Universidad Europea

Memoria Práctica 4

Martín de las Heras

Introducción

En esta práctica hemos implementado el algoritmo de *Instance Segmentation* YOLOv1, de modo que obtenemos una experiencia práctica a cómo implementar este tipo de algoritmos y las modificaciones de los datos de entrada que ello conlleva. Todo esto se ha realizado bajo el objetivo de diferenciar en varias imágenes de microscopio de la sangre los distintos tipos de células presentes. La práctica se divide en los siguientes apartados:

- Leer el *dataset* y redimensionarlo para que sea ingestable por YOLO.
- Traducir las anotaciones de las imágenes a la notación utilizada por el algoritmo.
- Definir y crear la clase YOLOv1
- Definir y crear la función de pérdida del algoritmo.
- Entrenar y dibujar los resultados obtenidos.

Lectura del *dataset*

En primer lugar, iteramos por el *dataset* disponible en Canvas, transformando las imágenes en el formato adecuado para el algoritmo:

```
dd = PIL.Image.open(f'./BCCD_Dataset/BCCD/{image}')
tt = torchvision.transforms.functional.pil_to_tensor(dd)
tt = tt.to(dtype=torch.float)

xscaler = 448./tt.shape[2]
yscaler = 448./tt.shape[1]

tt = torchvision.transforms.functional.resize(tt, (448, 448))
tt = tt[None, :, : :]
```

Traducción de las anotaciones

De la misma manera, leemos el fichero de anotaciones de las imágenes, donde se nos indican las posiciones de las distintas células. Para ello necesitamos los atributos que es capaz de leer YOLO: sus coordenadas del vértice superior izquierdo (x e y) su altura y su anchura:

```
rows = annotations.loc[annotations['filename'] == image]
rows = rows.reset_index()
bounding_boxes.append([{'cell_type':rows['cell_type'][cellid],
                        'xmin': int(rows['xmin'][cellid] * xscaler),
                        'xmax': int(rows['xmax'][cellid] * xscaler),
                        'ymin': int(rows['ymin'][cellid] * yscaler),
                        'ymax': int(rows['ymax'][cellid] * yscaler)} for cellid in range(len(rows))])
```

Crear YOLOv1

A continuación, creamos la clase de YOLOv1, la cual contiene las capas especificadas en el artículo original:

```
class YOLOv1(nn.Module):
    def __init__(self):
        super().__init__()
        self.depth = config.C + config.B*5
        layers = [
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.LeakyReLU(0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.LeakyReLU(0.1),
            nn.MaxPool2d(kernel_size=2, stride=2),
```

El código completo es mucho más largo, pero por brevedad se pone solo este fragmento.

Crear la función de pérdida

En este apartado definimos la parte más compleja del algoritmo, su función de pérdida. Para ello nos valemos de funciones auxiliares para calcular el índice de Jaccard (también conocido como IoU) y otra para calcular el error cuadrático medio o MSE:

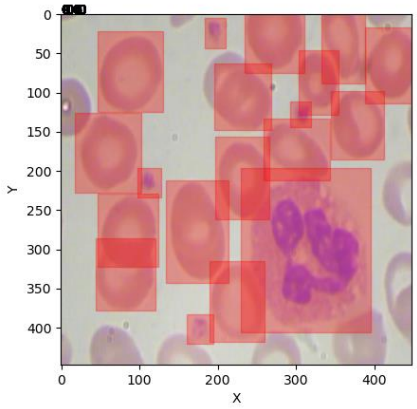
```
def mse_loss(a, b):
    flattened_a = torch.flatten(a, end_dim=-2)
    flattened_b = torch.flatten(b, end_dim=-2).expand_as(flattened_a)
    return F.mse_loss(
        flattened_a,
        flattened_b,
        reduction='sum'
    )


def get_iou(p, a):
    p_tl, p_br = bbox_to_coords(p)
    a_tl, a_br = bbox_to_coords(a)
```

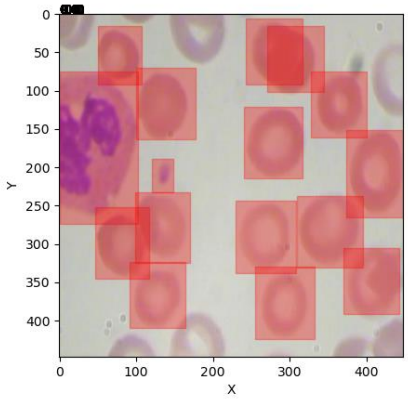
Al igual que antes, de la función de IoU se muestra solo un fragmento del código implementado por su longitud.

Lo siguiente es, haciendo uso de las funciones auxiliares, implementar la función de pérdida según viene definida en el artículo original.

Entrenamiento y predicciones

A esto prosigue un entrenamiento como los anteriormente vistos en la asignatura. A continuación, como muestra para la memoria de la práctica, se imprimen las predicciones realizadas sobre las imágenes determinadas de manera que podamos apreciar el nivel de precisión del algoritmo. Para ello hemos definido una función auxiliar, la cual convierte las coordenadas y medidas generadas por YOLO a medidas representables en la imagen. Con esto podemos observar resultados como los siguientes:





Podemos ver claramente cómo es capaz de detectar gran cantidad de células, si bien tiene mucho margen de mejora (se trata de la versión 1 del algoritmo, y actualmente van por la 8).

Conclusiones

En esta práctica hemos visto cómo implementar el algoritmo de segmentación de instancias YOLOv1, aplicado al caso de uso de conteo de células sanguíneas. De la misma manera, hemos aprendido cómo varía la manera de representar las *bounding boxes* en un algoritmo de este tipo y cómo representarlas. De esta manera, podríamos adaptar de manera sencilla otras iteraciones de este mismo algoritmo de cara a problemas futuros, así como aplicarlo a infinidad de casos. Esto nos ha aportado una experiencia práctica en cómo se trabaja con YOLO, donde la arquitectura de la red condiciona los datos que se pueden procesar, así como un caso de uso en el que las técnicas que hemos visto en clase producen resultados tangibles y de posible aplicación inmediata.