Informe de Práctica 2: Segmentación OCT

Correo	Nombre
i.moure@udc.es	Iván Moure Pérez
pablo.hernandez.martinez@udc.es	Pablo Hernández Martínez

Introducción

blablablablablablabla

Desarrollo

En general, intentamos encapsular la mayoría de tareas en funciones independientes para no repetir los mismos bloques de código varias veces y facilitar el proceso de *debug*.

Pasos previos

Para empezar a trabajar, nos basamos en el *notebook* provisto, que ya contiene clases para definir el dataset y la arquitectura de la red *UNet*. A continuación preparamos el *dataset* para el entrenamiento, asignando 40 imágenes al conjunto de entrenamiento, 5 al de validación y 5 al de test. Las imágenes se reescalan a una resolución de 416x624, y se crean los *dataloaders* con *batch size* de 8.

Entrenamiento

Para entrenar los modelos creamos una función que recibe todos los parámetros necesarios (*dataloaders*, número de *epochs*, optimizador, etc.) y devuelve el modelo entrenado. Además, también muestra una gráfica con la evolución del *loss* de entrenamiento y validación. El modelo se guarda a disco, y nos quedamos con la versión con mejor *loss* de validación.

Evaluación

Como la red *UNet* no tiene una capa sigmoide al final, utilizamos una función `get_binary_mask` para obtener la clase de salida. Esta función recibe las salidas (*logits*) del modelo, les aplica la sigmoide y las binariza según el umbral deseado.

Una vez entrenado el modelo, lo evalúamos con varios umbrales para decidir cual es el más adecuado. Se calculan las métricas *accuracy*, *precision*, *recall*, *IoU/Jaccard* y *F1/Dice*. Las métricas resultantes se muestran en formato tabla y gráfica. Esto nos permite, por ejemplo, elegir el umbral que equilibre mejor el *recall* y el *precision*. A mayores, también mostramos la curva *Precision-Recall* y el *average precision* para comparar el rendimiento de los modelos en cuanto a estas métricas.

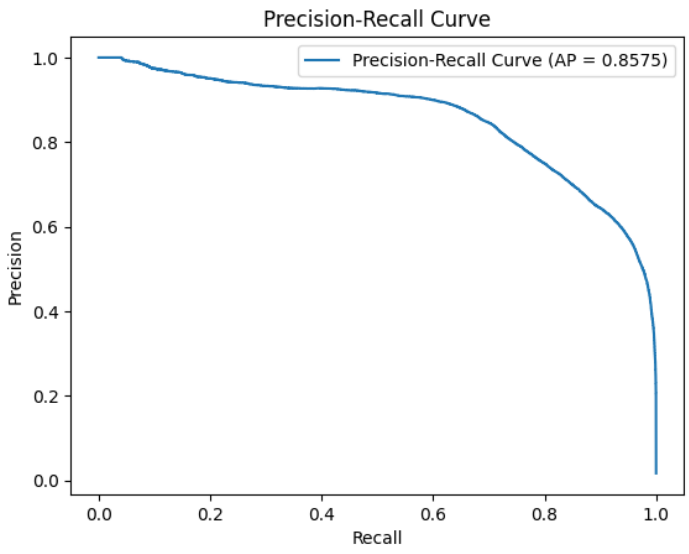

Figura 1: Ejemplo de curva *Precision-Recall*

Por último, creamos una función que muestra por pantalla las predicciones (máscaras) de un modelo, para evaluar su rendimiento visualmente. Creemos que es importante mostrar el comportamiento real del modelo

porque sería lo que verían los profesionales médicos al utilizar el sistema, y podríamos obtener *feedback* real sobre su precisión y fiabilidad.

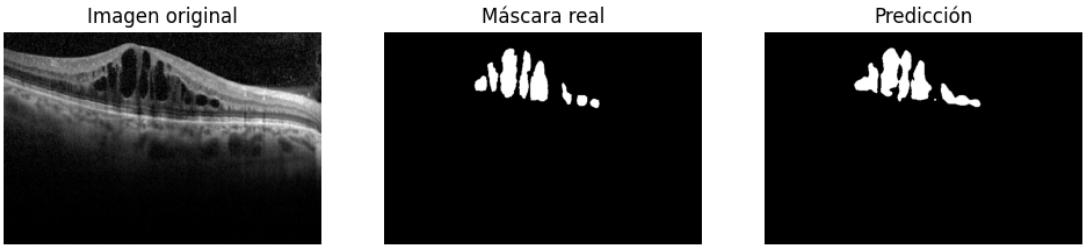

Figura 2: Ejemplo de predicción de uno de los modelos

Mejoras implementadas

Ponderación de la clase positiva

El primer modelo que entrenamos predecía todos los píxeles como negros. Esto no es sorprendente: el desbalanceo entre clase negativa (negro) y positiva (blanco) es considerable, y el modelo aprendió que predecir siempre 0 minimizaba la función de *loss*.

En vez de cambiar a otro tipo de modelo que fuese más resiliente frente al desbalanceo como un árbol de decisión, modificamos los parámetros de la función de loss para darle 40 veces más peso a la clase positiva. Probamos con valores más bajos (<=30) pero no eran suficientes para compensar el desbalanceo y converger en una solución satisfactoria. Con valores más altos (>=50) el modelo era demasiado propenso a predecir la clase positiva (a veces incluso predecía todo como positivo).

Esta mejora se aplica a todas las funciones de *loss* que incluyan *Binary Cross Entropy*. Somos conscientes de que tendríamos que tunear este hiperparámetro si adaptásemos el sistema para tratar con otro tipo de imágenes médicas que tuviesen una distribución de clases diferente.

Data Augmentation

Teniendo en cuenta que la cantidad de imágenes anotadas es escasa, probamos a utilizar *Data Augmentation* para introducir más variabilidad e intentar que los modelos generalizasen mejor. Las transformaciones por las que nos acabamos decantando fueron volteos horizontales (p=.1), rotaciones aleatorias (±5 grados) y variaciones de contraste (±10%). Inicialmente también empléabamos `RandomAffine`, pero comprobamos que era demasiado violenta y mermaba el rendimiento en la mayoría de ocasiones. A continuación, creamos un *dataloader* nuevo para poder pasar las imágenes transformadas a la función de entrenamiento.

Funciones de *loss*

Después de entrenar el *baseline* con *BCE*, se nos ocurrió incorporar las métricas pertinentes en una tarea de segmentación semántica (*Dice*, *F1*, *Hausdorff*) a la función de loss. Tras investigar un poco descubrimos que es posible, pero normalmente combinándolas con *BCE* para facilitar el flujo de gradiente y poder ponderar una de las clases (Azad, 2023). En estas funciones de *loss* combinadas, se añade un hiperparámetro *alpha* que nos permite ajustar la contribución de cada función al valor final:

> $\text{loss} = (\alpha)\text{loss}_1 + (1 - \alpha)\text{loss}_2$

Procedimos a crear subclases de `torch.nn.Module` para poder usar estas funciones de *loss* personalizadas en nuestro bucle de entrenamiento.

- *BCE + Dice*: para esta función, el *alpha* es 0.5 porque en nuestra experiencia los valores más bajos no convergen o tardan demasiado, y los valores más altos apenas mejoran el rendimiento del baseline
- *IoU*: para este loss no usamos *BCE* (*alpha*=0), porque no observamos ninguna dificultad con la convergencia durante el entrenamiento y además nos otorga el mejor rendimiento
- *BCE + Hausdorff*: aquí alpha es 0.999: puede parecer extremo, pero observamos que la *loss* de *Hausdorff* devuelve valores en una escala mucho mayor, así que es necesario para igualar la contribución y asegurar la convergencia

Decay de la tasa de aprendizaje

Durante el entrenamiento, nos dimos cuenta de que la tasa de aprendizaje inicial podía ser demasiado alta, ya que en ocasiones el modelo convergía en valores mucho más altos que en otros entrenamientos. Para evitar esto, añadimos un *decay* de la tasa de aprendizaje (`ReduceLROnPlateau` de `torch.optim`), que divide el *learning rate* entre 10 cada vez que el *loss* de validación pasa 7 epochs sin mejorar significativamente. Esto nos permitió estabilizar el entrenamiento y obtener resultados más consistentes. Al principio la paciencia era de 5 epochs, pero decidimos aumentarla a 7 porque era muy fácil que el modelo ralentizase demasiado el aprendizaje y no convergiese.

Hiperparámetros

Loss	**Epochs**	**LR inicial**	**¿LR Decay?**	**¿Data Augmentation?**	**Umbral**
BCE (baseline)	75	1e-4	No	No	0.95
BCE	100	1e-4	Si	Si	0.95
BCE + Dice	90	1e-4	Si	No	0.9
IoU	90	1e-4	Si	No	0.4
BCE + Hausdorff	120	1e-4	No	No	0.95

blablablablablablablablabla

Conclusiones

blablabla

Referencias

- Azad, R., Heidary, M., Yilmaz, K., Hüttemann, M., Karimijafarbigloo, S., Wu, Y., ... & Merhof, D. (2023). Loss functions in the era of semantic segmentation: A survey and outlook. arXiv preprint arXiv:2312.05391.