Departamento de Ingeniería Eléctrica
Facultad de Ciencias Físicas y Matemáticas
Universidad de Chile
EL7008 Procesamiento Avanzado de Imágenes

Proyecto EL7008

Detección de Rostros basado en *Deep Learning*

Integrantes:
Diego Irarrázaval I.

Profesor:
Javier Ruiz del Solar.

Auxiliar:
Patricio Loncomilla Z.

Ayudante:
Hans Starke D.

Fecha:
23 de enero de 2021

Índice

Índice de figuras

Índice de tablas

Índice de Algoritmos

1. Introducción

El presente informe contiene los modelos y procedimientos realizados para implementar un modelo basado en *Deep Learning* para resolver la tarea de detección de rostros. Se utilizo la red neuronal **RetinaFace** [6] con distintos *backbone* de extractores de caracteristicas: **ResNet** [3], **Res2Net** [7] y **EfficientNet** (version **b5**)[8]. De la misma forma, se probaron distintos optimizadores como **AdaBound**, introducido en [2] y **SDGP**, introducido en [5].

Para los entrenamientos, se utilizó la supercomutadora **DGX-1**[9]. Este computador cuenta con 8 **GPU's** de 32$[GB]$ de memoria RAM cada uno. Está especialmente diseñada para entrenamiento de modelos de inteligencia artificial y computación distribuída.

La tarea que el modelo debe completar es, como se mencionó anteriormente, la detección de rostros en una imagen. La imagen a continuación muestra lo que se espera que haga la red:

Figura 1: Tarea a cumplir por la red. Imagen obtenida de [4].

La implementación del proyecto implica modificar la configuración del repositorio original (en [4]) modificando el optimizador y el *backbone*. A continuación se explicará brevemente que es el backbone y el optimizador:

1.1. Backbone:

Hace referencia al modelo que se utilizará para extraer características de las imágenes. Usualmente, corresponde a una secuencia de redes convolucionales (con otras capas de regularización) que son las encargadas de aprender representaciones de las imágenes para luego entregar estos resultados a las siguientes capas clasificadoras.

Al utilizar como extractor de características, se pueden obtener representaciones de muchas dimensiones y combinaciones de estas, al extraer el *output* de capas intermedias. Esto es lo que se realiza en la red **Pytorch Retinaface**.

1.2. Optimizador:

Al entrenar una red neuronal, se busca optimizar la pérdida en función de los parámetros. Digamos que tenemos $L(\hat{y}, y)$, la función de pérdida donde y corresponde a la etiqueta real y $\hat{y}$ corresponde a la predicción que es una función que depende de todos los parámetros de la red. Entonces, el objetivo del optimizador es encontrar lo siguiente:

$$\underset{\theta}{argmin}\, L(\theta), \tag{1}$$

Donde θ corresponde a todos los parámetros del modelo o red neuronal.

Esta tarea es computacionalmente compleja debido a que los parámetros, θ, en las redes modernas son decenas de millones e incluso más dependiendo de la arquitectura usada. Para solucionar esto, los optimizadores utilizan heurísticas y distintos métodos para encontrar el punto óptimo de los parámetros y disminuir la función de pérdida.

En la implementación, se probaron 4 combinaciones distintas para verificar como afecta el *backbone* y el optimizador en los resultados de la detección de rostros.

En las siguientes secciones, se introducen los *backbone* y optimizadores utilizados, luego se comenta sobre el proceso de entrenamiento y los resultados obtenidos, para concluir con los principales aprendizajes y resultados.

2. Metodología

Como ya se introdujeron los conceptos más importantes en la sección anterior, a continuación se hará una breve descripción de cada modelo y optimizador utilizado, seguido de una descripción del procedimiento general:

2.1. Optimizadores:

Se asumirá en el informe que ya hay conocimiento de los optimizadores **SDG** y **Adam**.

2.1.1. SAM [1]: *Sharpness-Aware Minimization for Efficiently Improving Generalization*

A pesar de que este optimizador obtuvo pésimos resultados utilizando la red ResNet50 y hace que el proceso de entrenamiento sea considerablemente más lento, se menciona debido a que es interesante lo que se busca hacer: Busca aumentar la capacidad de generalización de la red minimizando la función de pérdida y, al mismo tiempo, intentando hacer que el punto óptimo sea menos 'puntiagudo':

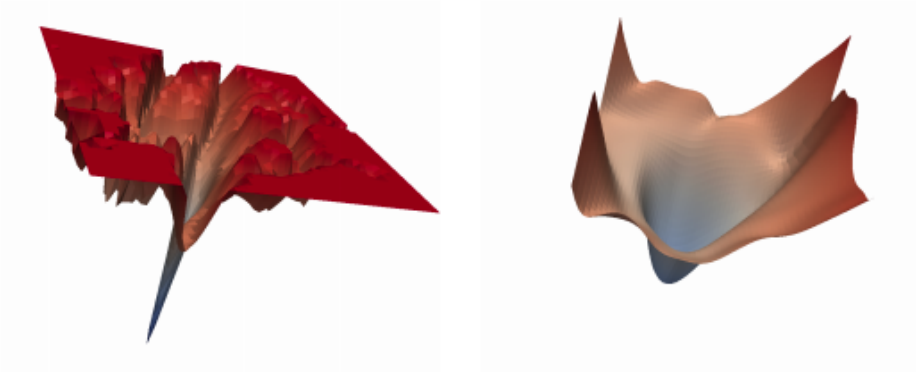


Figura 2: En la imagen de la derecha, la representación del óptimo encontrado con **SDG** y a la derecha, el encotrado con **SAM** para la red neuronal ResNet. [1]

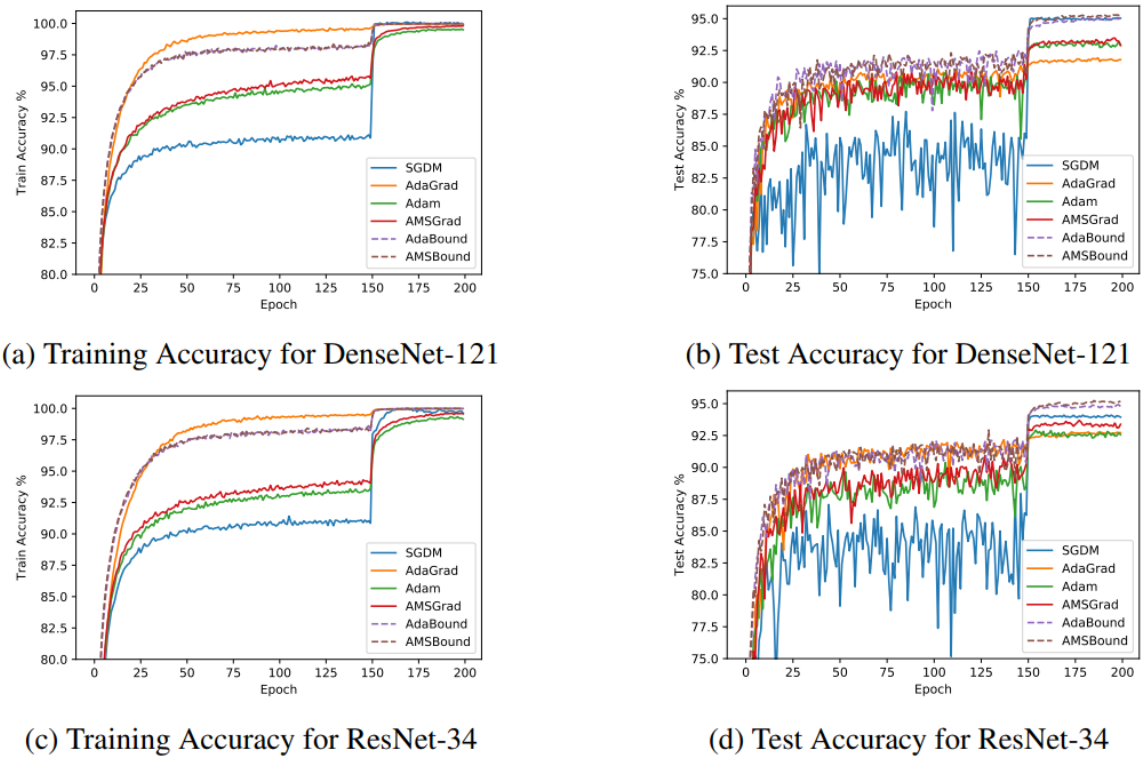
En la figura, se observa que al utilizar **SAM** se puede obtener una vecindad de parámetros

que tengan buenos resultados y minimicen la función de pérdida, haciendo que el modelo generalice mejor.

2.1.2. AdaBound: [2]

Este optimizador intenta juntar las ventajas de los optimizadores adaptativos, la velocidad de convergencia, con los métodos de gradiente estocásticos, la capacidad de generalización. La intuición detrás del método es que inicialmente, el optimizador tenga características de los adaptativos (la velocidad) y que al final del entrenamiento mejore la capacidad de generalización.

A continuación, se muestran resultados expuestos en el paper:



(a) Training Accuracy for DenseNet-121

(b) Test Accuracy for DenseNet-121

(c) Training Accuracy for ResNet-34

(d) Test Accuracy for ResNet-34

Figura 3: *Accuracy* en entrenamiento y test para DenseNet-121 y ResNet-34 en el dataset CIFAR-10. [2]

En la figura 3, se observa una rápida convergencia, a la par de los optimizadores Adam y, por otro lado, se observa que es consistente en entrenamiento y evaluación, lo que indica que

generaliza bien.

2.1.3. SDGP

El paper [5] introduce dos optimizadores: **Adamp** y **SDGP**. En este trabajo, se utilizó **SDGP** debido a que usualmente, los optimizadores de gradiente estocástico generalizan mejor.

Los optimizadores propuestos, aunque un 8 % más lentos que los normales, muestran mejoras significativas en comparación a los clásicos Adam y SDG:

Architecture	# params	SGD	SGDP (ours)	Adam	AdamW	AdamP (ours)
MobileNetV2	3.5M	71.55	**72.09** (+0.54)	69.32	71.21	**72.45** (+1.24)
ResNet18	11.7M	70.47	**70.70** (+0.23)	68.05	70.39	**70.82** (+0.43)
ResNet50	25.6M	76.57	**76.66** (+0.09)	71.87	76.54	**76.92** (+0.38)
ResNet50 + CutMix	25.6M	77.69	**77.77** (+0.08)	76.35	78.04	**78.22** (+0.18)

Figura 4: Tabla con resultados de clasificación en **ImageNet**.[5]

En los resultados, se observa una mejora sólo cambiando el optimizador. Esto puede ser importante en sistemas que no tienen capacidad para aumentar la cantidad de parámetros de una red, por ejemplo.

2.2. *Backbone*:

Los *backbone* utilizados se introducen a continuación

2.2.1. ResNet[3]

Esta arquitectura, se introdujo el año 2015 con un cambio de paradigmas en las redes convolucionales: Proponen entregar información de capas y representaciones anteriores a las capas más profundas. Un bloque *residual* básico tiene la siguiente forma:

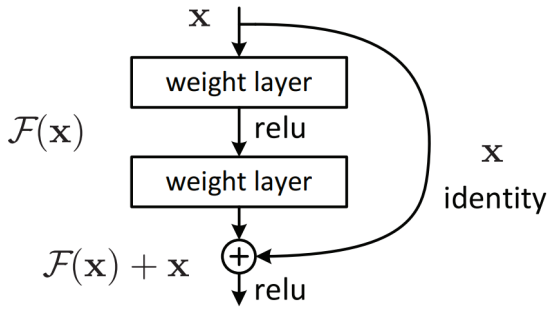


Figura 5: Bloque *residual* básico. Obtenido de [3].

Esta arquitectura, el año que fue propuesta (2015), opcupó el primer lugar en las más importantes competencias de Aprendizaje de Máquinas: *ILSVRC 2015, ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation.*

Esta arquitectura presenta muchas variantes dependiendo de la cantidad de capas con las que se implemente. Para el proyecto, se eligió la red **ResNet-50**. Se muestran en la siguiente tabla las distintas arquitecturas y la cantidad de parámetros:

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figura 6: Arquitecturas propuestas en [3] con la cantidad de parámetros.

2.2.2. Res2Net

El paper [7] propone la utilización de capas residuales como extractor de características en redes profundas. La intuición era incluir representaciones multi-escalas obtenidas de las distintas

capas residuales. Adicionalmente, se propone un cambio al bloque residual que se muestra en la siguiente imagen:

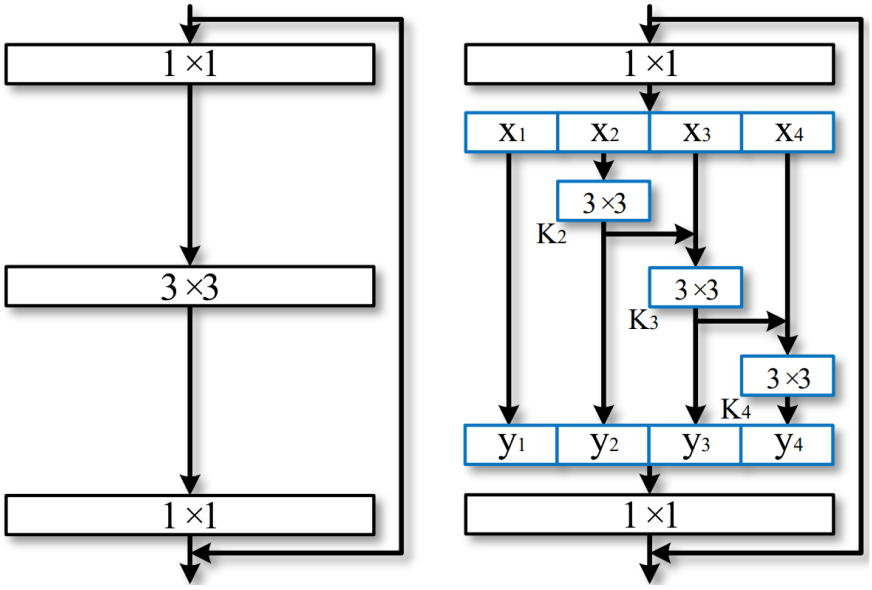


Figura 7: En la izquierda, el bloque *residual* original y a la derecha el propuesto en el paper.

Esta arquitectura, significa una mejora de al menos un punto porcentual (un 1 % mejor) en *ImageNet* en comparación con su antecesor, **ResNet**. En particular, para este proyecto se utilizó en su version **Res2Net-101 26w,4s**.

2.2.3. EfficientNet

La arquitectura (en realidad familia de arquitecturas) propuesta en [8], responde a la práctica común de escalar y aumentar (la profundidad por ejemplo) los parámetros y capas en la medida que haya recursos computacionales disponibles para mejorar el desempeño. En lugar de simplemente hacer las redes más profundas, o aumentar la cantidad de filtros, o mejorar la resolución de la imagen, proponen un escalamiento *compound*: escalar distintos parámetros de las redes de forma simultánea:

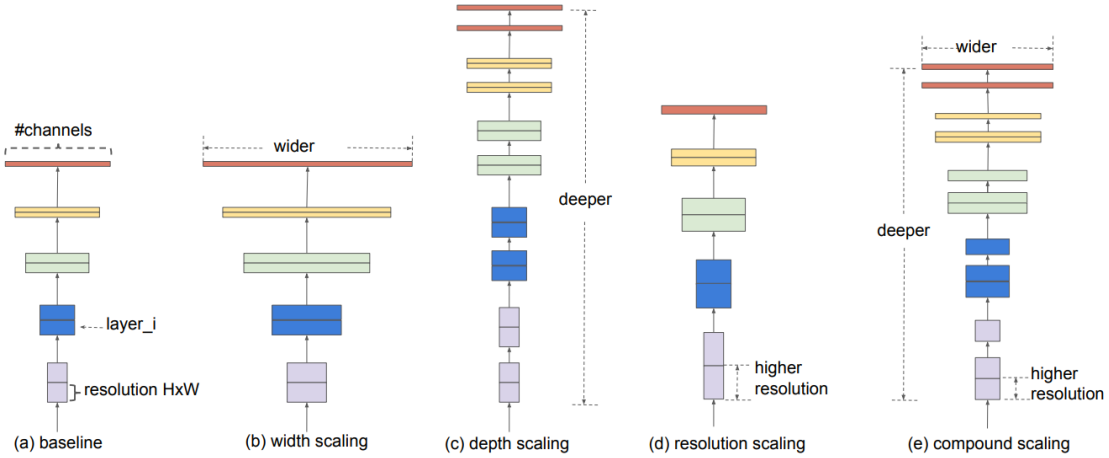


Figura 8: Escalamiento de parámetros: las figuras (a)-(d) corresponden a los métodos clásicos de escalamiento. La forma propuesta se ilustra en la figura (e).

Adicionalmente, proponen una arquitectura donde el principal componente son los bloque **MBConv**, propuestos en [10]. Estas arquitecturas tienen pocos parámetros en comparación a redes con desempeño similar y lograron el estado del arte en el dataset *ImageNet*, con una red 8,4x veces más pequeña. A continuación, se muestra una tabla con distintas arquitecturas de la **EfficientNet** en comparación con otras redes:

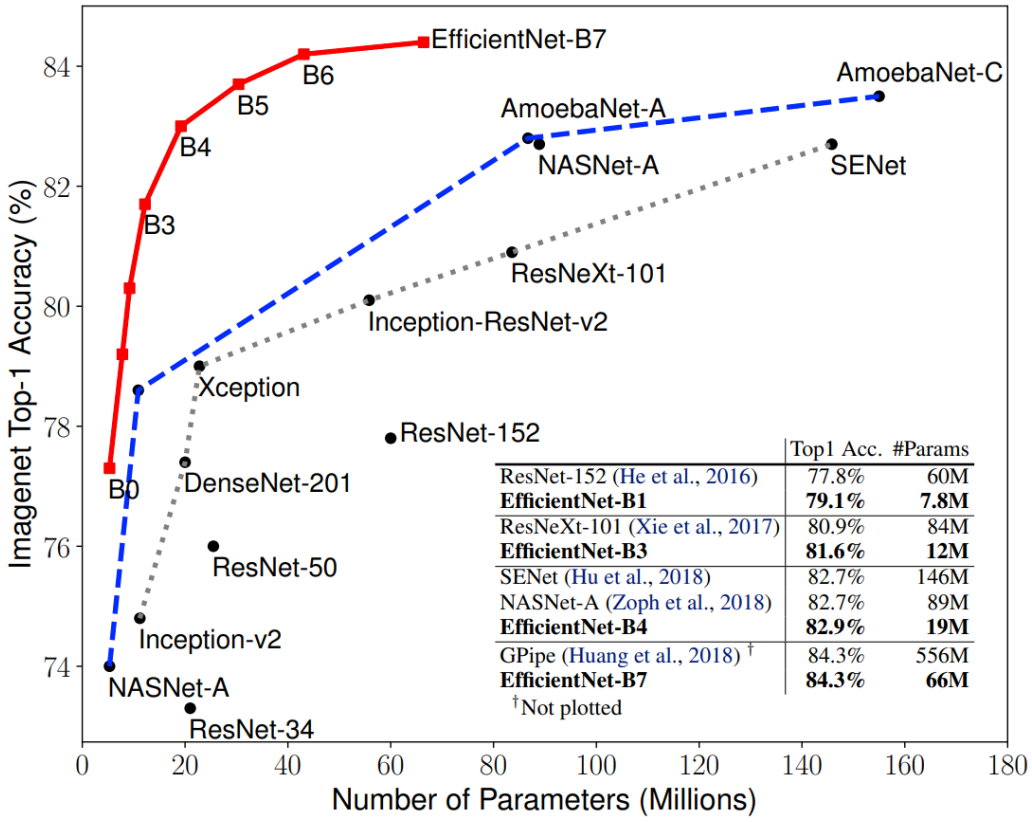


Figura 9: *Accuracy* en función de la cantidad de parámetros. La línea representa las arquitecturas **EfficientNet**.

En la siguiente sección, se comenta sobre la implementación, entrenamiento y resultados obtenidos.

3. Desarrollo

3.1. Entrenamiento

La implementación de la red **Pytorch-Retinaface** se basó en el repositorio de github ([4]) entregado por el ayudante. De esta implementación, se utilizó la arquitectura **ResNet-50**, con el extractor de *backbone* que tiene implementado el repositorio.

Código adicional se implementó para los *backbone*:

- **Res2Net**, basado en [11].
- **EfficientNet-b5**, utilizando la librería Timm, que contiene muchas implementacions de redes neuronales con modelos preentrenados y opcion de extraer solo caracteristicas.

Por otro lado, los optimizadores adicionales se obtuvieron de:

- **AdaBoundW**, basado en [12]. Corresponde a **AdaBound** con decaimiento de pesos (*Decoupled Weight Decay*).
- **SDGP**, utilizando la librería **Adamp** [13].

Todo lo anterior, se encuentra en un repositorio que se creó a partir de [4] y se agregaron los módulos necesarios: Diego-II/Pytorch_Retinaface.

3.2. Resultados y comentarios

Las combinaciones utilizadas se muestran junto con los resultados:

Modelo	Optimizador	Hiperparámetros	Resultados Easy	Resultados Medium	Resultados Hard
ResNet-50	SDG	150 épocas, bs 32	91.90 %	88.80 %	77.01 %
ResNet-50	AdaBoundW	150 épocas, bs 32	94.90 %	93.58 %	85.89 %
Res2Net-101	AdaBoundW	150 épocas, bs 16	95.60 %	93.62 %	86.01 %
EfficientNet-b5	SDGP	80 épocas, bs 12	00.12 %	00.19 %	00.15 %

Tabla 1: Modelos utilizados junto con los hiperparámetros del entrenamiento y los resultados obtenidos.

Cabe destacar que tanto en el tercer como cuarto modelo se ha disminuido el tamaño del *batch* debido a que se excedía la capacidad de la memoria al usar uno mayor.

Por otro lado, la arquitectura **EfficientNet-b5** es de entrenamiento mucho más lento. Tardaba en 80 épocas lo mismo que las demás arquitecturas en 150. Adicionalmente, tenía el mismo problema de memoria por lo que el tamaño **batch** se redujo a 12. **Importante:** Debido a temas de tiempo, se debió terminar el entrenamiento en la época 74, haciendo la evaluación con el *checkpoint* de la época 70. Tabmíen se debe mencionar que en la evaluación, se demoró más de 7 minutos por tarea (*easy*, *medium*, *hard*).

Lo primero a destacar es, la importante mejora que muestra la red **ResNet-50** al cambiar el optimizador. Significó un aumento de 3 puntos en el peor de los casos (*easy*) y en *hard*, mejoró sobre un 8 % adicional. Esto es importante considerando que no hubo cambio alguno en la arquitectura ni el uso de memoria. Si aumentó levemente el tiempo de entrenamiento de la red.

Por otro lado, se observa que aunque no aumenta de forma considerable el desempeño, **Res2Net** si obtiene mejores resultados que **ResNet-50**, aunque ayudado por el optimizador **AdaboundW**. Los resultados obtenidos en *easy* superan los obtenidos por el repositorio original (94,97 %).

Cabe destacar que se obtiene apenas un 1 % adicional agregando cerca de 20 Millones de parámetros a la red (diferencia entre Res2Net-101 y ResNet-50).

Sobre los resultados obtenidos con la **EfficientNet**, claramente no son los esperados ya que de las arquitecturas elegidas esta es la que presenta mejores resultados por ejemplo en el dataset *ImageNet*. Se piensa que se debe a una mala conexión entre el *backbone* y la siguiente capa **FPN** de la red **RetinaFace**.

4. Conclusiones y trabajos futuros

En primer lugar, se aprendió a utilizar *hardware* dedicado para inteligencia artificial. Hubo muchas dificultades inicialmente que terminaron con un proceso de entrenamiento completo (1 día y 5 hrs) en tiempo perdido. Luego de superar estas trabas, se pudo trabajar con normalidad, entrenar y obtener resultados.

Se aprendió sobre ambientes *Linux*, **Docker** y sobre cómo utilizar un *screen* para mantener vivos los procesos cuando se hacen conexiones por SSH.

Por el lado de los resultados, en primer lugar se destaca la importancia de la elección del **optimizador**. Sólo cambiando éste se obtuvo una mejora de un 8 % en las pruebas *hard*. Esto muestra el poder que tienen éstos para aproximar de mejor forma los parámetros de la red.

Se observa que no siempre vale la pena aumentar la profundidad de la red y la cantidad de parámetros: La red **Res2Net-101** aumentó en 20 Millones la cantidad de parámetros en comparación a **ResNet-50** y se aumentó, en el mejor de los casos, un punto porcentual el desempeño en las tareas.

Sobre los resultados obtenidos con **EfficientNet** se puede concluir sobre la importancia de investigar a profundidad como implementar de manera correcta este tipo de arquitecturas que incluyen un *backbone*.

Sobre los trabajos futuros, se propone incluir el optimizador **SAM**, que en realidad funciona como un *wrapper* para el optimizador base elegido. Por otro lado, se intentó de utilizar como *backbone* varias arquitecturas: **TResNet**, **EfficientNet** (desde b0 hasta b5), otras pero no se logró conectar bien con **RetinaFace** o los resultados eran incluso peores que los mostrados en la tabla 1. Se propone un trabajo de investigación más profundo para poder implementar esta arquitectura con mayor variedad de *backbones*.

Bibliografía

[1] Pierre Foret, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. Sharpness-Aware Minimization for Efficiently Improving Generalization. 2020.

[2] Liangchen Luo, Yuanhao Xiong, Yan Liu, and Xu Sun. Adaptive gradient methods with dynamic bound of learning rate. *CoRR*, abs/1902.09843, 2019.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[4] biubug6. Retinaface in pytorch. https://github.com/biubug6/Pytorch_Retinaface.

[5] Byeongho Heo, Sanghyuk Chun, Seong Joon Oh, Dongyoon Han, Sangdoo Yun, Gyuwan Kim, Youngjung Uh, and Jung-Woo Ha. Adamp: Slowing down the slowdown for momentum optimizers on scale-invariant weights, 2021.

[6] Jiankang Deng, Jia Guo, Yuxiang Zhou, Jinke Yu, Irene Kotsia, and Stefanos Zafeiriou. Retinaface: Single-stage dense face localisation in the wild. *CoRR*, abs/1905.00641, 2019.

[7] Shanghua Gao, Ming-Ming Cheng, Kai Zhao, Xinyu Zhang, Ming-Hsuan Yang, and Philip H. S. Torr. Res2net: A new multi-scale backbone architecture. *CoRR*, abs/1904.01169, 2019.

[8] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *CoRR*, abs/1905.11946, 2019.

[9] NVIDIA. Dgx-1. https://www.nvidia.com/es-la/data-center/dgx-1/.

[10] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018.

[11] Res2Net. Res2net pretrained models. https://github.com/Res2Net/Res2Net-PretrainedModels.

[12] Luolc. Adabound. https://github.com/Luolc/AdaBound.

[13] AdamP. Adamp: Slowing down the slowdown for momentum optimizers on scale-invariant weights. https://github.com/clovaai/adamp.