# Proyecto 2: Retinopatía diabética

Ignacio Boero CI: 5.150.913-2 Santiago Díaz CI: 5.172.004-5 Tomás Vázquez CI: 5.020.594-1

## I. Introducción

El problema consiste en la automatización de la detección de retinopatía diabética en pacientes a partir de imágenes de alta resolución de la retina de los ojos. Esta enfermedad es la principal causa de ceguera, y se estima que afecta al menos 93 millones de personas. Esta se desarrolla generalmente como consecuencia de la diabetes. Hoy en día el proceso de detección se realiza manualmente, y consiste en un profesional el cual analiza la imagen en busca de anomalías vasculares. Este proceso es efectivo, sin embargo presenta la limitación de la necesidad del tiempo de un profesional, lo que muchas veces genera que un estudio esté en espera horas, días, o hasta semanas antes de poder ser analizado por un profesional.

Es por esto que se plantea la automatización de esta tarea. Para esto se dispone de una base de datos la cual consiste en un conjunto de imágenes de retinas etiquetadas con un nivel de 0 a 4, donde a mayor número corresponde a una mayor etapa de desarrollo de la enfermedad. Para lograr esto se propone el uso de redes neuronales convolucionales.

La retinopatía diabética ocurre cuando los vasos sanguíneos dañados filtran sangre y otros líquidos en la retina, causando hinchazón y visión borrosa. Los vasos sanguíneos pueden obstruirse, pueden desarrollarse tejidos cicatriciales y, finalmente, puede producirse un desprendimiento de retina. La afección es más fácil de tratar en las etapas tempranas, por eso es importante someterse a exámenes oculares de rutina. Los síntomas que comúnmente presentan los pacientes que son posibles de detectar mediante las imágenes de fondo de ojo son: hemorragias internas, exudados duros, manchas de algodón, crecimiento anormal de vasos sanguíneos y aneurisma. Estos síntomas pueden visualizarse en la Figura 1.

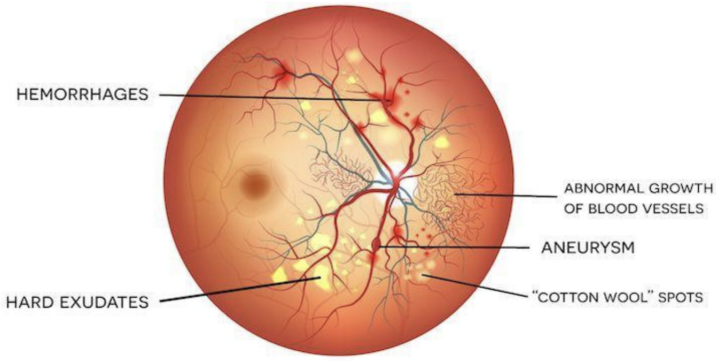


Figura 1: Posibles síntomas de un paciente con retinopatía diabética.

## II. Datos disponibles

### *II-A. Descripción de datos*

Los datos de entrenamiento consisten en 23 archivos con formato "TfRecord", donde los 22 primeros cuentan con 1536 imágenes cada uno, mientras que el último cuenta con 1334. Por lo tanto, se cuenta con un total de 35.126 imágenes a color de retinas de ojo humano. Cada una de estas imágenes presenta un ancho de 1024 y una altura que varía entre 640 y 730 píxeles.

Cada imagen contiene además un índice el cual corresponde al número de paciente, una etiqueta que indica si el ojo corresponde al izquierdo o al derecho, y otra que indica el grado de desarrollo de la enfermedad. Esta etiqueta toma valores enteros del 0 al 4, donde a mayor número mayor grado de retinopatía diabética (RD). Cualitativamente, estas características representan:

- 0: No hay RD
- 1: RD leve
- 2: RD moderada
- 3: RD severa
- 4: RD proliferativa

Se observa un ejemplo por categoría en la Figura 2.

### *II-B. Train vs Validación*

Un detalle importante es que todos los pacientes presentan una imagen por ojo. Es importante tener esto en cuenta a la hora de separar en conjuntos de entrenamiento y validación, pues las imágenes de los ojos de una misma persona están muy correlacionadas, por lo que tener un ojo en entrenamiento y otro en validación cae en contaminación de datos. Sin embargo, los datos están ordenados de forma que los ojos de una misma persona son consecutivos, por lo que simplemente separando los dos conjuntos antes de mezclar los datos es suficiente para que no queden contaminados.

Para la división de conjuntos se toman 28800 imágenes para el conjunto de entrenamiento, y 6326 para validación. Para esto supusimos que los distintos archivos tienen características similares. Esto puede llegar a ser un problema a futuro en caso de que los archivos estén ordenados de alguna manera

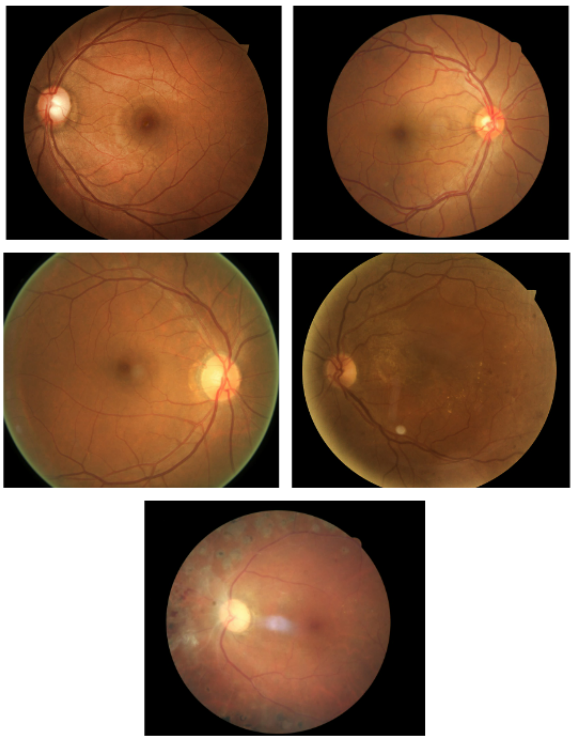

Figura 2: Ejemplo de imágenes para cada una de las cinco categorías. La categoría aumenta en sentido derecha/abajo.

en particular, por ejemplo que cada archivo corresponda a imágenes tomadas con distintos equipos, por lo que es algo a tener presente más adelante.

Una vez separados los datos en estos conjuntos, se visualizan las etiquetas de estos conjuntos mediante el histograma de la Figura 3. Como se puede observar, existe un desbalance de clases. Además, se observa que la distribución de datos por categoría es similar en entrenamiento y validación, lo cual es deseable.

### *II-C. Levantar los datos*

Debido a la gran cantidad de datos a trabajar, estos se presentan en formato de TFrecords. Este formato guarda todos los datos de forma secuencial guardando carácter por carácter. Esto permite reducir el tamaño total de almacenamiento que ocupan los datos, y es estándar cuando se trabaja con Tensor Flow. Esta librería tiene varios métodos los cuales permiten tanto cargar como guardar los datos originales en este formato. En particular, se utiliza la función **tf.data.TFRecordDataset**, la cual permite leer los datos del TFrecord transformándolos en un Dataset. Luego se utilizan un par de métodos más los cuales permiten obtener por separado cada dato y transformar las imágenes de texto plano a formato jpeg.

## III. Estrategias utilizadas

### *III-A. Preprocesamiento*

Consideramos el preprocesamiento uno de los puntos más importantes y complejos del proyecto. En muy alto nivel,

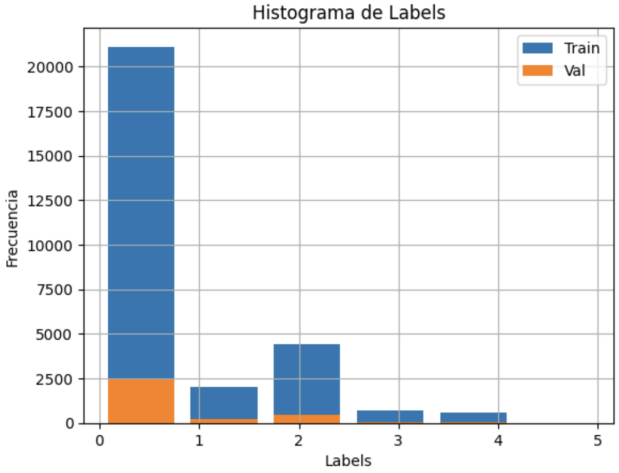


Figura 3: Histograma de las categorías para los conjuntos de entrenamiento y validación.

los algoritmos de aprendizaje automático buscan aprender patrones que se repiten entre datos de una misma clase, y que son exclusivos de esta. Distintos algoritmos buscan patrones de formas distintas. Por lo tanto, la facilidad que va a tener un modelo para aprender a realizar la tarea va a depender, entre otras cosas, de la varianza entre datos de clases distintas, comparada con la varianza que presentan datos de una misma clase. Lo que debe buscar el preprocesamiento entonces es aumentar la varianza para datos de categorías distintas, o disminuir la varianza entre datos de la misma clase.

En nuestro problema en particular, observamos que existe mucha varianza entre imágenes debido a diferentes condiciones en la cual fue tomada la imagen. Esto provoca que imágenes de ojos de una misma categoría pero tomadas por cámaras diferentes sean más distintas que imágenes de ojos de una distinta categoría pero tomadas con la misma cámara. En particular, observamos que hay dos razones que dominan la diferencia entre imágenes tomadas por cámaras distintas. En primer lugar, está la iluminación y contraste. En segundo lugar está el tamaño de los ojos. Por lo tanto, el preprocesado utilizado busca disminuir estos efectos.

Para lograr esto se utilizan dos estrategias inspiradas en el trabajo de Ben Graham [1], el ganador de la competencia de Kaggle en 2015. En primer lugar se busca homogeneizar el tamaño en píxeles de los ojos. Para esto se calcula el radio del ojo en píxeles, para luego escalar la imagen de forma tal que los ojos tengan todos un radio de 300 píxeles. Finalmente, se recorta la imagen a tamaño 600x600. De esta forma se tiene que todas las imágenes son del mismo tamaño, pero más importante, se tiene que todos los ojos son aproximadamente del mismo tamaño. Además, al recortar de ese tamaño, se están eliminando los bordes, que es ruido que le estamos sacando al modelo. De esta manera, se sabe que el ojo abarca toda la imagen.

Una vez logrado esto, se ataca el problema de iluminación y contraste. Una forma sencilla de corregir la iluminación es restarle a cada canal el valor medio y sumarle 127. Esto logra que todos los valores tengan el mismo valor medio, corrigiendo la iluminación. Sin embargo esto no cambia el contraste. Para esto se utiliza un método más sofisticado que restar la media, pero que la idea es la misma. Esta es restarle a la imagen, la imagen original pasada por un filtro gaussiano. Esto es similar a restarle a cada píxel la media de los píxeles vecinos. Qué tan grande se toma el vecinaje para realizar este promedio depende del parámetro $\sigma^2$ de la gaussiana utilizada. De esta forma, se logra homogeneizar tanto la iluminación como el contraste entre imágenes. En nuestro caso, el valor elegido fue $\sigma^2 = 60$.

Un problema que encontramos al implementar este preprocesamiento, es que el tiempo de preprocesado aumenta drásticamente el tiempo de entrenamiento, a tal punto de que no es posible entrenar en un tiempo razonable, estimando que demoraba más de un día en correr una época. Identificamos que el problema de esto es la convolución con la gaussiana a la hora de aplicar este filtro. Como primer solución implementamos aplicar el filtro mediante el uso de transformadas de Fourier. Esto permite acelerar el proceso, sin embargo el resultado obtenido no es idéntico que convolucionar. En particular, utilizar la FFT provoca efectos de borde los cuales hacen saturar la imagen en algunas zonas. Por lo tanto, utilizamos una segunda solución, la cual es generar un nuevo dataset con imágenes preprocesadas mediante el uso de la convolución. Esta tarea demora más de un día, sin embargo solo debe realizarse una vez. Por lo tanto generamos 23 TFRecords, con las imágenes ahora preprocesadas.

Un ejemplo de la comparación entre la imagen original, el preprocesado con la FFT y el preprocesado final con la convolución con la gaussiana se puede ver en la Figura 4. Se logra ver como para ambos preprocesamientos implementados se resaltan los efectos que se quieren identificar. En la imagen de la derecha vemos el primer método, en el que se utilizó la FFT, donde se producen efectos de borde no deseados. En la imagen inferior vemos el resultado del método basado en la convolución. Se logra un resultado mejor, que además de no tener efectos de borde resalta aún más los daños del ojo.

### *III-B. Desbalance de clases*

Como fue mencionado anteriormente, los datos se encuentran desbalanceados:

- Clase 0: 73,5 %
- Clase 1: 7,0 %
- Clase 2: 15,1
- Clase 3: 2,4 %
- Clase 4: 2,0 %

A priori lo que pensamos es que si no se aplica ninguna técnica para lidiar con este desbalance, podría pasar que el modelo tienda a sobreajustarse a las clase mayoritaria, desencadenando en un modelo débil frente a las clases minoritarias. Una estrategia que se utiliza mucho es entrenar con pesos, penalizando con mayor peso los errores de las clases minoritarias. Una variante muy utilizada es tomar los pesos como el logaritmo del inverso de la proporción correspondiente a cada clase. Esto resulta en los siguientes pesos, que son pasados al modelo a la hora de entrenar mediante el parámetro "class weights":

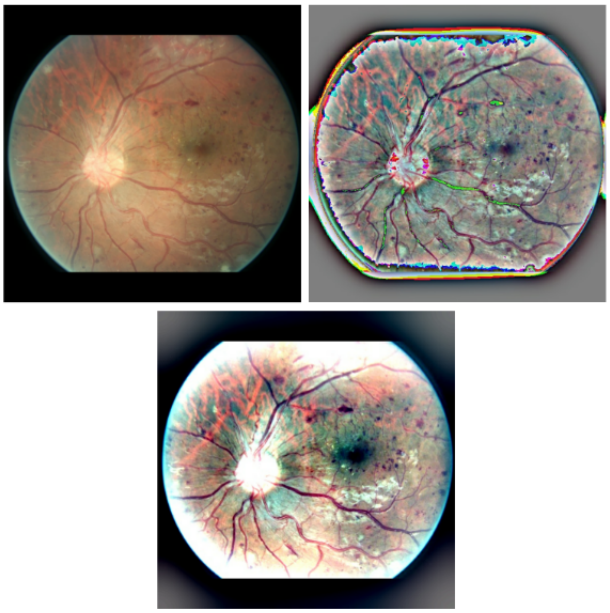

Figura 4: Comparación de los preprocesamientos

- Peso para clase 0: 0,1338
- Peso para clase 1: 1,1577
- Peso para clase 2: 0,8220
- Peso para clase 3: 1,6046
- Peso para clase 4: 1,6956

Esto también presenta ventajas en el ámbito médico. Por precaución, es preferible que el algoritmo diagnostique a un paciente con la enfermedad y que luego sea evaluado por un especialista, que no diagnosticar un paciente que efectivamente tiene la enfermedad.

### *III-C. Aumento de datos*

El aumento de datos es una estrategia que se utiliza para la regularización de modelos. Consta de generar nuevos datos para entrenamiento a partir de transformaciones de los datos originales. Esto permite ser robusto ante este tipo de variaciones al recibir nuevos datos.

Para imágenes, las transformaciones que generalmente se busca que la red convolucional tenga robustez son rotaciones, traslaciones, escalado, cambios en iluminación o contraste y ruido. En nuestro caso, se utiliza un preprocesado que iguala el tamaño de los ojos en la imagen y los coloca en el centro, por lo que no es necesario que el modelo sea robusto

a escalado o traslación. Lo mismo ocurre con la iluminación.

Por lo tanto, las estrategias de aumento de datos que realizamos son rotaciones, cambios de contraste y generación de ruido gaussiano en la imagen. El preprocesado busca aumentar el contraste, sin embargo no es igual en todas las imágenes, por lo que también los incluimos. Además, se agrega con probabilidad de 0.5 invertir la imagen con respecto al eje vertical u horizontal. De esta forma, se espera tener un modelo con mayor capacidad de generalización.

Una vez generada la capa de aumento de datos, se observan los resultados de aumentar una imagen, para visualizar correctamente su función. En la Figura 5 se puede ver las distintas transformaciones para una imagen aleatoria, en donde se utilizaron flips horizontales y verticales, rotaciones, cambios de contraste, cambios de brillo y adición de ruido gaussiano. Como se puede observar, las rotaciones introducen ruido a las imágenes, ya que la distancia de los bordes de la imagen al borde del ojo no es constante. Esto genera que al rotar, se deban rellenar esos espacios. Se podría elegir la opción de rellenar estos espacios con un valor constante, por ejemplo 0, pero esto también agrega ruido, ya que el fondo de las imágenes no es exactamente 0 debido al preprocesamiento realizado. Por lo tanto, decidimos descartar las rotaciones, manteniendo las técnicas mencionadas anteriormente. El factor de contraste utilizado es 0.25, el de brillo 0.1 y el $\sigma$ del ruido gaussiano como 1.

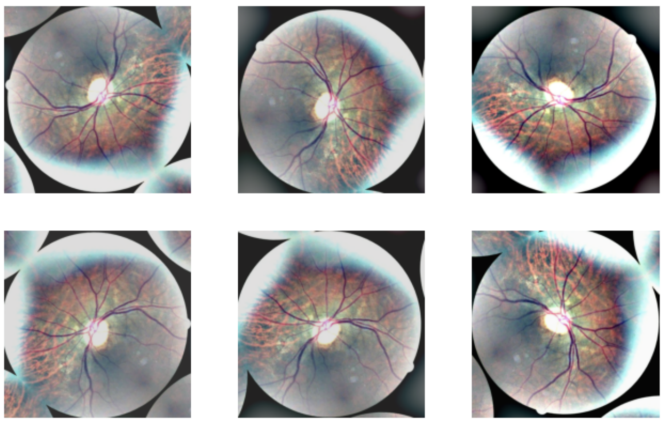

Figura 5: Aumento de datos

### *III-D. Métricas y Losses*

El problema fue abordado como uno de clasificación, aunque también se analizó la posibilidad de abordarlo como un problema de regresión. Como función objetivo se utilizó la "sparse_categorical_crossentropy" comúnmente utilizada en problemas de clasificación entre varias clases, mientras que la "accuracy" y la "quadratic weighted kappa" fueron elegidas como métrica.

La "quadratic kappa" es una medida de concordancia entre 2 clasificaciones. Una clasificación corresponde a las etiquetas que fueron analizadas previamente por especialistas, mientras que la otra clasificación corresponde a las predicciones que realiza nuestro algoritmo. Esta métrica puede valer entre -1 y 1, aunque típicamente toma valores entre 0 y 1, donde un valor de kappa de -1 indica un desacuerdo total entre la predicción y la etiqueta, 0 indica desacuerdo aleatorio y 1 acuerdo total. La manera de calcular este valor debe tener en cuenta la distancia entre las etiquetas "verdaderas" y las de nuestro algoritmo, ya que no es lo mismo predecir como 0 un paciente que en realidad tiene grado de retinopatía 4 que predecir un 3 a un paciente que tiene nivel 4 de enfermedad. Para tener esto en cuenta es que se utiliza la variante "weighted".

El objetivo de este trabajo es maximizar el valor de kappa, que luego es el que será utilizado por Kaggle para evaluar en el conjunto de test. Por esto mismo proponemos también el uso de esta métrica como loss para ver si tiene un mejor desempeño que la "sparse_categorical_crossentropy". Esta métrica es muy utilizada en contextos que relacionan la medicina con aprendizaje automático.

### *III-E. Transfer Learning*

Durante este proyecto se utiliza principalmente la estrategia de transfer learning para el entrenamiento de modelos. Esta consiste en tomar una arquitectura con pesos que fueron entrenados con datos distintos, en una tarea distinta, y reentrenar las últimas capas con los datos disponibles, de forma que se adapte al problema en particular a resolver. Esto se basa en que las primeras capas de las redes convolucionales aprenden filtros genéricos, como bordes o figuras geométricas, los que pueden ser reusados para nuestro problema con un buen desempeño.

Para el problema de clasificación de retinopatía excluimos la capa superior, e incluimos una capa final densa con 5 salidas y activación softmax luego de un Global Average Pooling. El entrenamiento se hace en dos etapas. En la primera se entrena únicamente la última capa densa agregada, fijando los pesos usados por transferencia. Para esto se usa un paso de aprendizaje más alto y durante unas pocas épocas. Una vez que el rendimiento es aceptable, se entrenan las capas medias y finales del modelo. Esto se hace con paso de aprendizaje más chico y durante varias épocas. De esta forma se logra adaptar una arquitectura previamente entrenada con otro problema, al nuestro.

### *III-F. Grad-Cam*

Aplicar un algoritmo del estilo Grad-Cam a los modelos implementados es importante para visualizar y entender qué partes de una imagen son relevantes para la clasificación realizada por la red. Esto proporciona una forma de interpretar y explicar las decisiones tomadas por la misma.

El algoritmo de Grad-Cam utiliza el modelo entrenado para obtener una predicción de una imagen. Luego, utiliza los gradientes de la salida de la red con respecto a la clase objetivo para ponderar las activaciones de la última capa de convolución y generar un mapa de activación. Se "visualiza" la última capa de convolución ya que es la que tiene mayor representación de las características aprendidas. Por último, genera un mapa de calor que indica qué partes de la imagen fueron más relevantes para la clasificación realizada. [2]

## IV. Modelos y Resultados

En esta sección se presentan los distintos modelos entrenados. Para cada uno de ellos se registraron las métricas en Comet [1]. Para identificar cada experimento, se utilizó una nomenclatura para el nombre de cada experimento. Por ejemplo, si se entrena una ResNet50 con aumento de datos, preprocesamiento, se entrena utilizando los pesos, un learning rate de 0.001 para entrenar la capa densa y un learning rate de 0.003 para entrenar las últimas capas del modelo, el nombre del experimento será el siguiente: "ResNet50-pre-aug-weights-1e-3-3e-3".

### *IV-A. Callbacks utilizados*

Dada la cantidad de imágenes que se tiene para entrenar y las arquitecturas que utilizamos, los tiempos de entrenamiento son significativos. Dicho esto, se utilizan algunos callbacks:

- Model Checkpoint: en caso de que la ejecución sea interrumpida, podemos levantar los pesos y continuar entrenando el modelo
- Early stopping: para evitar que el modelo siga entrenando cuando la cohen kappa deja de mejorar en el conjunto de validación, se utiliza early stopping. Esto sirve en primer lugar como técnica de regularización, ya que se evita que el modelo siga mejorando en el conjunto de entrenamiento cuando en realidad ya dejó de mejorar en validación, y además para que los tiempos de ejecución se vean reducidos.

### *IV-B. Transfer Learning, Métricas y Losses*

En esta primera etapa se busca tener un modelo sencillo pero funcional, teniendo un primer pipeline el cual mejorar. Para esto se trabaja con una cantidad reducida de datos para agilizar el entrenamiento, teniendo 6400 imágenes para entrenamiento y 3200 para validación.

Para esto se prueban tres arquitecturas mediante transfer learning: Xception, ResNet50 y VGG16. Para el preprocesado de los datos, únicamente se escalan las imágenes de forma que todas queden de igual tamaño, $640 \times 1024$. Para las métricas, se visualizan tanto el porcentaje de aciertos como el Cohen Kappa, utilizando el segundo como métrica primaria (la que se observa para decidir EarlyStopping, o guardado de modelos). Para las losses, se prueban dos estrategias diferentes. En un primer lugar, se utiliza la "sparse_categorical_loss", pero también se prueba usar una loss derivada de la métrica de cohen kappa.

Se observó que utilizando la loss derivada del cohen kappa, los modelos no logran aprender. Observando la documentación de esta loss, se observa que dice que la librería a la que pertenece está descontinuada, por lo tanto, se decidió no seguir utilizando esta loss, y quedarse únicamente con la "sparse_categorical_crossentropy".

Con esto se tienen los primeros tres modelos a entrenar. Se entrenan 2 épocas con todas menos la última capa fijas, con paso de aprendizaje de $0{,}0005$, y luego 3 épocas habilitando a entrenar todas menos las primeras 56 capas, con paso de aprendizaje de $0{,}0002$. Los resultados se observan en la Tabla I. Hay que destacar que el resultado aparentemente tan bueno del modelo ResNet50 no es representativo de la realidad, ya que solo entrenamos con los primeros 400 batches y validamos con los siguientes 200.

Respecto a los modelos, se observa que el desempeño de la ResNet50 es muy superior al resto. Por lo tanto se decide continuar con este modelo en lo que sigue del proyecto. Por lo tanto, se utiliza este modelo y la loss previamente mencionada para entrenar con todos los datos. El resultado de esto es un Cohen Kappa de 0.723 en train y 0.677 en validación.

| Modelo | Kappa Train | Kappa Val |
|---|---|---|
| ResNet50 | 0.80 | 0.76 |
| VGG16 | 0.24 | 0.25 |
| Xception | 0.05 | 0.09 |

Cuadro I: Desempeño de tres modelos probados, para 400 batches de train y 200 de validación

### *IV-C. Regularización Mediante Aumento de datos*

Observando el resultado de entrenar con todos los datos en la sección anterior, se observa que el modelo está sobreajustado. En el curso hemos visto varias estrategias para disminuir el error de generalización, donde la principal es disminuir la capacidad expresiva del modelo. En nuestro caso, observamos que esto puede ser complejo, ya que implica modificar una arquitectura estándar como es la ResNet50. Por lo tanto, se utiliza la segunda estrategia más popular, que es aumentar la cantidad de datos. Para esto se utilizan las transformaciones mencionadas en la sección de aumento de datos.

Se entrena de igual manera que el último modelo de la sección anterior, solo que agregando una capa de aumento de datos previa al inicio del modelo. El desempeño en esta ocasión en train es de 0.736 y en validación 0.720 . Se observa cómo el modelo está mucho menos sobreajustado, y además mejora el rendimiento. Por lo tanto el aumento de datos es una estrategia exitosa, que se mantiene para el resto de modelos a entrenar.

[1]Link a Comet:https://www.comet.com/tomasvazquez99/retinopatia/view/new/panels

### *IV-D. Preprocesamiento y Desbalance de Clases para Mejorar el Modelo*

Una vez conseguido un primer modelo aceptable, se pasa a observar el desempeño al agregar el preprocesamiento diseñado. Como explicamos en la sección de preprocesamiento, utilizamos los TFrecords generados por nosotros, los cuales buscan igualar el tamaño del ojo, la iluminación y el contraste. De esta manera, se utilizan estos datos para entrenar el modelo de ResNet50 y aumento de datos. Los resultados en la métrica de Kappa son de 0.78 en train y 0.74 en validación.

Comparando con el modelo sin este preprocesamiento, se observa cómo el desempeño mejora. Por lo tanto, se considera útil el uso de este preprocesamiento. Sin embargo, todavía se tiene espacio para mejora. Para esto, se calcula la matriz de confusión en validación, la cual se observa en la Figura 6.

Se puede observar que el modelo se inclina más a predecir los valores 0 y 2, que son los que más imágenes tiene. Por tal razón, creemos que es una buena idea entrenar penalizando en mayor medida los errores de la clases minoritarias agregando el parámetro class_weight a la hora de entrenar. El peso que se le da a cada error es el mencionado en la sección de desbalance de clases.

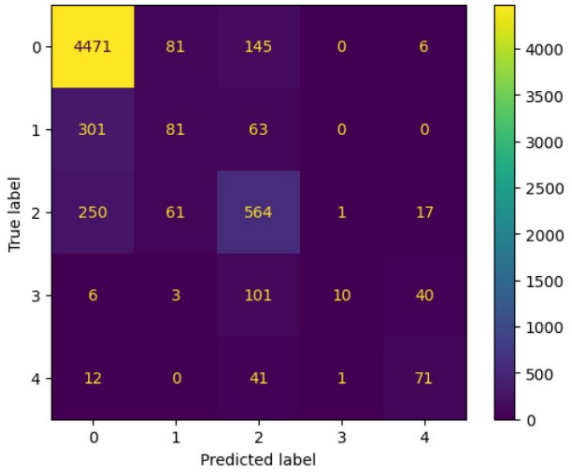


Figura 6: Matriz de confusión en validación para el modelo ResNet50 entrenada sin class_weight.

Se entrena el mismo modelo con los pesos, y se obtiene un desempeño en Kappa de 0.61 en train y 0.57 en validación. También se observa la matriz de confusión en la Figura 7. Se tiene que el desempeño empeora bastante. Observando la nueva matriz de confusión, tenemos que las predicciones por clase ahora están más balanceadas, pues se predicen más cantidad de 1, 3 y 4. Sin embargo, la cantidad de estos es muy baja comparada con la cantidad de 0 y 2, por lo que se tienen más errores.

De esto deducimos que el uso de pesos para balancear las clases se debe utilizar cuando el desbalance existe en los datos de entrenamiento, pero no en la realidad. En nuestro caso, el desbalance probablemente ocurra también en la realidad, por lo que balancear las clases puede resultar contraproducente al evaluar el desempeño. Es por esto que decidimos descartar el uso de pesos para futuros modelos.

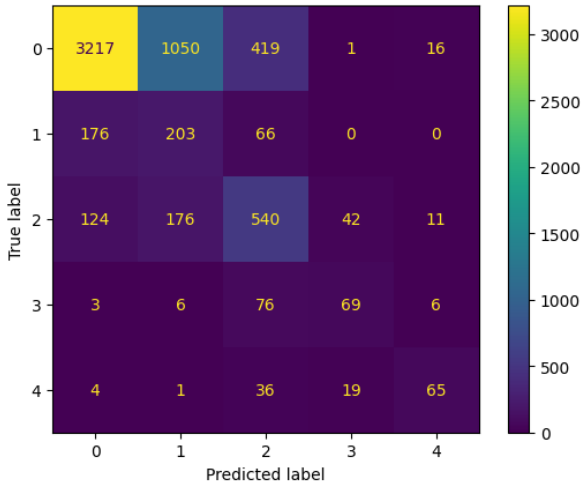


Figura 7: Matriz de confusión en validación para el modelo ResNet50 entrenada con class_weight.

### *IV-E. Arquitectura ResNet50-V2*

Como siguiente estrategia para mejorar el desempeño, se prueba con utilizar la segunda versión de ResNet50, llamada ResNet50V2. Esta se entrena con todos los datos, preprocesado y aumento de datos tal como explicado previamente. El resultado de esta, comparado con el de ResNet50, se observa en la Tabla II. Podemos observar que el mejor desempeño lo sigue teniendo la arquitectura ResNet50, por lo que continuamos trabajando con esta.

| | Kappa entrenamiento | Kappa validación |
|---|---|---|
| ResNet50 | 0.78 | 0.74 |
| ResNet50V2 | 0.65 | 0.57 |

Cuadro II: tabla de comparación entre ResNet50 y ResNet50V2.

### *IV-F. Análisis con Grad-Cam*

Para seguir buscando cómo poder mejorar el modelo, se propone observar la técnica de Grad-Cam para entender cómo toma las decisiones el clasificador. Al observar esto para distintas imágenes, concluimos que el clasificador trabaja correctamente. Un ejemplo se puede ver en la Figura 8. En este caso la predicción fue un 4, lo cual es correcto. Se observa cómo el modelo se concentra más en las partes donde el ojo está más dañado, siendo esa parte de la imagen la más usada para clasificar la misma.

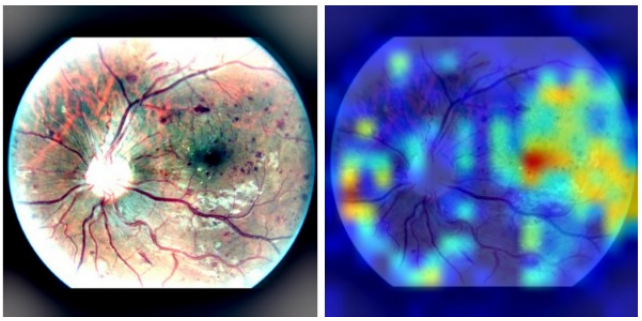

Figura 8: Grad-Cam implementado que muestra la decisión tomada por la última capa convolucional en uno de los modelos finales entrenados

Por lo tanto, este análisis nos confirma que el modelo entrena de forma correcta, pero no nos proporciona ninguna estrategia para seguir mejorándolo.

### *IV-G. Ajuste de hiperparámetros*

Una vez que ya tenemos el modelo a utilizar (ResNet50), el preprocesamiento definido, la estrategia de aumento de datos y cómo lidiar con el desbalance (no utilizar pesos), hacemos una búsqueda de hiperparámetros. El valor del learning rate decidimos no probar con distintos valores, ya que en la gráfica de la loss visualizada en Comet, vemos que la velocidad de descenso de la misma es razonable. Un hiperparámetro que decidimos variar es el tamaño del batch. En un principio se había utilizado 16, pero probamos también con 32. En la Tabla III se puede ver una comparación de los resultados obtenidos. Como se puede observar, se obtuvo un mejor resultado para tamaño de batch 16, por lo cual se utilizará este valor para entrenar el modelo final.

| | Kappa entrenamiento | Kappa validación |
|---|---|---|
| batch = 16 | 0.78 | 0.74 |
| batch = 32 | 0.76 | 0.71 |

Cuadro III: Comparación de desempeños para batch =16 y batch = 32

### *IV-H. Desempeño en Test del Mejor Modelo*

En esta última sección se hace una recapitulación del mejor modelo obtenido, el cual va a ser utilizado para evaluar el desempeño en test.

Respecto a los datos, se utiliza un preprocesamiento donde se escalan las imágenes para que el radio del ojo sea de 300 píxeles, y luego se recortan las imágenes en un tamaño de $600 \times 600$. Además, se iguala la iluminación restándole a cada canal de la imagen el mismo canal pasado por un filtro gaussiano de varianza $\sigma^2 = 60$.

Respecto al modelo, se utiliza transfer learning, re-entrenando un modelo ResNet50 pre-entrenado con el database de imagenet. Además, se agrega una capa de aumento de datos, donde las transformaciones aleatorias consisten en modificar el contraste e iluminación de forma aleatoria, además de borroneado gaussiano y flips horizontales y verticales.

Finalmente, respecto al entrenamiento, se entrena únicamente la última capa por 2 épocas, con un learning rate de 5e-4. Luego se entrena desde la capa 54 en adelante por 10 épocas con un paso de 2e-5.

Al ir a evaluar este modelo en test, nos encontramos con el problema práctico de que debemos aplicar el preprocesamiento a todos los datos de test. Esto implica replicar lo que hicimos en train, pero para casi el doble de imágenes. Esto nos llevo un día y medio de computo para train, por lo que para test esto llevaría al menos dos días. Además, ya estábamos teniendo problemas con la asignación de GPU del Colab. Por lo tanto, se decidió entrenar un modelo solo con la primera parte del preprocesamiento, la del escalado del radio del ojo a 300 píxeles, descartando la parte de modificar el contraste e iluminación. Esto es porque esta segunda parte es por lejos lo más costoso computacionalmente. Ahora sí, se procede a entrenar este modelo con toda la data de train (en este caso no se separa un conjunto para validación) y evaluar en test. El resultado obtenido es de un Cohen Kappa de 0.76147, el cual se puede observar en la competencia interna de Kaggle.

Por úlitmo, se deja un link a una carpeta en Drive [2], en la cual se puede levantar este último modelo, evaluar en conjuntos de entrenamiento y validación, así como generar una submission con los datos de test para subir a Kaggle.

## V. Conclusiones

En este proyecto se logra desarrollar una solución para el problema de punta a punta. Para esto se exploran diferentes técnicas estudiadas en el curso observando el efecto de cada una en el desempeño del modelo. En primer lugar, se aprende sobre el manejo de grandes cantidades de datos. Se familiariza con el uso de TFrecords y de Dataset, mostrando la importancia del uso de estos para poder traer a memoria y procesar de a batches, pues sino la cantidad de imágenes sobrepasa la cantidad de cómputo disponible.

Con respecto al modelo, la principal estrategia utilizada que se mantiene durante todo el proyecto es la de transfer learning. Esta estrategia demostró ser muy útil a la hora de trabajar con modelos complejos, y trabajar con grandes cantidades de datos. Los modelos entrenados de esta forma se utilizaron como modelos básicos para luego mejorarlos. Las estrategias para mejorar este son las vistas en el curso, especialmente en la parte de redes convolucionales.

[2]Link a carpeta de Drive con modelo y código:https://drive.google.com/drive/folders/1wOUpb_mUL939rg-3txIlCEv31q3Hgh_I?usp=share_link

La que opinamos que más importancia tuvo fue el aumentado de datos. Este permitió no solo compensar el sobreajuste, pero además mejoro el desempeño del modelo. Por otro lado, lo que más trabajo nos llevó fue el preprocesado de datos. Este demostró ser beneficioso, sin embargo dada la cantidad de tiempo que llevó, comparado por ejemplo con el aumento de datos que es simplemente agregar 4 líneas de código, la mejora proporcionada no es tan determinante. Esto nos muestra lo impresionante del aprendizaje profundo. Las estrategias clásicas de procesado de datos se basan en una ingeniería pensada y particular para el problema a resolver, generando tareas como las hechas en el preprocesamiento (escalar todos los ojos a igual radio, o igualar iluminación), sin embargo el aprendizaje profundo escapa de estos, logrando aprender a pesar de que no se tenga este preprocesado tan complejo, siempre que se tenga una gran cantidad de datos.

Como cierre, creemos que este proyecto mantuvo un comportamiento bastante lineal, empezando por un modelo sencillo y agregando capas de complejidad una a una, las cuales permitieron terminar con un modelo de muy buen desempeño. Esto nos permitió familiarizarnos con estrategias muy útiles en el ámbito de aprendizaje automático, y en particular, aprendizaje profundo.

## REFERENCIAS


[1] https://www.kaggle.com/code/ratthachat/aptos-eye-preprocessing-in-diabetic-retinopathy
[2] https://keras.io/examples/vision/grad_cam/