Tecnología Digital VI: Inteligencia Artificial

Olivier Saint-Nom, Valentina de Diego, Lucas Piñera

Trabajo Práctico 3

Experimentos para la Optimización de una Red Neuronal en CIFAR-10

Esta investigación tiene como objetivo explorar y evaluar una variedad de hiperparámetros en el contexto del entrenamiento de redes neuronales para la clasificación de imágenes utilizando el conjunto de datos CIFAR-10. El objetivo es comprender cómo diferentes configuraciones afectan el rendimiento de modelos de deep learning y cuáles son las estrategias más efectivas para optimizar los modelos. A través de una serie de aspectos avanzados como arquitectura, funciones de activación, optimizadores, regularización y otros, buscamos identificar las mejores configuraciones que maximizan la precisión de la clasificación. A lo largo de este informe, presentaremos los resultados de nuestros experimentos.

El informe se detallará en las siguientes etapas:

1. Configuración

Para comenzar, realizamos una división del 80% para entrenamiento y 20% para validación. Esto resultó en la creación de conjuntos de datos de entrenamiento y validación, que luego se dividieron en lotes de tamaño batch_size para su procesamiento durante el entrenamiento. Registramos tanto la pérdida (*loss*) y precisión (*accuracy*) en el conjunto de entrenamiento como en el conjunto de validación en cada iteración durante el proceso de entrenamiento para evaluar el rendimiento del modelo en tiempo real a lo largo de toda la exploración. En estas iteraciones también registramos los valores de los hiperparametros utilizados para mantener registro de los experimentos y llegar a conclusiones claras sobre nuestros enfoques.

A lo largo de todo el experimento, adoptamos un enfoque parecido al método científico, donde identificamos y aislamos las diferentes modificaciones como variables independientes, manteniendo las otras condiciones constantes. Este enfoque nos permitió evaluar precisamente cuál de las modificaciones generaba las mejores mejoras en nuestro modelo de clasificación.

2. Experimentos con Arquitectura Densa

En esta etapa, decidimos partir de las arquitecturas proporcionadas en clase, y de ahí poder ir modificando la cantidad de capas densas, nodos, hidden layers e ir evaluando y aprendiendo cual es el mejor experimento para nuestro objetivo. Estos fueron los siguientes enfoques que fuimos probando:

- "**`MLP`**": Comenzamos con un modelo base MLP que consta de múltiples capas densas, cada una con una cantidad ‘moderada’ de nodos. El modelo se encarga de la clasificación de imágenes y termina con 10 nodos de salida, correspondientes a las 10 clases de objetos en nuestro conjunto de datos.

- "**`fc_hl_bigvalues`**": En este experimento, aumentamos el número de nodos en algunas de las capas densas, lo que hace que el modelo sea más complejo.

Específicamente, la capa oculta intermedia la aumentamos de 120 a 150 nodos, y se mantienen capas adicionales para procesar los datos antes de la capa de salida.

- "**`fc_hl_1layers`**": En este caso, decidimos experimentar al tener una sola capa oculta con 120 nodos. Esto reduce la complejidad del modelo en comparación con "`fc_hl_bigvalues`" y nos permitió evaluar si un enfoque más simple sigue siendo efectivo.

- "**`fc_hl_bigvalues_and_2layers`**": Este experimento incorpora dos capas ocultas con 200 nodos cada una, lo que agrega una mayor profundidad a la red. Además, mantenemos la estructura de capas adicionales entre las capas ocultas y la capa de salida.

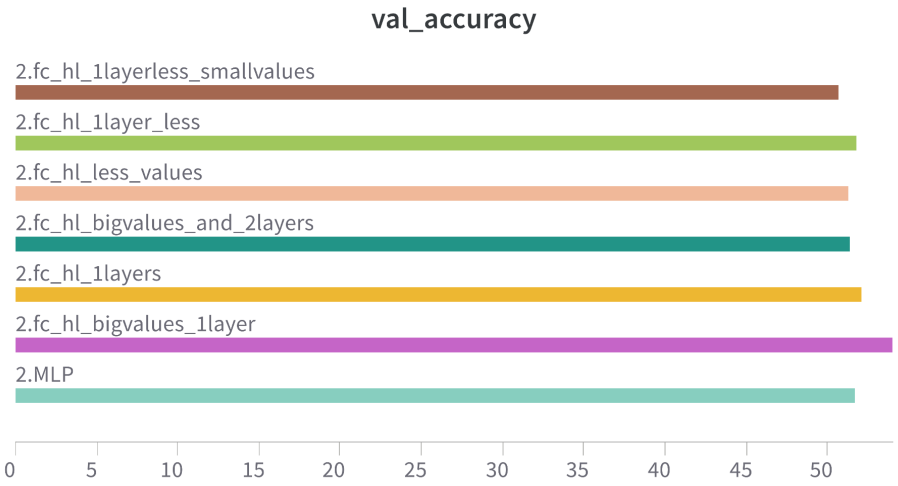
- "**`fc_hl_less_values`**": En este experimento, reducimos drásticamente la cantidad de nodos en las capas densas, lo que simplifica la arquitectura. La capa oculta intermedia consta de solo 50 nodos y se mantiene una estructura simple.

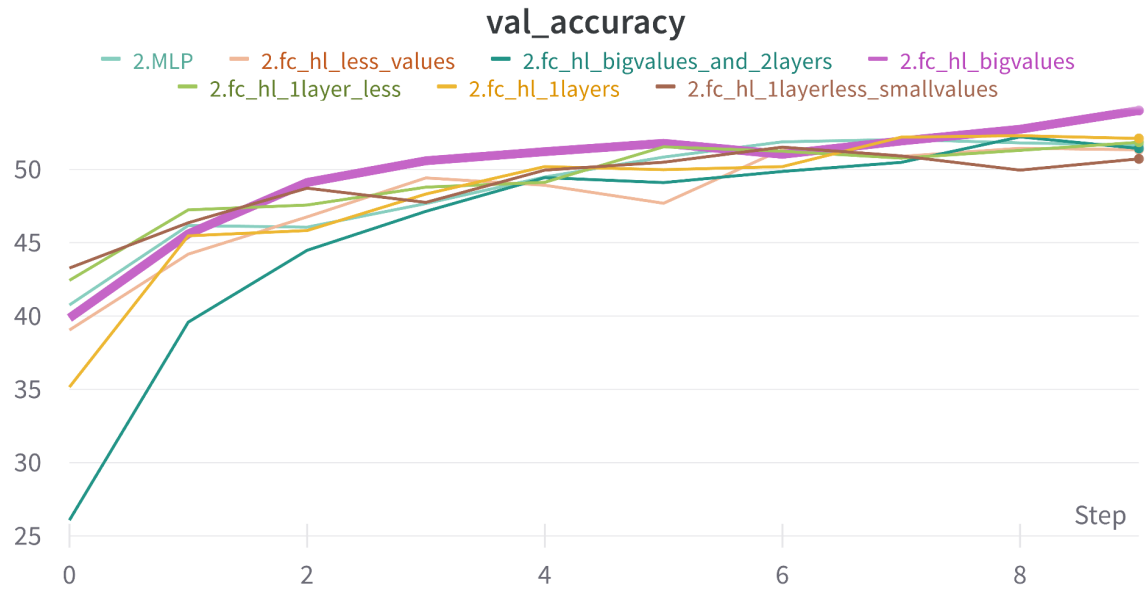
- "**`fc_hl_1layer_less`**": Este experimento lo implementamos teniendo solo una capa oculta con 120 nodos, lo que simplifica aún más la arquitectura al eliminar una de las capas intermedias.

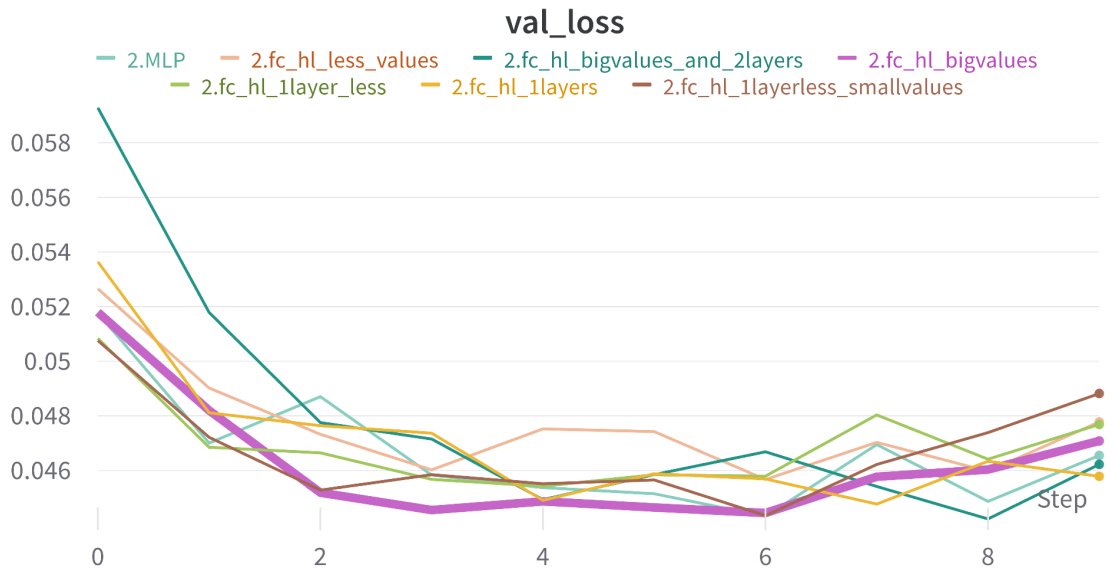
- "**`fc_hl_1layerless_smallvalues`**": En este último experimento, disminuimos significativamente la complejidad al utilizar una sola capa oculta con 50 nodos y reducir aún más el número de nodos en las capas densas.

Luego de esta serie de pruebas, encontramos que el mejor resultado nos lo daba el experimento "**fc_hl_bigvalues**". Detalladamente, en este experimento decidimos aumentar la complejidad del modelo al agregar una capa oculta con 150 nodos. Esta capa se sitúa entre las capas iniciales de procesamiento y la capa oculta intermedia de 150 nodos adicionales. La adición de nodos y capas pudo proporcionar al modelo una mayor capacidad de representación, lo que resultó en una mayor capacidad de capturar patrones más sutiles o detallados en las imágenes. Sin embargo,

también es importante tomar en cuenta que este aumento de complejidad llevó un costo en términos de recursos computacionales y tiempo de entrenamiento.





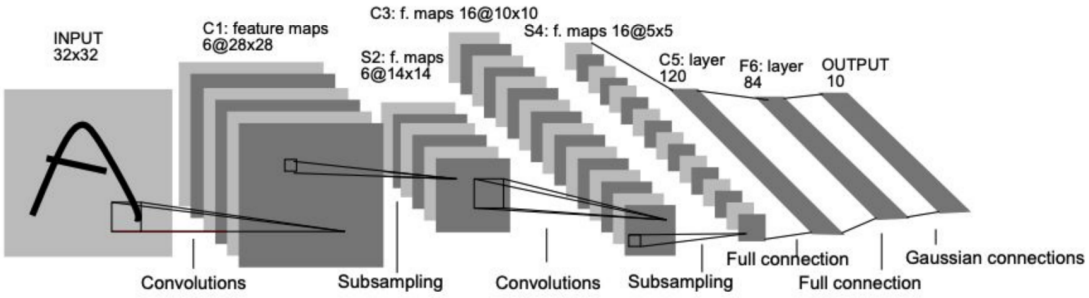


3. Experimentos con Arquitectura CNN

Estos son los siguientes experimentos con capas convolucionales que llevamos a cabo para aprender cuál de ellas nos brindaba el mejor enfoque para clasificar imágenes:

- "**`cnn_lessvalues`**": Este experimento se basó en una arquitectura de red con capas convolucionales, seguida de capas completamente conectadas. Las capas convolucionales consistían en dos convoluciones con capas ocultas. A pesar de usar menos parámetros que algunas de las variantes, produjo resultados notables.

- "**`CNN_LeNet-5`**": Esta variante se basó en la arquitectura LeNet-5, una red neuronal convolucional clásica conocida.

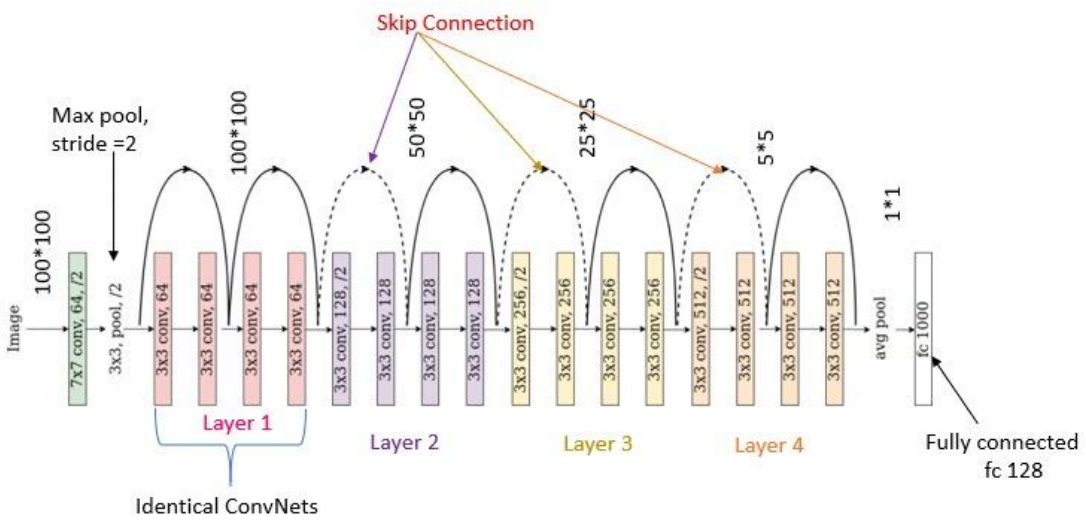


Esta está compuesta de 7 capas internas diferentes, 3 de convolución, 2 de agrupación y otras 2 que están densamente conectadas con su capa anterior. Es una red en la que la complejidad de los patrones analizados por las capas convolucionales aumenta a medida que nos acercamos a la salida.

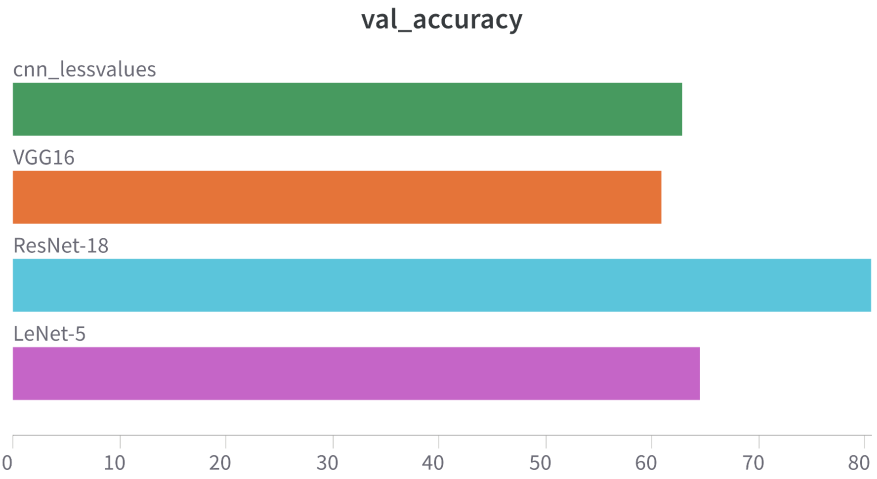
- "**`CNN_ResNet-18`**": Aca exploramos la arquitectura ResNet-18, que es conocida por su profundidad y conexiones residuales. La red utilizó bloques básicos y una estructura más profunda.

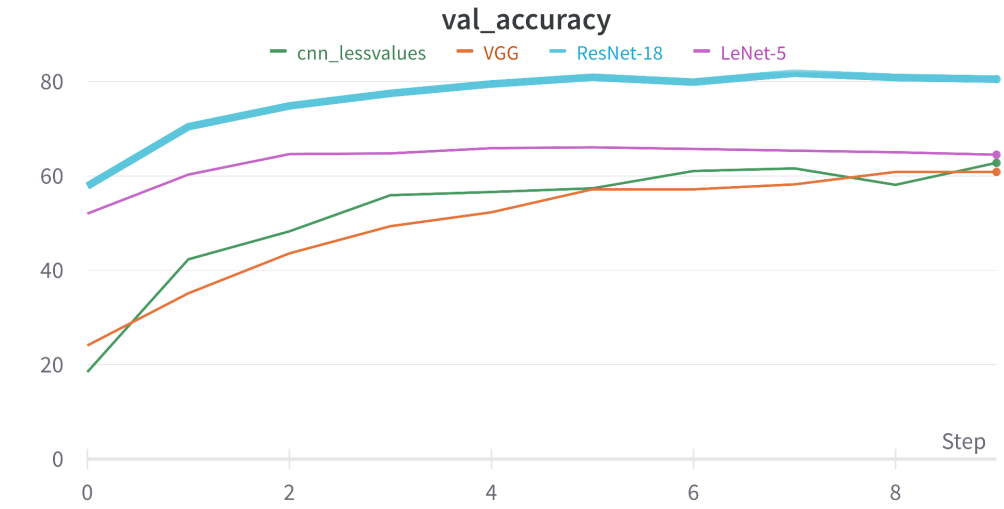
- "**`CNN_VGG16`**": Inspirada en la arquitectura VGG-16, esta variante utilizó múltiples capas convolucionales y una capa completamente conectada. Su complejidad se basa en la gran cantidad de capas y parámetros, lo que puede requerir una potencia computacional significativa.

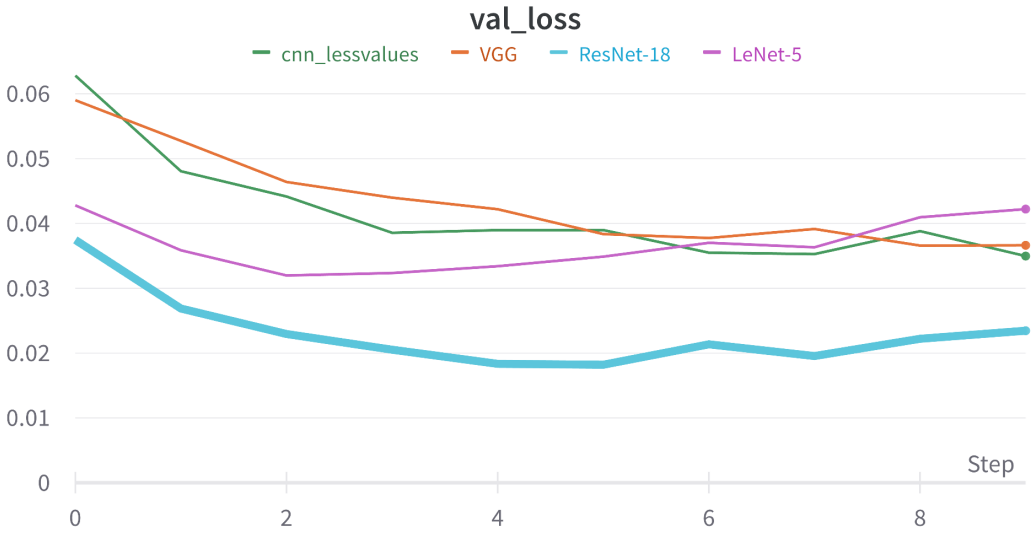
Luego de todos estos enfoques, vimos que la "**`CNN_ResNet-18`**" (Residual Network) logró obtener los mejores resultados, por lo cual lo detallaremos en mayor profundidad.



ResNet18 se compone de bloques básicos llamados "BasicBlock", que se apilan en capas para construir la red. Cada bloque básico contiene dos capas de convolución, capas de normalización y una capa de activación ReLU. La capa de convolución inicial conv1 toma imágenes de 3 canales de color y las convierte en 64 características. Cada `BasicBlock` realiza las siguientes operaciones: Convolución (con un kernel de 3x3 para extraer características de la imagen), normalización (para estandarizar las características), función de activación ReLU (para introducir no linealidad) y vuelve a aplicar convolución y normalización. Después de pasar por las capas convolucionales, se utiliza una capa de Average Pooling para reducir las características espaciales a un solo valor promedio en cada canal. Luego se aplanan y se conectan a una capa fully connected para realizar la clasificación en las clases de salida.







4. Experimentos con Funciones de Activación

En esta etapa, probamos varias diferentes funciones de activación en la arquitectura "`CNN_ResNet-18`". Las funciones de activación son las que determinan cómo una neurona debe responder a la suma ponderada de sus entradas. Cada función de activación tiene características específicas, por lo cual vamos a detallar cuales exploramos y en que se destacaron:

- **`ReLU`**: Esta función de activación es la más común en las redes neuronales y se utiliza para introducir no linealidad en el modelo. Transforma cualquier valor negativo a cero y

mantiene los valores positivos sin cambios. Es eficiente en cuanto a cálculos y ayuda a superar el problema de la desaparición del gradiente.

- **`Leaky ReLU`**: Leaky ReLU es una variante de ReLU que permite un valor muy pequeño, no nulo, para entradas negativas. Esto ayuda a abordar el problema de las neuronas "muertas" que no se activan durante el entrenamiento.

- **`ReLU6`**: ReLU6 es una modificación de ReLU que limita la salida a un rango de [0, 6]. Esta restricción puede ser útil en situaciones en las que se desea una activación acotada.

- **`ELU`**: Esta es otra función de activación que permite valores negativos, pero de una manera suave y diferenciable. Esto puede ayudar a las redes a aprender de manera más eficiente y converger más rápido.

- **`Tanh`**: La función tangente hiperbólica tiene valores entre -1 y 1. A diferencia de ReLU, es simétrica y puede ser útil en situaciones donde las entradas son negativas o positivas.

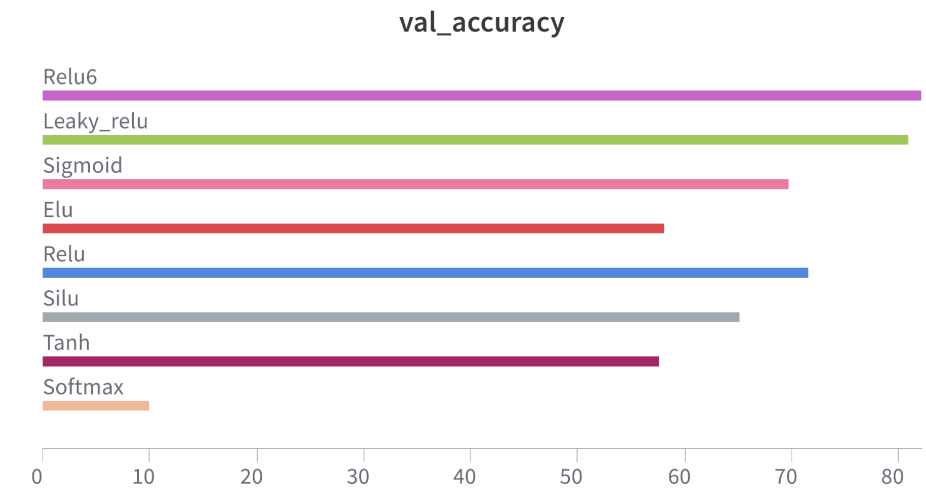
- **`SiLU`**: Es otra función de activación que se basa en una versión suavizada de la función sigmoide que implica multiplicar el valor de entrada por el resultado de la función sigmoide aplicada al mismo valor de entrada.

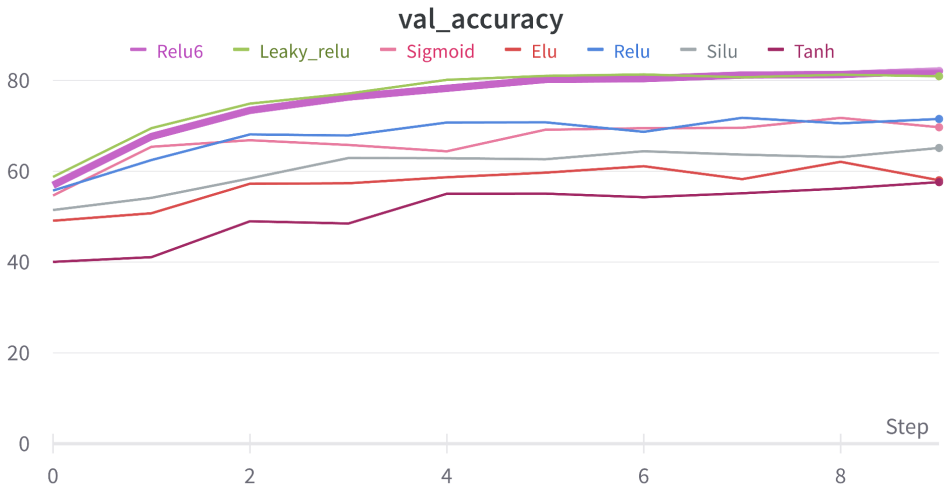
- **`Sigmoid`**: La función sigmoide se usa para problemas de clasificación binaria. Transforma las entradas en el rango de 0 a 1 y se utiliza para estimar probabilidades.

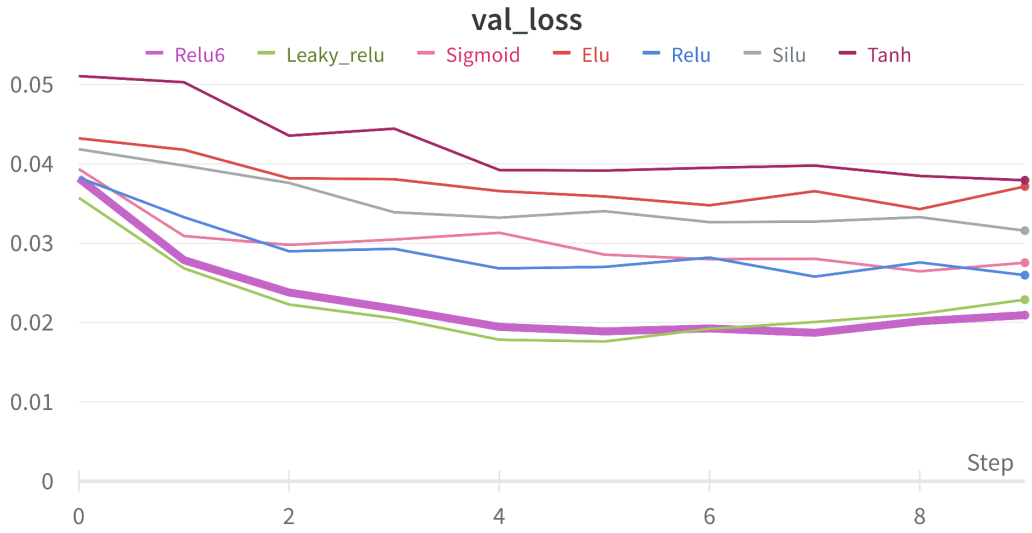
- **`Softmax`**: La función Softmax se emplea comúnmente en la capa de salida de redes neuronales para la clasificación multiclase. Convierte un vector de puntuaciones en una distribución de probabilidad, donde la suma de todas las salidas es igual a 1.

Después de evaluar el rendimiento de todas estas funciones de activación en nuestro modelo de clasificación, vimos que la mejor precisión y perdida la traía la función de activación **Relu6**. Por

esta razón, seguimos implementando esta en los próximos puntos de la exploración para seguir construyendo un mejor modelo.



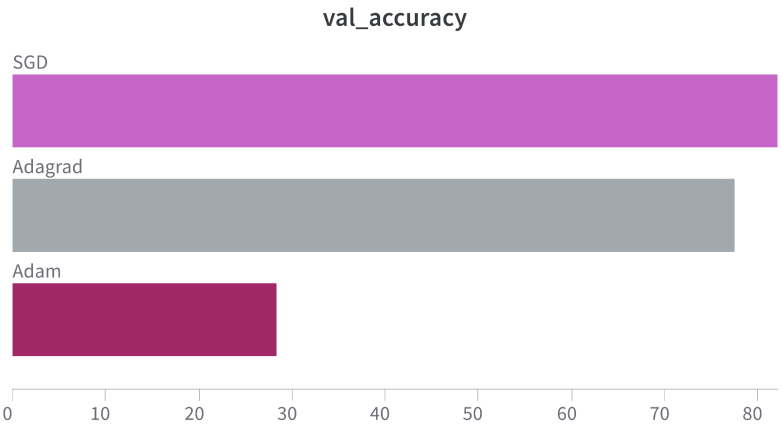


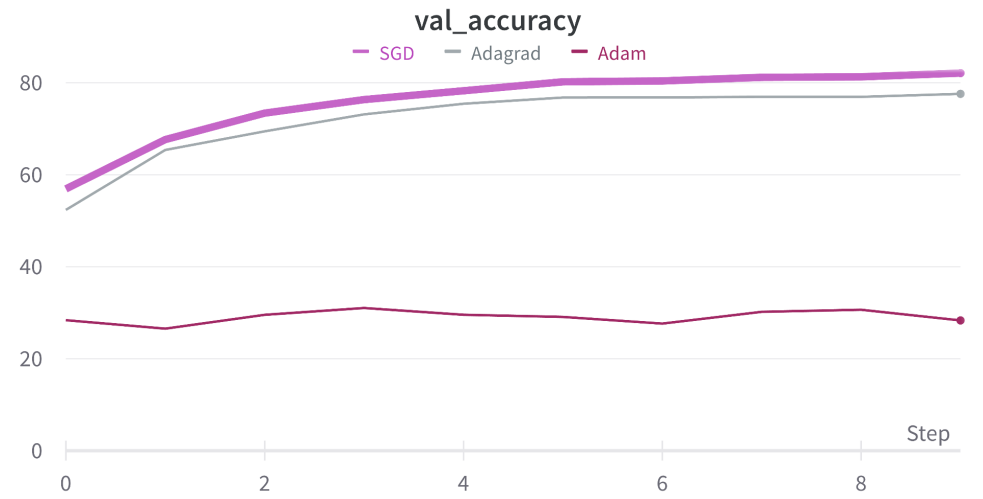


5. Experimentos con Optimizadores

En el estudio de optimización, evaluamos varios optimizadores y schedulers para determinar cuál producía el mejor rendimiento y comparamos el rendimiento de los siguientes optimizadores:

- **`SGD`** (Stochastic Gradient Descent)
- **`Adam`**
- **`Adagrad`**



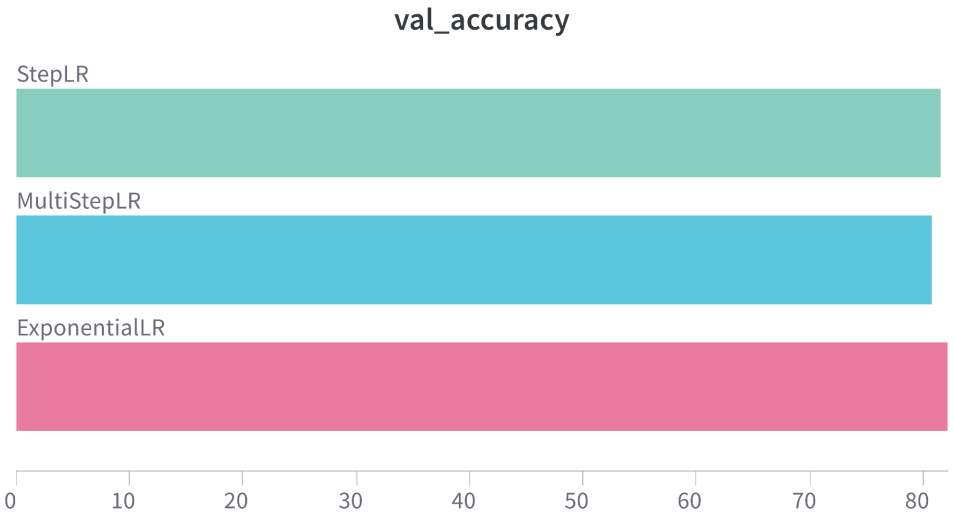


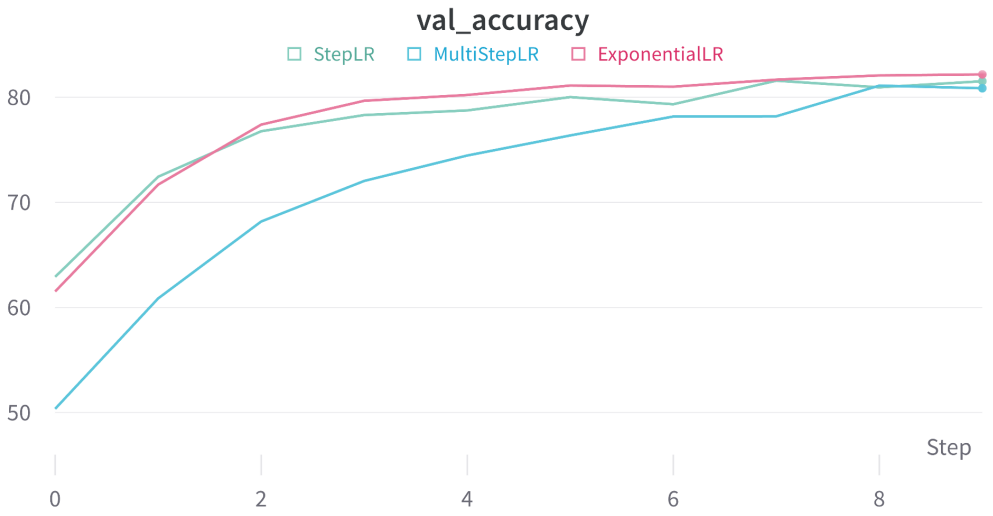
Podemos ver que el optimizador SGD con un learning rate constante de 0.01 y un valor de momentum de 0.9 nos dio el mejor equilibrio y rendimiento para nuestro experimento en comparación con los otros optimizadores. Acá aprendimos también que las diferentes combinaciones de optimizadores con funciones de activación pueden generar una variedad en la efectividad final del modelo, ya que puede que algunos no funcionen bien entre sí. Específicamente, se pudo ver que el optimizador Adam no funcionaba efectivamente con la función de activación (Relu6) que habíamos establecido.

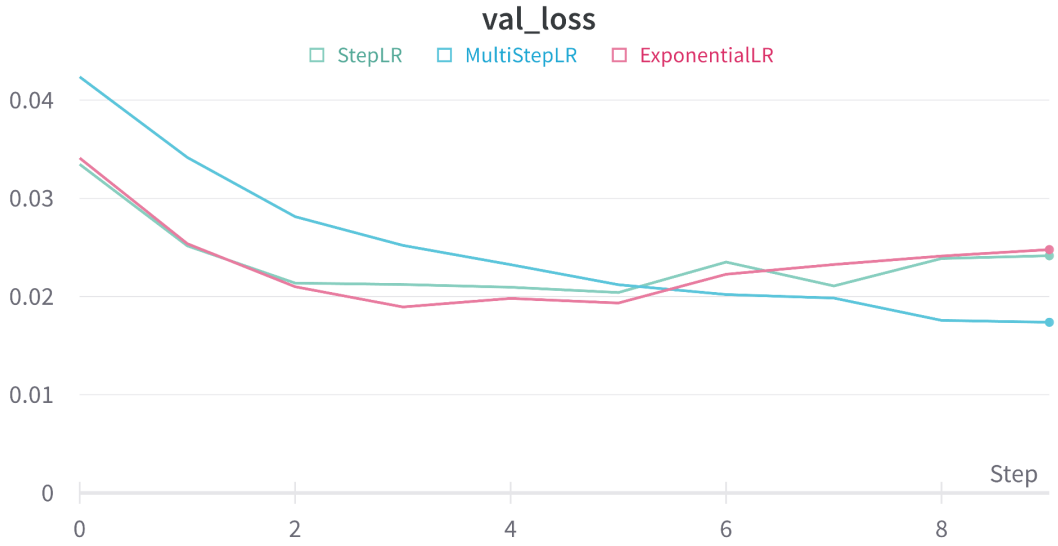
Otros optimizadores que investigamos fueron RMSprop y Adadelta, que no terminaron siendo de uso para nuestro experimento, aunque podría ser una oportunidad de mayor experimentación a futuro..

Además, utilizamos varios schedulers para ajustar el learning rate durante el entrenamiento, incluyendo:

- **`ExponentialLR`**
- **`MultiStepLR`**
- **`StepLR`**







A partir de estos experimentos, concluimos que el mejor scheduler nos lo brindó el ExponentialLR, ya que tuvo la precisión más alta del modelo.

6. Experimentos con Entrenamiento

En esta sección, la idea fue mejorar el rendimiento de nuestra red neuronal mediante la optimización de hiperparámetros. Nuestro objetivo era explorar una variedad continua de configuraciones de hiperparámetros para determinar y elegir cuáles producirían el mejor resultado en términos de precisión y pérdida en nuestro conjunto de datos. Los rangos de valores que probamos fueron los siguientes:

- Learning rate: [0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1]
- Momentum: [0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9]
- Epochs: list(range(10, 80, 5))
- Weight Decay: [1e-2, 1e-3, 1e-4, 1e-5]
- Batch size: [8, 16, 32, 64, 128, 256, 512]

Luego de investigar la combinación entre todos los posibles valores de los hiperparametros, pudimos identificar una combinación específica que resultó con en el mejor desempeño del modelo. Estos valores fueron los siguientes:

- Learning rate: 0.001
    - Momentum: 0.7
    - Epochs: 25
    - Batch size: 64
    - weight_decay = 0.0001

Name (15 visualized)	val_acc	train_accuracy	val_loss	train_loss	epochs	learning_rate	batch_size	momentum
6.CNN_ResNet-18 relu6 SGD optHP	81.74	99.73	0.03355	0.0002653	30	0.02	16	0.9
6.CNN_ResNet-18 relu6 SGD lr=0.1 batch…	83.91	87.415	0.007382	0.005664	15	0.1	64	0.6
6.CNN_ResNet-18 relu6 SGD lr=0.05 batc…	85.13	86.575	0.006886	0.006008	25	0.05	64	0.6
6.CNN_ResNet-18 relu6 SGD lr=0.05 batc…	63.09	62.768	0.03393	0.03402	15	0.05	32	0.6
6.CNN_ResNet-18 relu6 SGD lr=0.02 batc…	84.05	85.72	0.01426	0.01297	10	0.02	32	0.7
6.CNN_ResNet-18 relu6 SGD lr=0.02 batc…	82.37	85.853	0.00397	0.003198	20	0.02	128	0.8
6.CNN_ResNet-18 relu6 SGD lr=0.01 batc…	81.01	82.6	0.01732	0.01578	10	0.01	32	0.7
6.CNN_ResNet-18 relu6 SGD lr=0.005 bat…	74.37	74.63	0.002937	0.002828	10	0.005	256	0.8
6.CNN_ResNet-18 relu6 SGD lr=0.001 bat…	83.51	85.383	0.01504	0.01319	70	0.001	32	0.9

El seleccionado anteriormente es el modelo con la combinación óptima de hiperparametros que investigamos.

7. Experimentos con Regularización

Para mitigar el problema de overfitting muy común en estos casos, exploramos diversas estrategias de regularización. Para esto, implementamos Data Augmentation, Dropout, Ridge, Lasso para determinar cuál producía el mejor rendimiento y comparamos el rendimiento de los siguientes optimizadores:

Data augmentation:
Para enriquecer el conjunto de datos de entrenamiento, aplicamos transformaciones aleatorias a las imágenes. Estas transformaciones incluyen el relleno (padding) de las imágenes con píxeles, recortes aleatorios, volteos horizontales, normalización de valores de píxeles y la posibilidad de eliminar regiones aleatorias en las imágenes. El objetivo de esto es aumentar la diversidad de los

ejemplos de entrenamiento, lo que ayuda al modelo a mejorar su capacidad para reconocer patrones en nuevas imágenes.

Dropout:

Durante el entrenamiento de la red neuronal, se "apaga"de manera aleatoria una fracción de las neuronas en cada capa, lo que conduce a la eliminación temporal de la información y conexiones asociadas con esas neuronas, lo que a su vez obliga a que la red aprenda de manera más robusta y generalice mejor a datos no vistos.

Ridge:

El objetivo es minimizar la función de costo que mide la diferencia entre las predicciones del modelo y los valores reales observados. En la regresión ridge, además de este objetivo, se agrega un término de regularización que penaliza los coeficientes que se vuelven muy grandes. Esta penalización se basa en la norma L2 (norma euclidiana), y su objetivo es limitar la magnitud de los coeficientes, previniendo así el sobreajuste.

Lasso:

En contraste con la penalización L2 utilizada en la regresión ridge, la regresión Lasso emplea una penalización L1 (norma de valor absoluto) sobre los coeficientes del modelo. Además tiene la propiedad de "encoger" algunos coeficientes hasta cero, lo que conlleva a la selección de características y a la reducción de la dimensionalidad del modelo.

8. Evaluación final

Finalmente, después de haber explorado diversos aspectos del problema de clasificación de imágenes, como las capas convolucionales, las funciones de activación, los optimizadores y las técnicas de regularización, logramos construir nuestro mejor modelo. Este modelo optimizado, que incorpora todas las mejoras detalladas en el informe, ha demostrado una precisión del 83%.

Referencias

Lista de referencias a recursos utilizados en el trabajo:

- *Implementing ResNet18 for Image Classification | Kaggle*
- *ResNet18 from scratch using Pytorch | Kaggle*
- *Writing VGG from Scratch in PyTorch*