Instituto Tecnológico de Buenos Aires

Departamento de Ingeniería Electrónica

25.63 - Redes Neuronales

Trabajo Práctico

Skin Dataset Classification

Alumno

Nicolás Pablo Singermann

nsingermann@itba.edu.ar

61136

Docentes

Carlos Selmo

Rodrigo Cardenas

2025

Índice

1. Preguntas: Clasificación de imágenes con PyTorch y MLP

1.1. Dataset y Preprocesamiento

- **¿Por qué es necesario redimensionar las imágenes a un tamaño fijo para una MLP?**

Ambos datasets (train y val) tienen imágenes de distintos tamaños/resoluciones, es necesario redimensionar las imágenes a cierto tamaño porque una MLP no puede recibir imágenes de distintos tamaños (espera un vector de entrada de tamaño fijo, a diferencia de una CNN).

- **¿Qué ventajas ofrece Albumentations frente a otras librerías de transformación como torchvision.transforms?**

Albumentations es más complejo y permite realizar más una mayor cantidad de transformaciones, también es más rápido, lo cual va a ser útil para datasets grandes. Torchvision.transforms resulta más básico, solo puede rotar, redimensionar y recortar imágenes.

- **¿Qué hace A.Normalize()? ¿Por qué es importante antes de entrenar una red?**

Normaliza la imagen, es decir: Los pixeles pasan de tener un valor en el rango [0, 255] a [0, 1]. Luego les aplica la típica formula de normalización (restar la media y dividir por desvío estándar con media y desvío prestablecidos por la librería). Esto mejora la convergencia de la red, evitando el gradient exploding, haciendo que el entrenamiento sea más estable.

- **¿Por qué convertimos las imágenes a ToTensorV2() al final de la pipeline?**

Esto se debe a que las redes neuronales en PyTorch esperan tensores, a diferencia de Albumentations que trabaja con vectores de numpy. El tensor es un tipo de dato propio de PyTorch el cual es esperado por la MLP.

1.2. Arquitectura del Modelo

- **¿Por qué usamos una red MLP en lugar de una CNN aquí? ¿Qué limitaciones tiene?**

Principalmente por que MLP es más simple y va a ser mayor el beneficio desde el punto de vista didáctico. Sin embargo tiene limitaciones, va a requerir de una mayor cantidad de parámetros que CNN y no aprovecha el hecho de que pixeles cercanos/adyacentes en una imagen están relacionados, cosa que CNN logra capturar gracias a las convoluciones locales. Con MLP también se va a requerir de una mayor cantidad de datos para poder entrenar el modelo.

- **¿Qué hace la capa Flatten() al principio de la red?**

Convierte cada imagen de 64x64x3 en un vector unidimensional de 12288 pixeles, esto se debe a que las capas lineales que vienen luego solo admiten vectores unidimensionales como entrada.

- **¿Qué función de activación se usó? ¿Por qué no usamos Sigmoid o Tanh?**

Ambas capas ocultas (la primera de 512 neuronas y las segunda de 128) tienen como función de activación la RELU. Esta es más rápida computacionalmente (converge más rápido), si la entrada es negativa da 0, y si es positiva copia la entrada a la salida, por lo que su función es simple. También evita el problema de vanishing gradient porque no satura para valores positivos.

- **¿Qué parámetro del modelo deberíamos cambiar si aumentamos el tamaño de entrada de la imagen?**

Habría que cambiar el *input_size* al nuevo tamaño de las imágenes.

1.3. Entrenamiento y Optimización

- **¿Qué hace optimizer.zero_grad()?**

Por defecto PyTorch acumula los gradientes, cada vez que se llama a *loss.backward()* los gradientes calculados se suman a los anteriores. Por lo que antes de calcular los gradientes del batch actual, hay que resetear los gradientes que ya se tenían calculados de antes, para que no se acumulen, esto se logra con el *optimizer.zero_grad()*.

- **¿Por qué usamos CrossEntropyLoss() en este caso?**

Es la función de perdida estándar para clasificación con muchas clases y resulta adecuada para el caso de skin classification.

- **¿Cómo afecta la elección del tamaño de batch (batch_size) al entrenamiento?**

Los gradientes serán más estables y la loss sería más suave, sin embargo va a requerir más memoria para el entrenamiento.

- **¿Qué pasaría si no usamos model.eval() durante la validación?**

Si no se utiliza el *model.eval()* durante la validación, PyTorch deja el modelo en modo de entrenamiento, en donde el dropout está activado (apagar aleatoriamente ciertas neuronas de la capa oculta). A su vez el *batch normalization* será utilizando únicamente estadísticas del batch actual y no un promedio de todos los batches. Esto haría que la loss y el accuracy sean menos confiables, la validación no reflejaría realmente el desempeño del modelo.

1.4. Validación y Evaluación

- **¿Qué significa una accuracy del 70 % en validación pero 90 % en entrenamiento?**

Diría que se tiene cierto overfitting, el modelo aprendió muy bien los datos durante el entrenamiento, pero no generaliza del todo bien con datos nuevos. Se puede buscar aumentar la cantidad de datos de entrenamiento, o reducir la complejidad del modelo.

- **¿Qué otras métricas podrían ser más relevantes que accuracy en un problema real?**

La precisión, que resulta útil cuando los falsos positivos pueden ser costosos, como es en el caso de detección de enfermedades. Otra métrica de gran importancia puede ser la matriz de confusión.

- **¿Qué información útil nos da una matriz de confusión que no nos da la accuracy?**

Mientras que el accuracy nos da el porcentaje de predicciones que fueron correctas, la matriz de confusión nos permite analizar que clases tienden a confundirse entre sí. Idealmente solo tendría valores en su diagonal principal (matriz diagonal), pero no es lo que ocurre en la práctica:

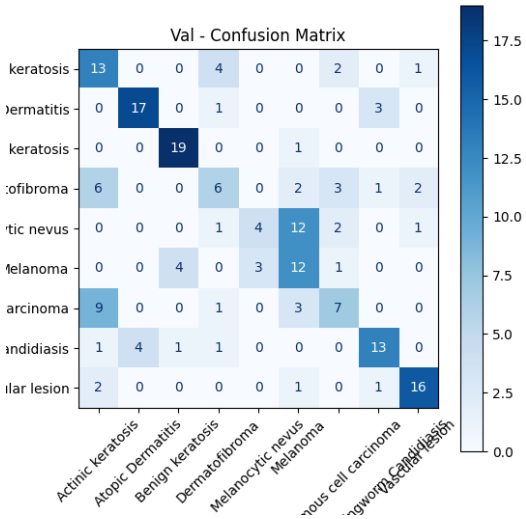

Figura 1: Matriz de confusión val epoch 9 - TensorBoard

Notar como la clase Nº7, *Squamous Cell Carsinoma*, se la tiende a predecir como si fuera la clase Nº1, *Actinic keratosis*. Algo similar ocurre con las 3 clases anteriores. Por lo que llegamos a la conclusión de que de las 9 clases que tenemos para clasificar, el modelo funciona bastante bien dentro de todo para 5 de ellas, mientras que las otras 4 se las confunde seguido.

- **En el reporte de clasificación, ¿qué representan precisión, recall y f1-score?**

Precisión, recall y f1-score son métricas muy utilizadas para problemas de clasificación. La **precisió**n indica el porcentaje de veces que el modelo al predecir cierta clase, acertó. Se puede ver por ejemplo que en este epoch, el modelo acertó con una precisión del 64 % la segunda y tercer clase, y como no acertó nunca a la cuarta clase por ejemplo, es decir que las veces que predijo que la imagen pertenecía a dicha clase, erró siempre. El **recall** es también conocido como la sensibilidad y es un porcentaje que indica, de todas las imágenes que pertenecían a cierta clase, cuantas se detectaron. Puede prestar algo de confusión con la precisión, la clave es pensar lo siguiente:

- De todas las veces que dije "A", acerté el P %
- Del total de imágenes que realmente eran "A", detecté el R %

Entonces, se llega a la siguiente conclusión: Si la precisión es alta pero recall bajo, se predice pocas veces esa clase, pero cuando lo hace, acierta. Si el recall es alto pero la precisión es baja, entonces predice la clase muchas veces, pero también se equivoca mucho. En una aplicación real, la precisión es importante cuando los falsos positivos son costosos, mientras que el recall es de interés cuando los falsos negativos son costosos.

Por último, el **f1-score** combina la precisión y el recall, indicando de cierta forma el trade off entre predecir correctamente, y cubrir todos los positivos. Es la media armonica entre ambos y se quiere que sea cercano a 1:

- F1 cercano a 1: El modelo es bueno tanto en precisión como en recall
- F1 cercano a 0: El modelo erra bastante, ya sea porque predice la clase incorrectamente (precisión baja), o no detecta muchos ejemplos reales (recall baja)

A continuación se ve el classification report del primer epoch de validación:

	precision	recall	f1-score	support
Actinic keratosis	0.21	1.00	0.34	20
Atopic Dermatitis	0.64	0.86	0.73	21
Benign keratosis	0.64	0.90	0.75	20
Dermatofibroma	0.00	0.00	0.00	20
Melanocytic nevus	0.69	0.55	0.61	20
Melanoma	0.30	0.15	0.20	20
Squamous cell carcinoma	0.00	0.00	0.00	20
Tinea Ringworm Candidiasis	0.50	0.05	0.09	20
Vascular lesion	0.00	0.00	0.00	20
accuracy			0.39	181
macro avg	0.33	0.39	0.30	181
weighted avg	0.33	0.39	0.31	181

Figura 2: Classification report val epoch 0 - MLFlow

1.5. TensorBoard y Logging

- **¿Qué ventajas tiene usar TensorBoard durante el entrenamiento?**

Visualizar el progreso de las métricas en los distintos epoch de entrenamiento, tanto el accuracy como la loss.

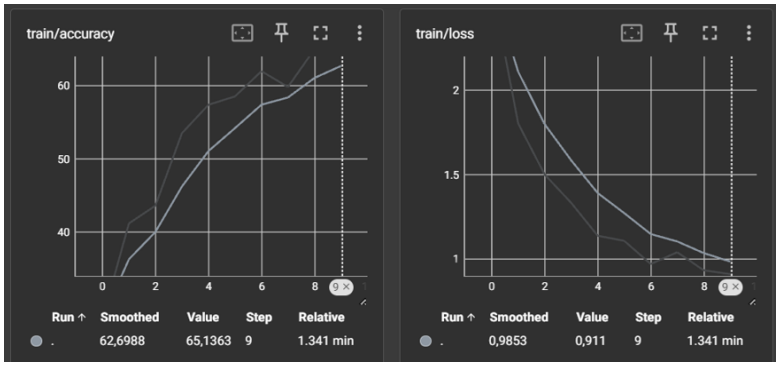

Figura 3: Accuracy y loss en train

- **¿Qué diferencias hay entre loguear add_scalar, add_image y add_text?**

Para registrar/loguear métricas numéricas, como el accuracy o la loss por epoch, se utiliza el **add_scalar**, esto le permitirá al TensorBoard trazar las curvas. Para loguear imágenes en TensorBoard utilizamos **add_image**, como fue en el caso de la validación, en donde se loguean las 8 imágenes del primer batch de cada epoch, y para loguear texto se usa el **add_text**, utilizado en el ejemplo para loguear classification reports:

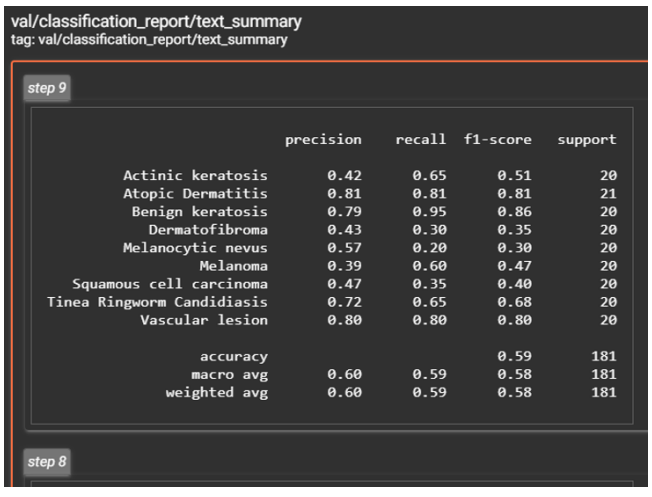

Figura 4: Classification report val - TensorBoard

- **¿Por qué es útil guardar visualmente las imágenes de validación en TensorBoard?**

No tengo del todo claro porque ver las imágenes propiamente dichas sería útil, si el accuracy, la loss y la matriz de confusión, pero ver las imágenes en si no le encuentro utilidad, me interesaría mas ver las del train las cuales tienen un pipeline de transformaciones mas complejo, con el posible flip horizontal y el ajuste de brillo o contraste.

- **¿Cómo se puede comparar el desempeño de distintos experimentos en TensorBoard?**

Hay que crear un segundo directorio de logs, en donde se hagan los nuevos logueos habiendo realizado alguna modificación al modelo, como por ejemplo, cambiar el tamaño de los batches.

```
#Crear un segundo directorio de logs
log_dir = "runs/mlp_experimento_2"
writer = SummaryWriter(log_dir=log_dir)
```

Figura 5: Crear un segundo experimento

Cree un segundo experimento en donde aumente el tamaño de los batches a 64. Ahora, para iniciar TensorBoard, en consola se ejecuta: *tensorboard –logdir=runs/*

Y en TensorBoard puedo comparar ambos modelos:

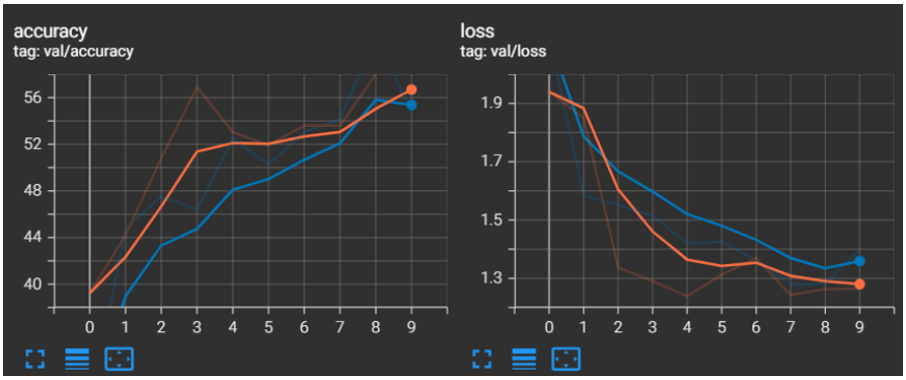

Figura 6: Comparación de 2 modelos

Como se puede ver, en lo que respecta al accuracy y a la loss, haber aumentado el batch size (curva azul) empeoro el desempeño del modelo.

1.6. Generalización y Transferencia

- **¿Qué cambios habría que hacer si quisiéramos aplicar este mismo modelo a un dataset con 100 clases?**

Para aplicar este mismo modelo sin cambios al nuevo dataset, simplemente habría que cambiar el valor de la variable **num_classes** a 100.

- **¿Por qué una CNN suele ser más adecuada que una MLP para clasificación de imágenes?**

La CNN resulta más adecuada por diversos motivos, entre ellos:

- Usa convoluciones locales que permiten detectar patrones, como bordes y texturas
- Es mas eficiente, en el sentido que requiere de una menor cantidad de parámetros para entrenar el modelo

- Pueden generalizar mejor con menos datos
- A diferencia de con MLP, puede procesar imágenes de distintas resoluciones
- **¿Qué problema podríamos tener si entrenamos este modelo con muy pocas imágenes por clase?**

Se tendría overfitting excesivo, en donde el modelo aprende de memoria el dataset, por lo que no va a generalizar de forma adecuada. Para solucionarlo se puede aprovechar de las Albumentations para obtener más imágenes, técnicas de regularización, incluso reducir la complejidad del modelo.

- **¿Cómo podríamos adaptar este pipeline para imágenes en escala de grises?**

El pipeline de transformaciones no cambiaría, lo que cambia es el **input_size** porque ahora las imágenes son de un solo canal, no de 3 como antes (RGB). El **input_size** de la MLP debería ser de 64x64x1. A su vez, el *getitem* de la clase *CustomImageDataset* no debe convertir las imágenes a "RGB", sino a "L" indicando escala de grises.

Se creo un tercer experimento en donde se hizo esto, obteniendo los siguientes resultados:

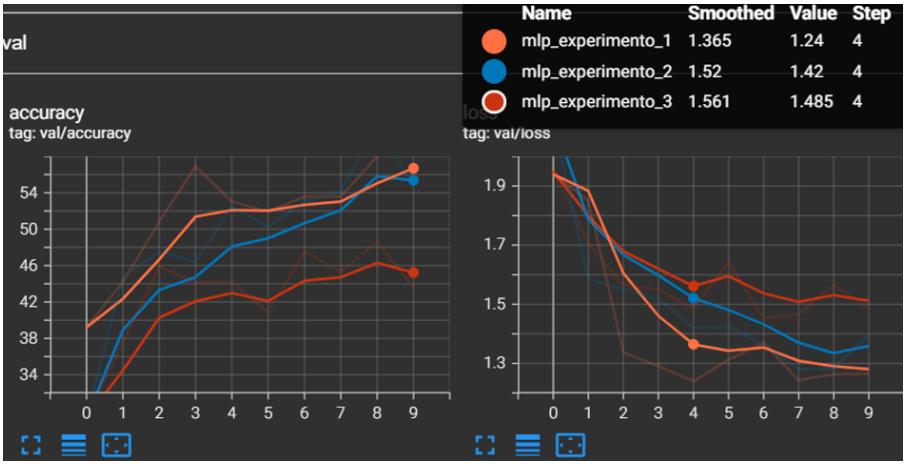

Figura 7: Comparacion con escala de grises

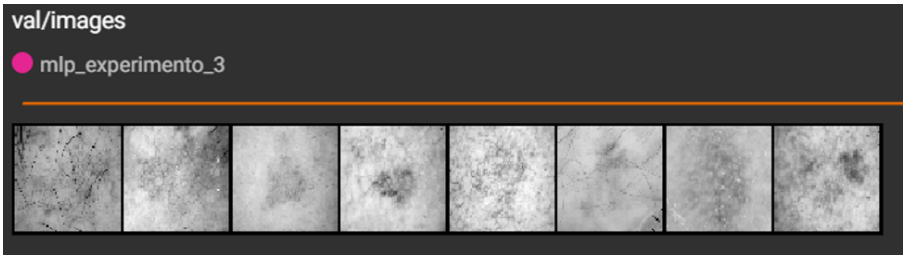

Figura 8: Escala de grises

El desempeño del modelo empeoró aún más, lo cual tiene sentido puesto a que ahora se cuenta con un dataset que contiene una menor cantidad de información. Cualquier información de una clase que podría tener un color que la distinga o sea característico de la misma se pierde.

1.7. Regularización

1.7.a. Preguntas teóricas

- **¿Qué es la regularización en el contexto del entrenamiento de redes neuronales?**

La regularización es una técnica utilizada para combatir el overfitting, se utiliza para que los pesos no aumenten significativamente. Existen distintas técnicas de regularización: Regularización L2, Regularización L1, Early Stop y Dropout.

- **¿Cuál es la diferencia entre Dropout y regularización L2 (weight decay)?**

Dropout elimina/apaga neuronas de forma aleatoria (durante el entrenamiento), eliminando considerablemente la cantidad de conexiones que tiene nuestra MLP. Esto se aplica durante el entrenamiento, durante la evaluación el dropout se desactiva, y notar que la cantidad de neuronas a apagar es un hiperparámetro. Por otro lado, **regularización L2** agrega un término a la función de costos para evitar que los pesos aumenten significativamente. Esto se debe a que si los pesos aumentan considerablemente, el modelo se vuelve mas complejo y propenso al overfitting. En este caso se tiene el hiperparámetro lambda:

$$J_R(w) = J(w) + \lambda \cdot \sum_{i}^{d} w_i^2 \tag{1}$$

- **¿Qué es BatchNorm y cómo ayuda a estabilizar el entrenamiento?**

Batch Normalization no es propiamente dicho una técnica de regularización, sino que sirve para disminuir los tiempos de entrenamiento. Busca normalizar las observaciones a la salida de la capa oculta en donde se lo aplique, para lo que va a calcular media y desvío de las activaciones. Cabe destacar que no se puede aplicar batch normalization y dropout a la misma capa oculta porque se interfieren.

- **¿Cómo se relaciona BatchNorm con la velocidad de convergencia?**

Esto se debe principalmente a que resuelve el inconveniente del *Internal Covariate Shift*, el cual consiste en que cuando los pesos cambian durante el entrenamiento, cambian las distribuciones de las capas de la red, por lo que las siguientes capas se tienen que volver a adaptar a estas nuevas distribuciones. Batch Normalization busca estandarizar la salida de cada capa para que tenga media nula y desvío unitario, provocando que las capas puedan aprender más rápido, aumentando la velocidad de convergencia de la red.

- **¿Puede BatchNorm actuar como regularizador? ¿Por qué?**

Si bien no fue diseñado como regularizador, Batch Normalization aporta de forma implícita regularización al modelo. Esto se debe principalmente a que introduce ruido en el entrenamiento, es decir ruido en las estadísticas del batch, logrando regularizar y reducir el overfitting.

- **¿Qué efectos visuales podrías observar en TensorBoard si hay overfitting?**

El efecto visual mas evidente del overfitting es comparando la loss en entrenamiento y en validación. En el entrenamiento baja de forma constante hasta incluso poder llegar a un valor cercano a cero, en cambio, en la validación, cuando se tiene overfitting, a partir de cierto punto la loss comenzará a crecer. Realizo la prueba con el ejemplo, creando un cuarto experimento en donde voy a aumentar la cantidad de epochs (para que aumente el riesgo de overfitting debido a que el modelo comienza a aprender de memoria los datos en lugar de generalizar). Se obtuvieron los siguientes resultados para 70 epochs:

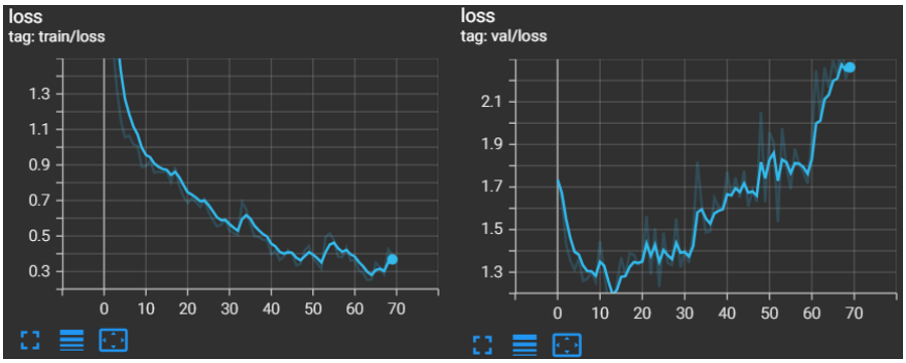

Figura 9: Overfitting

Se puede ver como a partir de los 14 epochs el modelo comienza a overfittear. Otro indicativo es comparando el accuracy en el entrenamiento y en la validación. En el entrenamiento sube hasta llegar de forma asintótica a cierto valor, mientras que cuando se tiene overfitting, en la validación, el accuracy comenzará a disminuir a partir de cierto punto.

- **¿Cómo ayuda la regularización a mejorar la generalización del modelo?**

La regularización fuerza al modelo a aprender patrones más simples que le van a servir para generalizar de forma adecuada, penalizando modelos complejos (con L2), evitando la gran dependencia de ciertas neuronas (con dropout) y estabilizando las activaciones (con Batch Normalization).

1.7.b. Actividades de modificación

- **Agregar Dropout en la arquitectura MLP**

Insertando el dropout en el modelo y creando un nuevo experimento, se obtuvieron los siguientes resultados:

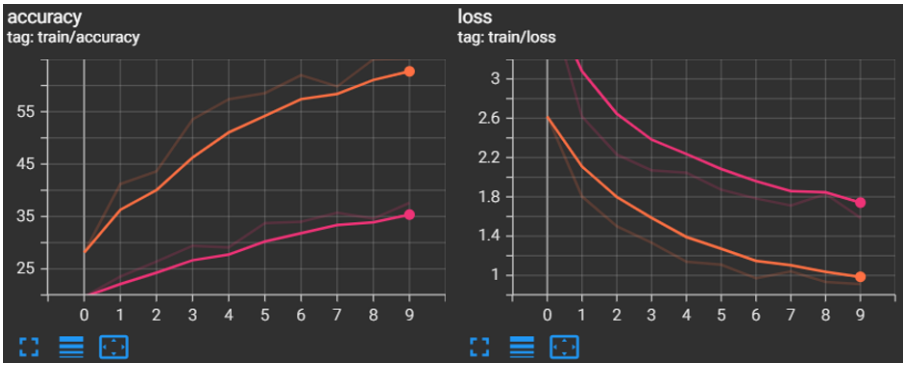

Figura 10: Comparación entrenamiento con/sin dropout

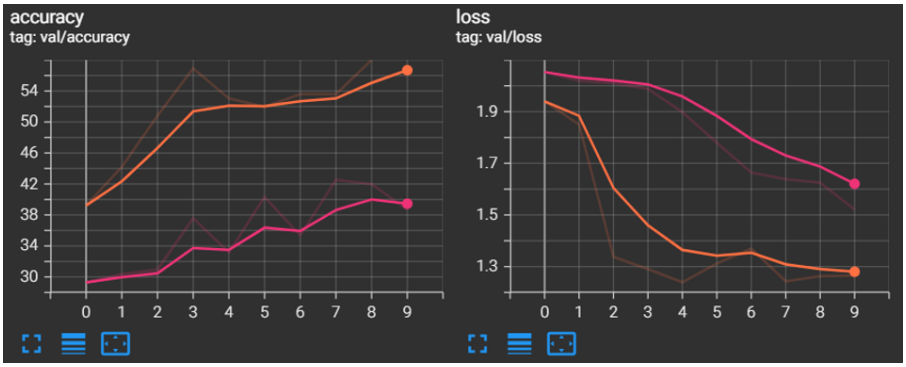

Figura 11: Comparación validación con/sin dropout

El modelo parece empeorar su desempeño con respecto al original sin dropout, entiendo esto se debe a realizar pocos epochs y el hecho de que el dropout es mas eficiente para una mayor cantidad de neuronas en la red. Procedo a realizar los experimentos con 50 epochs de ahora en más, por más que me gustaría hacerlos de 100, por una cuestión de tiempo tomo 50. Comparación con y sin dropout teniendo 50 epochs:

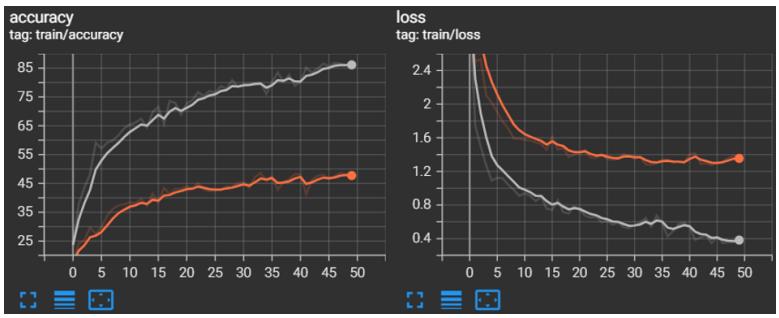

Figura 12: Comparación entrenamiento con/sin dropout - 50 epochs

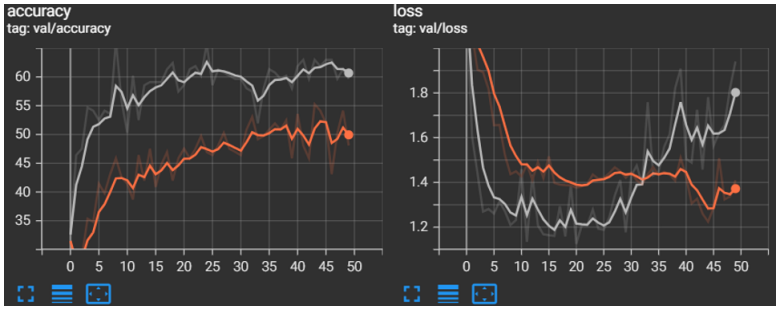

Figura 13: Comparación validación con/sin dropout - 50 epochs

Como se puede ver ahora (en naranja con dropout) no se tiene el overfitting que se tenía antes.

- **Agregar Batch Normalization**

Agregando batch normalization, continuando con 50 epochs, se tiene la siguiente comparación con el modelo original:

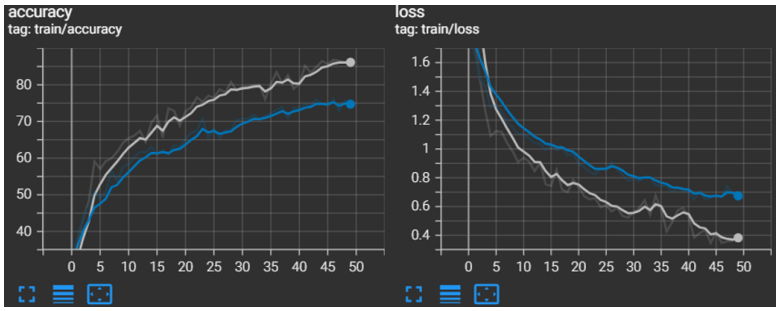

Figura 14: Comparación entrenamiento con/sin batch normalization

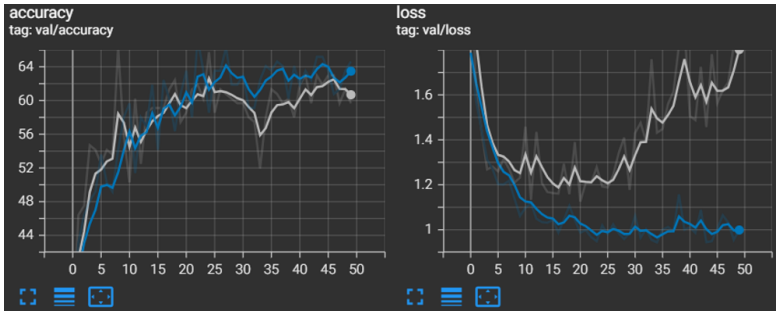

Figura 15: Comparación validación con/sin batch normalization

La mejora es considerable con respecto al modelo anterior donde solo se tenia dropout! No solo el accuracy aumento considerablemente en el entrenamiento, sino que en la validación (donde realmente importa) superó al modelo original, evitando el overfitting como se ve en la loss de la validación.

- **Aplicar Weight Decay (L2)**

Aplicando regularización L2 se obtienen la siguiente comparación con el modelo original:

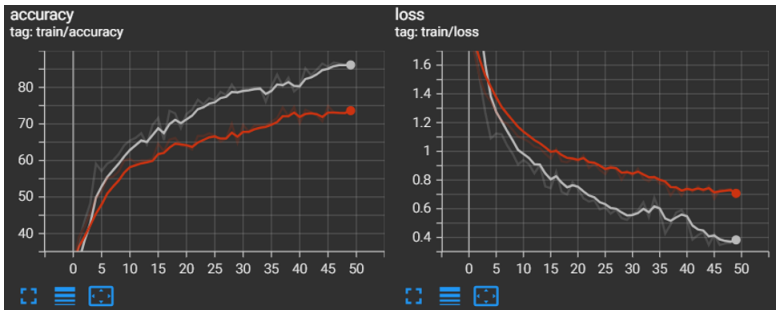

Figura 16: Comparación entrenamiento con/sin L2

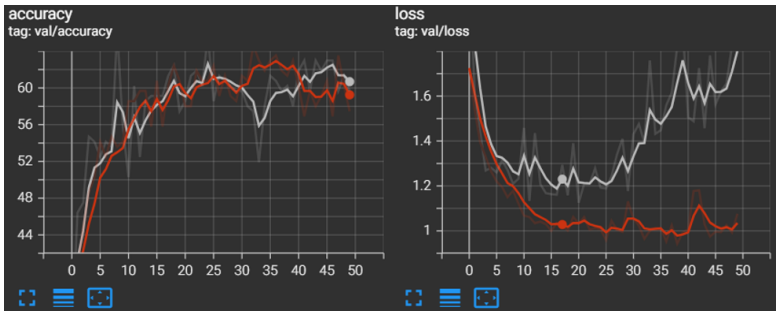

Figura 17: Comparación validación con/sin L2

Comparando ahora con el modelo antes de aplicar L2 (el caso que tenía batch normalization y dropout):

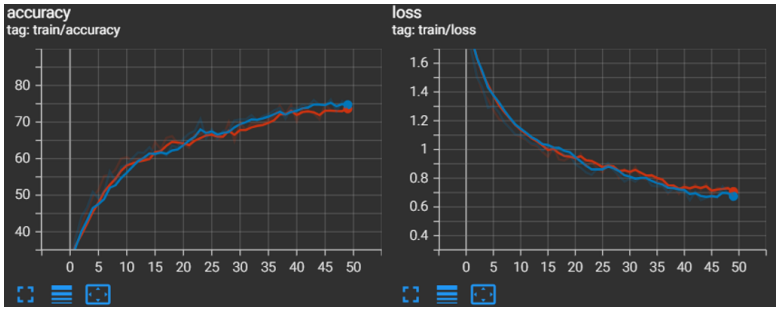

Figura 18: Comparación entrenamiento con/sin L2

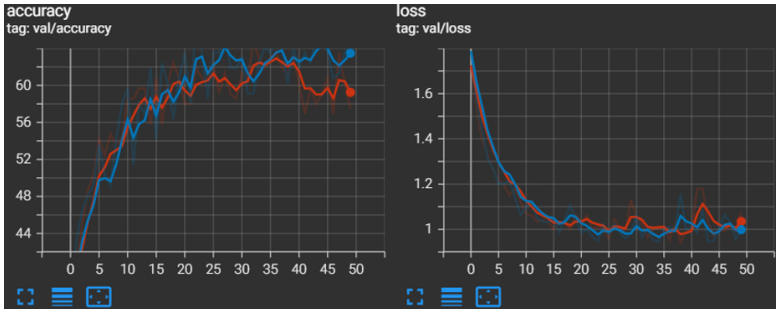

Figura 19: Comparación validación con/sin L2

Pareciera que el modelo tenía un mejor desempeño previo a aplicar L2, sin embargo son muy parecidos los resultados en estos 50 epochs, la clave parece estar en como el accuracy en validación

comienza al incorporar el L2, cosa que antes no ocurría. Entiendo que la regularización que incorpora L2 es innecesaria, dado que dropout y batch normalization ya actúan como regularizadores.

- **Reducir overfitting con data augmentation**

Aplicando el siguiente pipeline de transformaciones durante el entrenamiento:

```python
train_transform = A.Compose([
    A.Resize(64, 64),
    A.HorizontalFlip(p=0.5),
    A.ShiftScaleRotate(shift_limit=0.05, scale_limit=0.1, rotate_limit=15, p=0.5),
    A.RandomBrightnessContrast(brightness_limit=0.2, contrast_limit=0.2, p=0.2),
    A.Normalize(),
    ToTensorV2()
])
```

Figura 20: Modificando el pipeline de transformaciones

Manteniendo la regularización L2, se obtiene el siguiente resultado (comparando con el caso anterior que tenía L2 pero no la modificación en el pipeline de transformaciones)

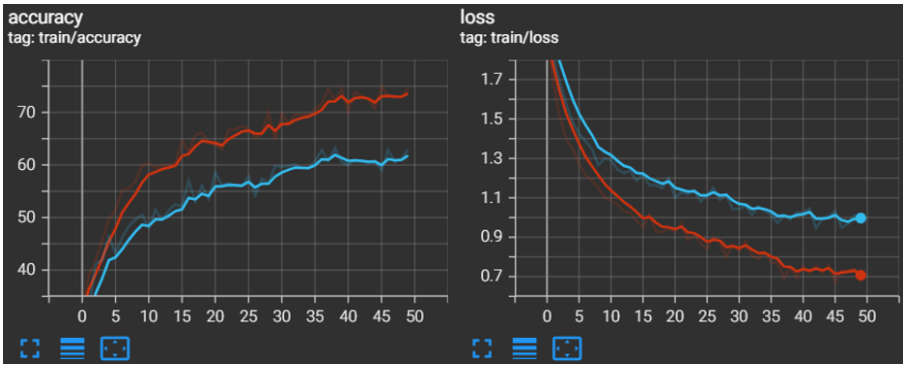

Figura 21: Comparación entrenamiento con/sin data augmentation

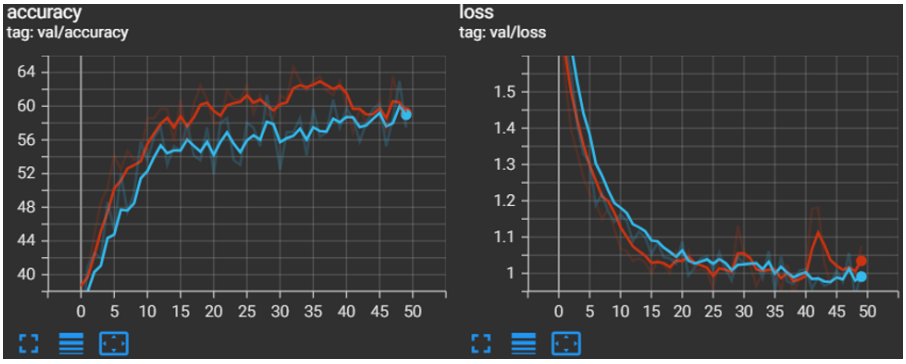

Figura 22: Comparación validación con/sin data augmentation

1.7.c. Preguntas prácticas

- **¿Qué efecto tuvo BatchNorm en la estabilidad y velocidad del entrenamiento?**

A continuación se puede ver la comparación con y sin batch normalization (teniendo dropout en ambos), para ver los efectos puros del batch normalization:

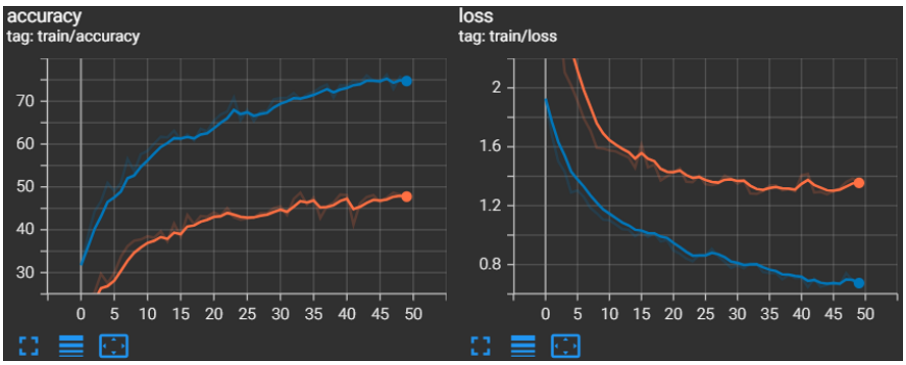

Figura 23: Estabilidad y velocidad con Batch normalization

Como se puede ver (en azul con batch normalization) la mejora es considerable, no solo la velocidad de convergencia aumento claramente, sino que también aumento la estabilidad del modelo puesto que se tienen menos oscilaciones en las curvas.

- **¿Cambió la performance de validación al combinar BatchNorm con Dropout?**

Si, y considerablemente, a continuación se puede ver como aumentó el accuracy en validation al incorporar el batch normalization, y como a su vez se obtuvo una menor loss.

Figura 24: Performance al combinar Batch normalization con Dropout

- **¿Qué combinación de regularizadores dio mejores resultados en tus pruebas?**

Según mis pruebas, la mejor combinación se dio al tener tanto dropout como batch normalization, como se vio anteriormente, al incorporar L2 el desempeño del modelo empeoró levemente.

- **¿Notaste cambios en la loss de entrenamiento al usar BatchNorm?**

La misma se reduce considerablemente, a continuación se ve la comparación con cuando solo se tenía dropout (en azul con batch normalization):

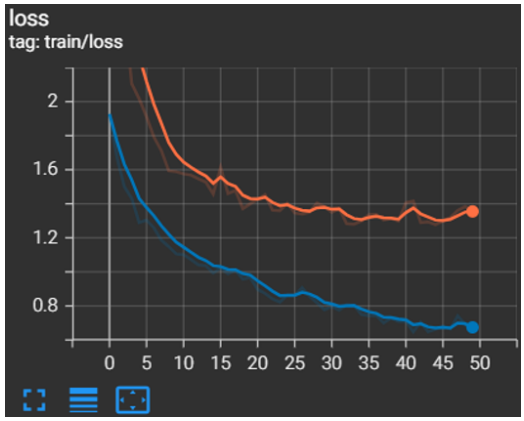

Figura 25: Cambios en la loss al usar Batch normalization

1.8. Inicialización de Parámetros

1.8.a. Preguntas teóricas

- **¿Por qué es importante la inicialización de los pesos en una red neuronal?**

Si todos los pesos se inicializan con valores muy elevados, se corre el riesgo de tener *exploding gradient*, mientras que si se los inicializa con valores muy pequeños, puede ocurrir el *vanishing gradient*. Ambos casos impactan de forma perjudicial en el aprendizaje del modelo.

- **¿Qué podría ocurrir si todos los pesos se inicializan con el mismo valor?**

Al tener esa simetría todas las neuronas van a comportarse de la misma forma, todas van a producir el mismo gradiente. Se va a obtener un accuracy extremadamente bajo, puesto que la red no es capaz de aprender, y la loss fluctuara alrededor de cierto valor permaneciendo prácticamente constante.

- **¿Cuál es la diferencia entre las inicializaciones de Xavier (Glorot) y He?**

Las inicializaciones de Xavier/Glorot fueron diseñadas para usar con activaciones sigmoideas o tangente hiperbólicas. Por otro lado, las inicializaciones de He fueron diseñadas para RELU y sus variantes. Ambas buscan que la varianza de las activaciones y gradientes se mantengan estable mientras atraviesan capas.

- **¿Por qué en una red con ReLU suele usarse la inicialización de He?**

Porque la inicialización de He está diseñada para compensar el efecto que la ReLU tiene sobre la propagación de la varianza en la red. La activación RELU descarta (estadísticamente) la mitad de los valores, puesto que pone a cero los negativos, por lo que la varianza de las activaciones se reduce aproximadamente a la mitad. Si la varianza se va achicando a medida que se avanza en las capas, las señales se vuelven cada vez mas pequeñas por lo que se puede tener vanishing gradient. La inicialización de He compensa esta pérdida definiendo:

$$\sigma^2 = \frac{2}{fan_{in}} \tag{2}$$

Siendo fan_in la cantidad de neuronas de entrada a la capa dada.

- **¿Qué capas de una red requieren inicialización explícita y cuáles no?**

Las capas que requieren de una inicialización explícita son aquellas donde la inicialización impacta en el aprendizaje, por ejemplo en las capas lineales, o las capas convolucionales. Luego por ejemplo, las funciones de activación (que si bien no forman una capa en su totalidad, son parte de ellas) no requieren de una inicialización.

1.8.b. Actividades de modificación

- **Agregar inicialización manual en el modelo**

Se modificó la clase MLP, agregando la función indicada init_weights. La misma es llamada al final del *init*

- **Probar distintas estrategias de inicialización**

Se probaron las distintas estrategias de inicialización sobre el modelo que tiene dropout, batch normalization y L2, utilizando una cantidad de 50 epochs. Se obtuvieron los siguientes resultados (comparando con el modelo indicado sin inicialización manual de parámetros en las capas lineales):

Con **inicializaciones de He** (con He es la curva rosa)

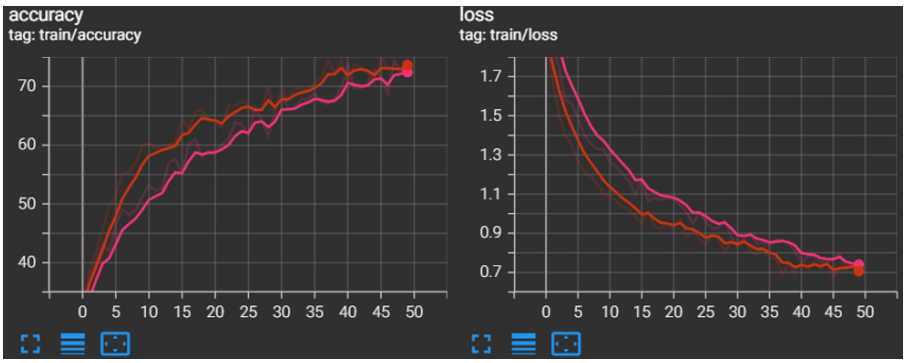

Figura 26: Comparación entrenamiento con/sin inicializaciones de He

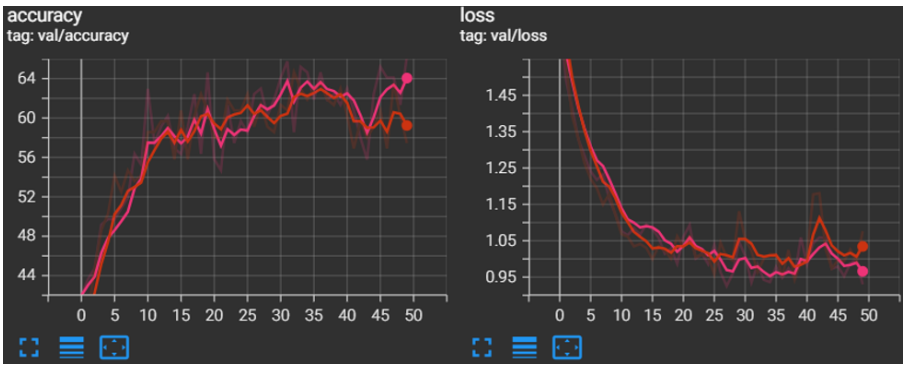

Figura 27: Comparación validación con/sin inicializaciones de He

Se ve una mejora durante la validación con respecto a no realizar la inicialización manual, tiene sentido puesto a que se utilizan activaciones RELU para las cuales la inicialización utilizada fue diseñada. Con **inicializaciones de Xavier/Glorot** (con Xavier/Glorot es la curva verde)

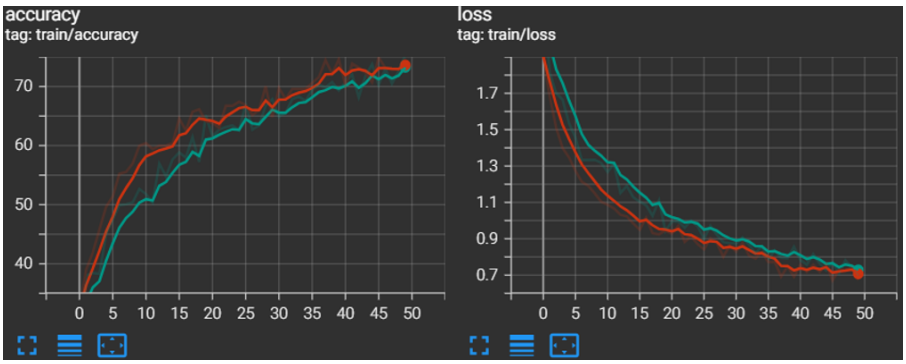

Figura 28: Comparación entrenamiento con/sin inicializaciones de Xavier

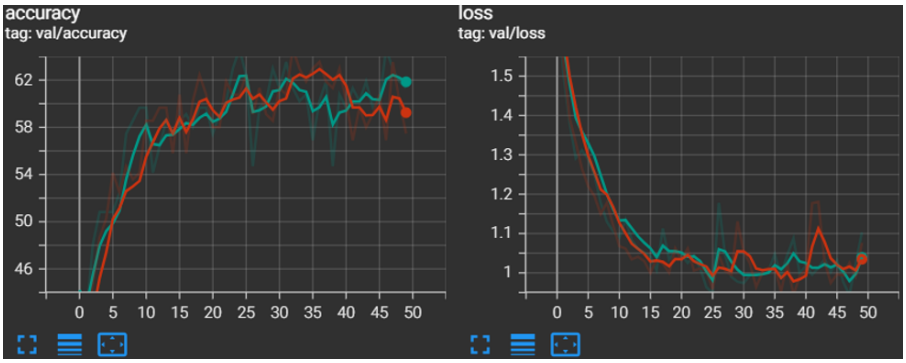

Figura 29: Comparación validación con/sin inicializaciones de Xavier

Con **inicializaciones aleatorias uniformes**

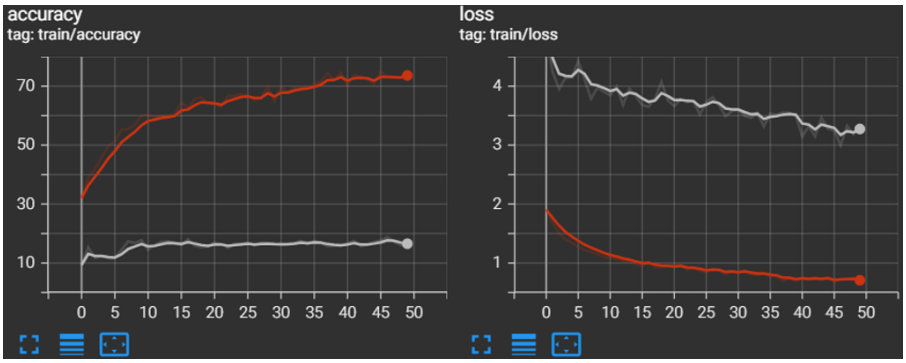

Figura 30: Comparación entrenamiento con/sin inicializaciones aleatorias uniformes

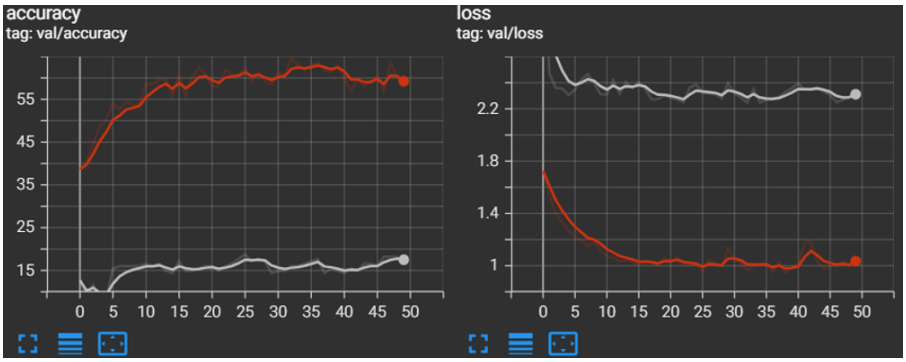

Figura 31: Comparación validación con/sin inicializaciones aleatorias uniformes

Comparación de las tres inicializaciones

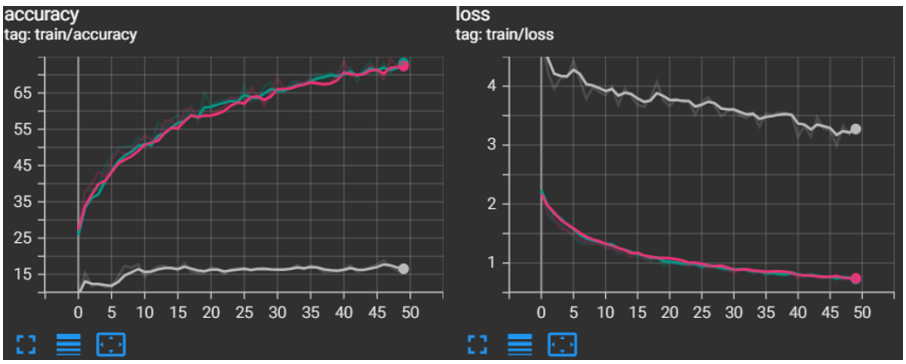

Figura 32: Comparación de las 3 inicializaciones

Como se puede ver, en lo que respecta a la velocidad durante el entrenamiento inicialización con He y con Xavier/Glorot resultan ser muy similares, mientras que inicialización aleatoria uniforme resulta mas lento en gran medida. En lo que respecta a la estabilidad, inicialización aleatoria uniforme resulta ser el menos estable, mientras que los otros 2 son establemente similares.

- **Visualizar pesos en TensorBoard**

Agregando el siguiente código:

```
if epoch == 0:
    for name, param in model.named_parameters():
        writer.add_histogram(name, param, epoch)
```

Figura 33: Visualizar pesos en TensorBoard

Y entrenando nuevamente el modelo pero con inicialización de parámetros de He, se obtienen los siguientes histogramas vistos en TensorBoard:

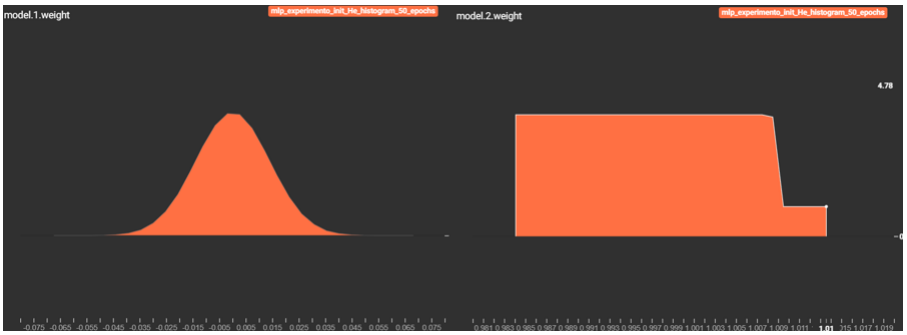

Figura 34: Pesos de la primer capa lineal y el primer BN

Siendo esos los correspondientes a la primer capa lineal y el primer batch normalization respectivamente.

1.8.c. Preguntas prácticas

- **¿Qué diferencias notaste en la convergencia del modelo según la inicialización?**

Tanto con inicializaciones de He, como con inicializaciones de Xavier/Glorot, la convergencia resulta ser muy similar, mientras que con inicializaciones aleatorias uniformes la convergencia empeora considerablemente.

- **¿Alguna inicialización provocó inestabilidad (pérdida muy alta o NaNs)?**

Con las inicializaciones aleatorias uniformes la loss aumenta en gran medida, como se pudo ver en el gráfico, siendo prácticamente el doble que en los otros dos casos.

- **¿Qué impacto tiene la inicialización sobre las métricas de validación?**

Tanto con inicializaciones de He como de Xavier/Glorot se logra un mayor accuracy en validación que si el modelo no inicializara parámetros de forma manual, y logran una menor loss en validación. Como ya se mencionó, con inicializaciones aleatorias uniformes el desempeño empeora de la misma forma que en el entrenamiento.

- **¿Por qué bias se suele inicializar en cero?**

A diferencia de los pesos, el bias no va a causar simetría entre las neuronas, simplemente va a desplazar la activación hacia arriba o hacia abajo, por lo que es seguro inicializar todos los bias en cero. Además, inicializarlos de forma aleatoria podría traer inconvenientes, si son muy elevados producirán la saturación en sigmoideas o tangentes hiperbólicas, de la misma forma que si son muy negativos con activaciones RELU se apagaran neuronas, disminuyendo la convergencia del modelo.

2. Búsequeda de Hiperparámetros con MLP

Correr todos los modelos para la búsqueda de hiperparámetro tardo:

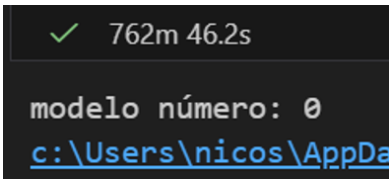

Figura 35: Tiempo de entrenamiento

Para comparar los 88 modelos analizados es de gran utilidad el MLFlow, a continuación se pueden ver resultados obtenidos:

2.1. Variando el tamaño de los batch

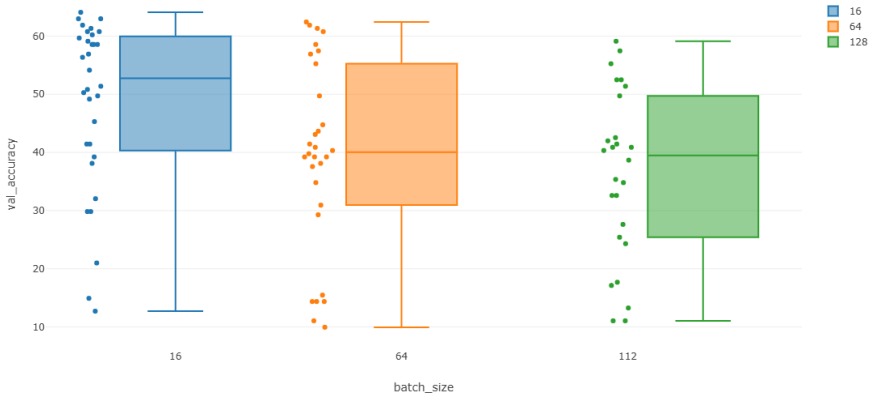

Figura 36: Accuracy en validación para distintos batch size

Se puede ver que el accuracy en validación resulta mayor para batch size menores, siendo el mejor caso con batch size de 16.

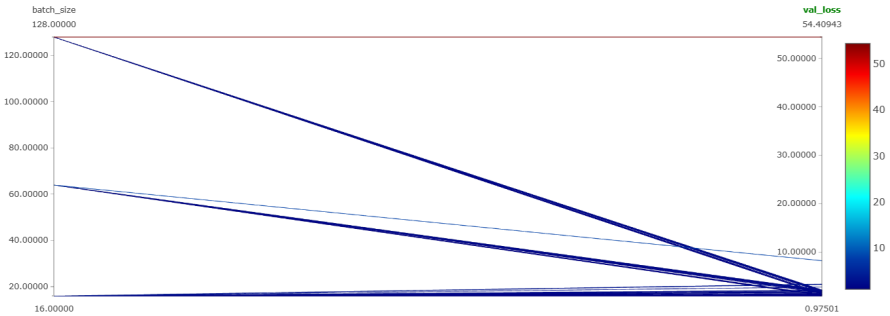

Figura 37: Loss en validacion para distintos batch size

En lo que respecta a la loss en validación para los distintos batch size el comportamiento suele ser un tanto similar, todos logran alcanzar valores pequeños, la diferencia radica en la velocidad de convergencia.

2.2. Variando el dropout

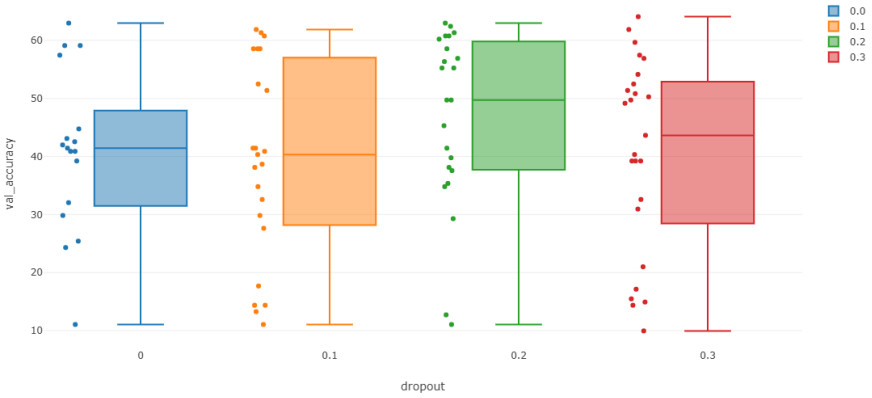

Figura 38: Accuracy en validación para distintos dropout

En lo que respecta a la accuracy en la validación, se obtuvieron los mejores resultados con un dropout de 0.2.

2.3. Variando el learning rate

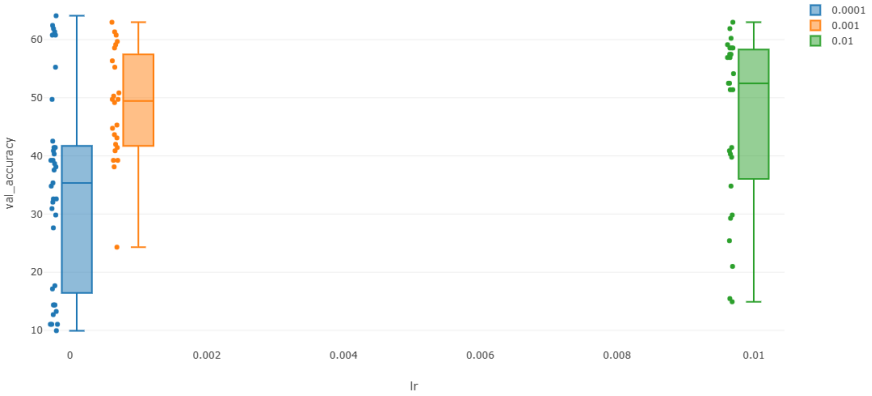

Figura 39: Accuracy en validación para distintos learning rate

Si bien el mejor modelo para los distintos learning rates, desde el punto de vista del accuracy de la validacion, fue con el menor valor de $lr = 0{,}0001$, notar como tambien es el que mas dispersión tiene. Por lo que el mejor caso resultó ser aquel con $lr = 0{,}001$, siendo este el mas estable.

2.4. Mejor modelo

Dentro de los 88 modelos entrenados, el que dio mejores resultados fue el siguiente:

- $HFlip = VFlip = RBContrast = 0$
- $batch\_size = 16$
- $dropout = 0.3$
- $input\_size = 32$
- $lr = 0.0001$
- $optimizer = Adam$

El entrenamiento del mismo tardo 7.4 minutos, deteniendo el mismo a los 29 epochs, dado que el mejor epoch se dió en el número 22, obteniendo los siguientes resultados:

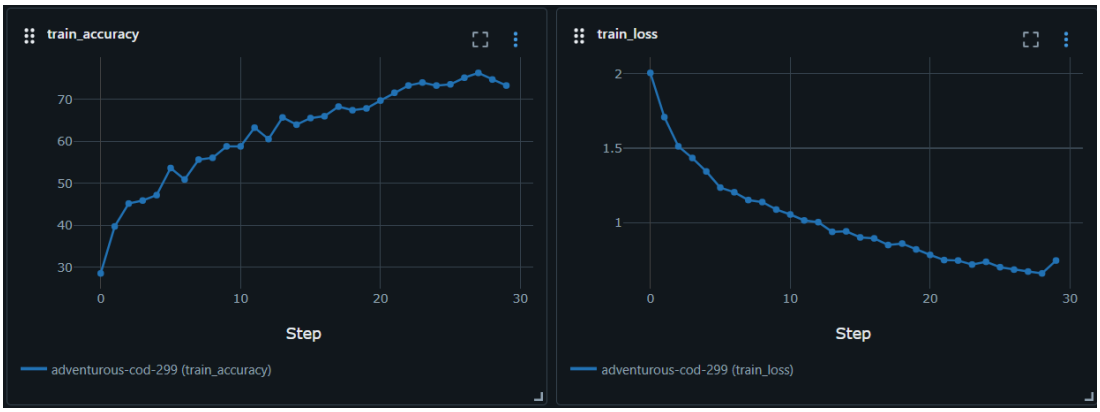

Figura 40: Mejor modelo durante el entrenamiento

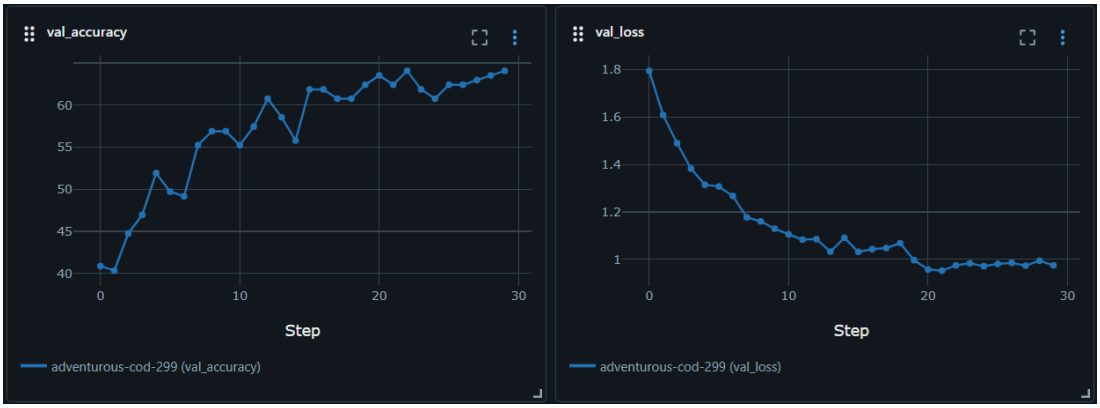

Figura 41: Mejor modelo durante la validación

Los valores máximos del accuracy resultaron ser de 73.31 para el entrenamiento y 64.09 durante la validación. Mientras que los mínimos valores de la loss fueron 0.746 en el entrenamiento y 0.975 en la validación.

3. Búsequeda de Hiperparámetros con CNN

Pasando a la PC (que tiene una mejor CPU y una GPU externa) para aprovechar los recursos de la misma y reducir los tiempos de entrenamiento, entrenar un total de 75 modelos para la búsqueda de hiperparámetros con *CNN* tardo 307 minutos.

3.1. Variando el tamaño de los batch

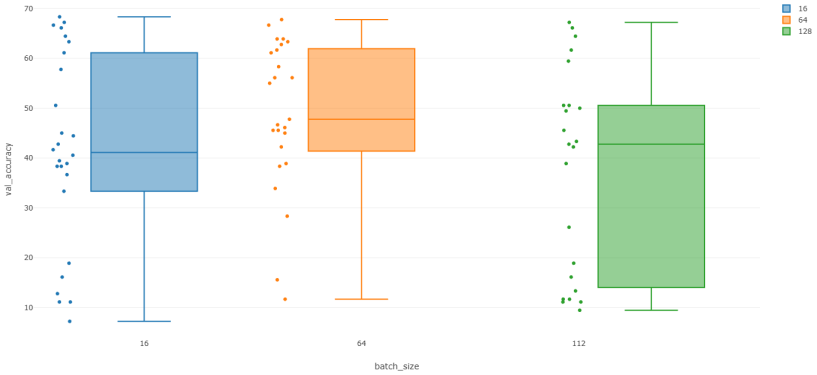

Figura 42: Accuracy en validación para distintos batch size

3.2. Variando el dropout

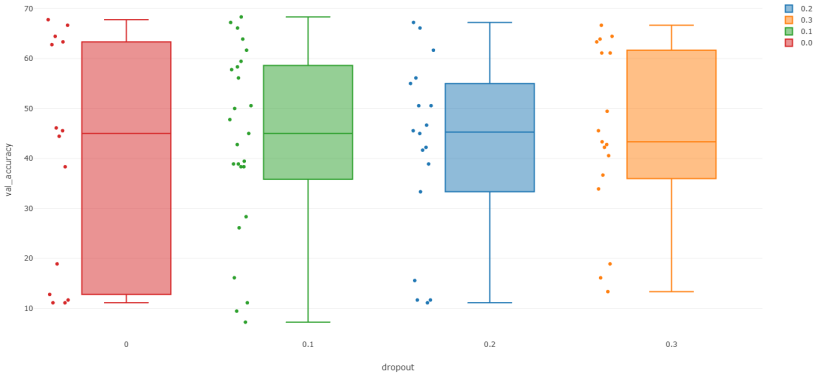

Figura 43: Accuracy en validación para distintos dropout

3.3. Variando el learning rate

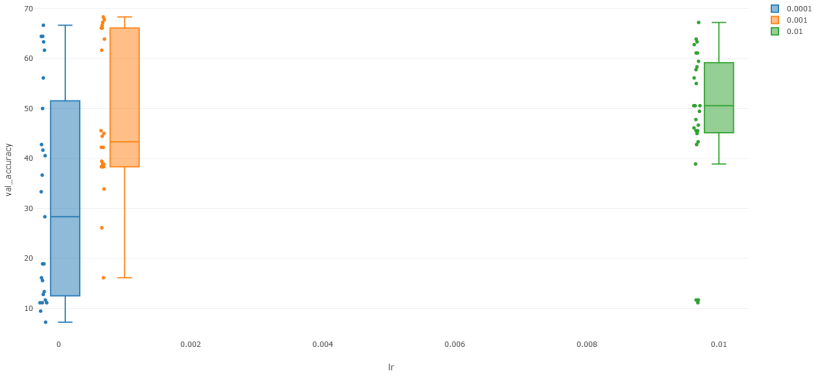

Figura 44: Accuracy en validación para distintos learning rate

3.4. Mejor modelo

Los 2 mejores modelos que se encontraron tienen las siguientes características:

	Modelo 1	Modelo 2
HFlip	0	0
VFlip	0.5	0
RBContrast	0	0
input size	128	128
batch size	128	16
LR	0.001	0.0001
Optimizer	*Adam*	*Adam*

Cuadro 1: Características de cada modelo

Dando los siguientes resultados:

	Modelo 1	Modelo 2
Acc (train)	83.62	91.95
Loss (train)	0.458	0.263
Acc (val)	67.22	66.67
Loss (val)	0.936	0.988

Cuadro 2: Máxima accuracy y mínima loss para cada modelo

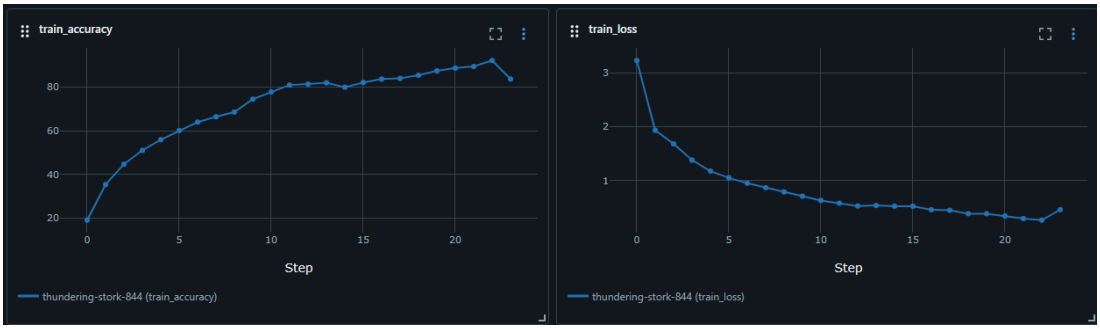

Figura 45: Modelo 1 durante el entrenamiento

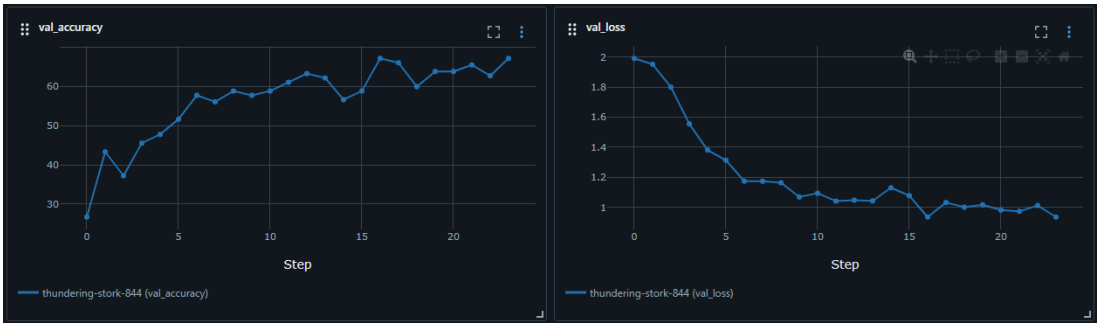

Figura 46: Modelo 1 durante la validación

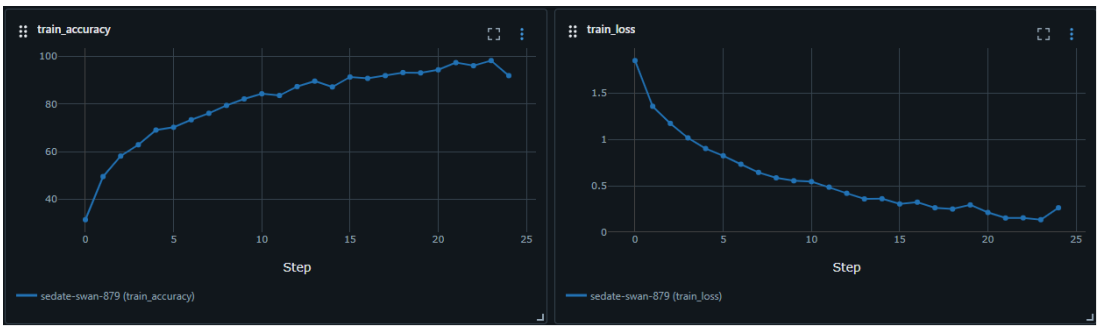

Figura 47: Modelo 2 durante el entrenamiento

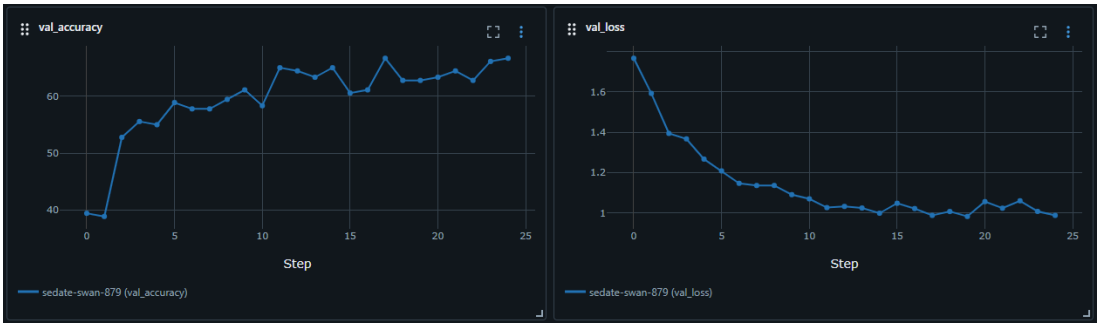

Figura 48: Modelo 2 durante la validación

Ambos resultaron ser muy similares y no se notan mejoras significativas uno frente al otro.