0.001 luego del epoch 375. El entrenamiento termina a los 500 epochs.

4.4.2 Resultados obtenidos

En 2 se resumen los principales resultados de los experimentos de entrenamiento estocástico. Estos experimentos, considerando que tienen un costo computacional más alto y entrenan por 3 veces más epochs que las anteriores pruebas, tardan mas que las pruebas para los 6 modelos sin entrenamiento estocástico. Se utiliza regularización dropout acorde a las pruebas reportadas en el paper. Se implementa la regularización shake-shake mediante la suma de ruido uniforme a la salida de cada bloque residual. Sin embargo, al no tener un valor de referencia en el paper, no se ejecutaron las pruebas.

Vemos que los valores de precisión obtenidos se corresponden con los reportados, dentro de márgenes de error aceptables para ambas configuraciones de la arquitectura LM-ResNet.

Resulta destacable que LM-ResNet56 con entrenamiento estocástico tiene un error menor que LM-ResNet110 y LM-ResNet164 sin entrenamiento estocástico, y con la mitad de parámetros. LM-ResNet110 con entrenamiento estocástico logra una mejora de 13.01 % error relativo respecto de LM-ResNet164 sin entrenamiento estocástico y **24.10 %** respecto de LM-ResNet110 sin entrenamiento estocástico.

	Replicación		**Paper**				
	Acc.	**Error (%)**	**Acc.**	**Error**	**Diferencia de error**	**Parámetros entrenables**	**Mult-Add totales (G)**
LMResNet-56	0.9440	5.60	0.9486	5.14	-0.46	856789	16.10
LMResNet-110	0.9512	4.88	0.9520	4.80	-0.08	1731760	32.40

Tabla 2: Tabla de resumen de resultados obtenidos para entrenamiento estocástico.

En 15 se muestra la curva de precisión sobre el set de test para ambos modelos entrenados con entrenamiento estocástico. Se observa un comportamiento similar al de experimentos anteriores donde al disminuir el learning rate se estabiliza la precisión. En 16 se observa la pérdida durante entrenamiento tanto para el set de test como de entrenamiento. Las diferencias enter ambas curvas se observan mejor en 17. En éste último gráfico se observa un fenómeno no observado previamente: la pérdida de entrenamiento comienza a subir antes de volver a disminuir el learning rate a 0.001, donde vuelve a bajar.

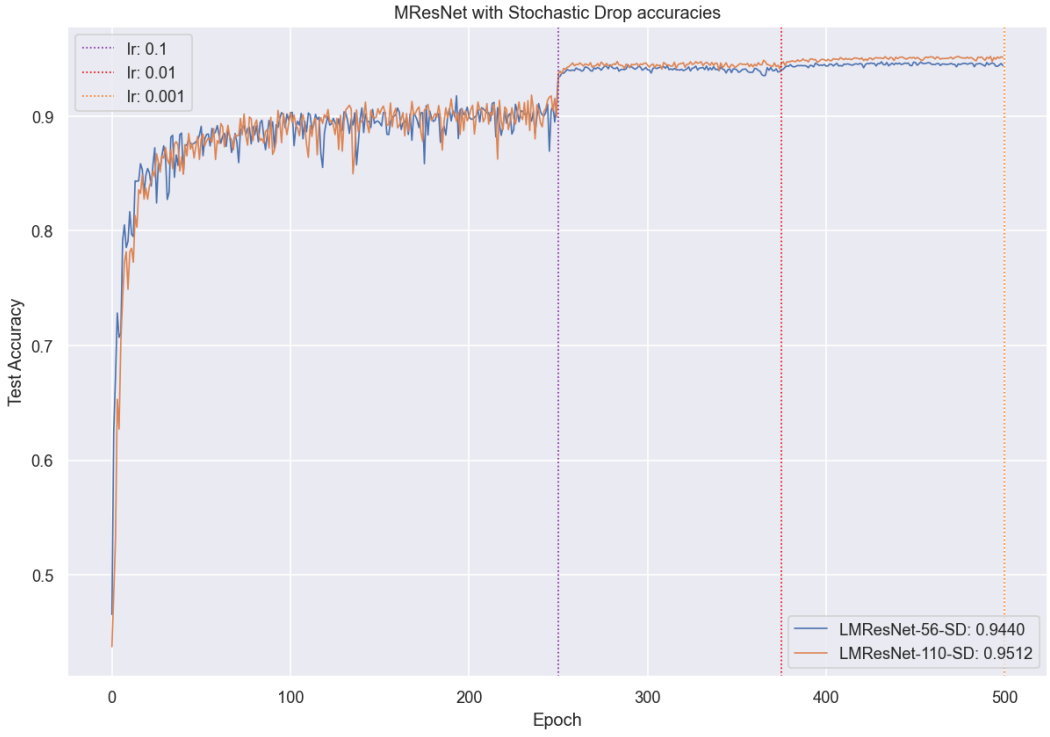

Figura 15: Accuracy sobre el set de test

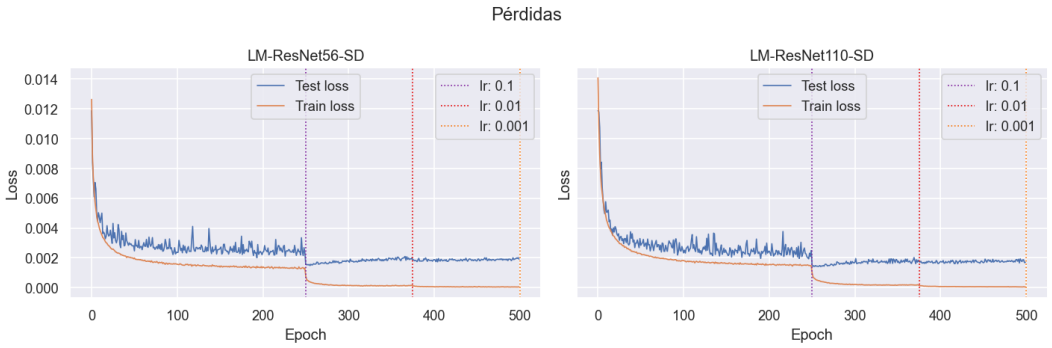

Figura 16: Pérdida (cross-entropy) de cada modelo durante entrenamiento

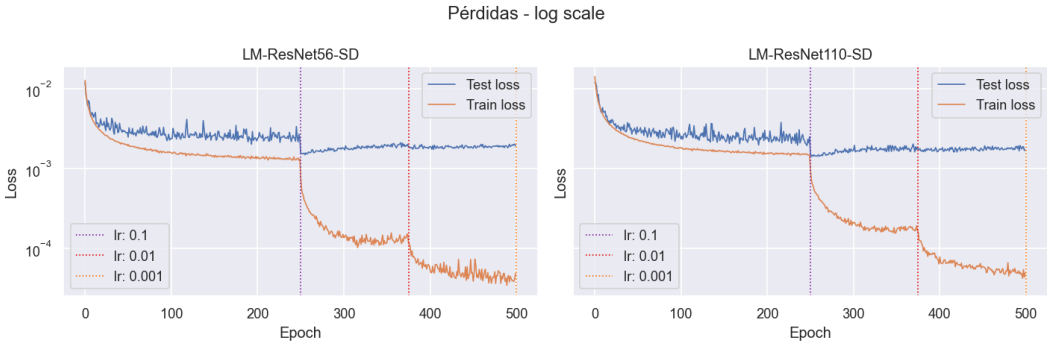

Figura 17: Pérdida (cross-entropy) de cada modelo durante entrenamiento. Escala logarítmica.

Tal como en experimentos anteriores, en 18 se observa que el comportamiento de los valores k_n aprendidos para cada capa residual se acelera rápidamente hacia el final de la red.

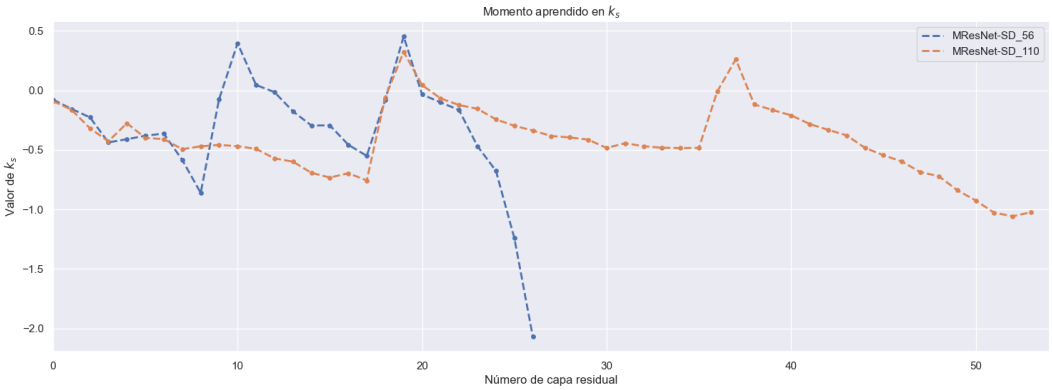

Figura 18: Momento aprendido en cada capa .

5 Una nota sobre el método de ecuaciones modificadas

La ecuación modificada de un esquema de análisis numérico que aproxima un sistema de ecuaciones por diferencias es otra ecuación diferencial que puede ser mejor aproximado por el esquema numérico. Se utilizan para describir las propiedades de dichos esquemas. La conclusión relevante del paper es que cuando f, la función aproximada por el esquema de Euler de primer orden de ResNet, es un flujo de gradientes, la ecuación de diferencias de LM-ResNet tiene una condición de estabilidad $-1 \leqslant k_n \leqslant 1$.

6 Conclusión

En el presente trabajo se explica el paper "Beyond Finite Layer Neural Networks", haciendo un repaso del estado del arte previo, su motivación y los temas más relevantes de la literatura relacionada. Se implementan los pipelines de entrenamientos necesarios para replicar los resultados, se reporta la diferencia obtenida respecto de los resultados mencionados en el paper y se hacen notar varios problemas en el paper original y el código.

Considero que los objetivos del trabajo han sido cumplidos, sin embargo, se abren las puertas a varias líneas de mejora sucesivas.

6.1 Trabajo futuro propuesto

Un camino que no se exploró es el de replicar los resultados para CIFAR100 e ImageNet. Esto se debió a los tiempos de entrenamiento necesarios, pero principalmente debido a que no parecía aportar demasiado respecto de las pruebas realizadas. Se propone también hacer una revisión sobre los valores de k_n para ver si estos están siendo considerados durante la backpropagation de los gradientes.

En la página de SOTA de CIFAR10 de Papers with Code se ve que desde la publicación del paper objeto de estudio del presente informe, el estado del arte movió el mejor valor reportado de 97.02 % a 99.5 % en 2021. Desde entonces, y hasta la fecha en 2023, no se han reportado mejores resultados.

Hay dos hechos notables que pienso que ofrecen una oportunidad interesante de investigación. Por un lado, [9] introduce BiT, una arquitectura basada en ResNet-v2 que marca un nuevo récord de precisión sobre CIFAR10. Este récord se mantiene desde 2020 hasta que [1] introduce ViT-H/14 en 2021. Esta nueva arquitectura está basada en [16] la arquitectura Transformer, en la cual se basan las arquitecturas de mejor performance a la fecha, en diferentes dominios como NLP y Computer Vision.

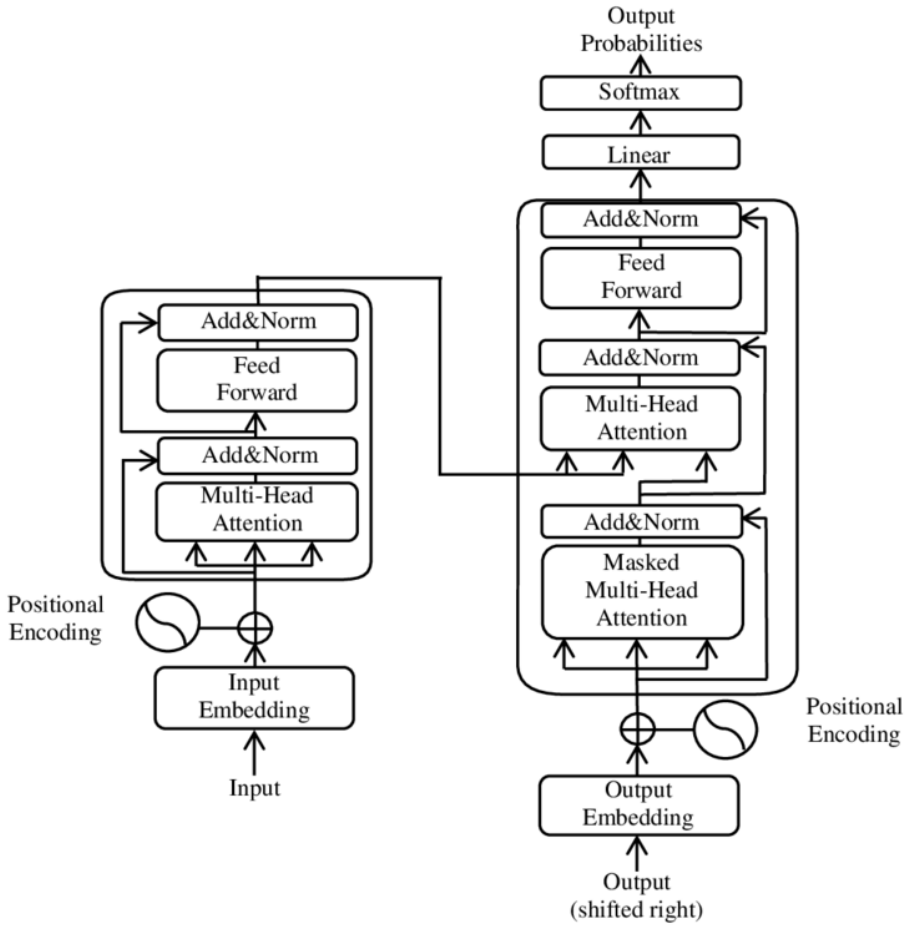

Figura 19: Bloque de transformer

En primer lugar, propongo adaptar la arquitectura LM a BiT y evaluar si los resultados obtenidos mejoran respecto de la publicación original. Por otro lado, si observamos en 19, vemos que tanto el encoder como el decoder de la arquitectura de Transformer poseen conexiones residuales, lo cual lleva a pensar que quizás se puedan aplicar los mismos principios aquí estudiados. Sin embargo, resulta más interesante que [13], con primer autor al mismo autor del paper objecto de estudio del presente informe, estudia a la arquitectura de Transformers como un método de resolución de EDOs para una ecuación se convección-difusión en un sistema dinámico de múltiples partículas. Propongo evaluar agregar este paper a la lista de papers posibles sobre los cuales realizar una monografía dentro del marco de la materia.

Referencias

[1] Alexey Dosovitskiy y col. *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. 2021. arXiv: 2010.11929 [cs.CV].

[2] Weinan E. «A Proposal on Machine Learning via Dynamical Systems». En: *Communications in Mathematics and Statistics* 5 (1 2017), págs. 1-11. ISSN: 2194-6701,2194-671X. DOI: 10.1007/s40304-017-0103-z. URL: http://doi.org/10.1007/s40304-017-0103-z.

[3] Xavier Gastaldi. *Shake-Shake regularization*. 2017. arXiv: 1705.07485 [cs.LG].

[4] Aidan N. Gomez y col. *The Reversible Residual Network: Backpropagation Without Storing Activations*. 2017. arXiv: 1707.04585 [cs.CV].

[5] Kaiming He y col. «Deep Residual Learning for Image Recognition». En: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[6] Kaiming He y col. *Identity Mappings in Deep Residual Networks*. 2016. arXiv: 1603.05027 [cs.CV].

[7] Jie Hu y col. *Squeeze-and-Excitation Networks*. 2019. arXiv: 1709.01507 [cs.CV].

[8] Gao Huang y col. *Deep Networks with Stochastic Depth*. 2016. arXiv: 1603.09382 [cs.LG].

[9] Alexander Kolesnikov y col. *Big Transfer (BiT): General Visual Representation Learning*. 2020. arXiv: 1912.11370 [cs.CV].

[10] Alex Krizhevsky. «One weird trick for parallelizing convolutional neural networks». En: *CoRR* abs/1404.5997 (2014). arXiv: 1404.5997. URL: http://arxiv.org/abs/1404.5997.

[11] Gustav Larsson, Michael Maire y Gregory Shakhnarovich. *FractalNet: Ultra-Deep Neural Networks without Residuals*. 2017. arXiv: 1605.07648 [cs.CV].

[12] Yiping Lu y col. «Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations». En: *Proceedings of the 35th International Conference on Machine Learning*. Ed. por Jennifer Dy y Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. Stockholmsmässan, Stockholm Sweden: PMLR, oct. de 2018, págs. 3282-3291. URL: http://proceedings.mlr.press/v80/lu18d.html.

[13] Yiping Lu y col. *Understanding and Improving Transformer From a Multi-Particle Dynamic System Point of View*. 2019. arXiv: 1906.02762 [cs.LG].

[14] Rupesh Kumar Srivastava, Klaus Greff y Jürgen Schmidhuber. *Highway Networks*. 2015. arXiv: 1505.00387 [cs.LG].

[15] Christian Szegedy y col. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV].

[16] Ashish Vaswani y col. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL].

[17] Xingcheng Zhang y col. *PolyNet: A Pursuit of Structural Diversity in Very Deep Networks*. 2017. arXiv: 1611.05725 [cs.CV].