Semana 2: Entrenamiento, validación y evaluación de redes neuronales

1 Introducción

En esta segunda semana, se proporcionarán las herramientas necesarias para comprender el proceso de entrenamiento de redes neuronales o "modelos" en el ámbito de la visión artificial, específicamente para la tarea de detección de RoboMasters. Para ello, se utilizará **PyTorch**, una de las bibliotecas más importantes de Python en el campo del aprendizaje profundo.

La **base de datos** que los alumnos anotaron la semana anterior servirá como punto de partida para esta segunda semana. La anotación de datos es un paso fundamental en el entrenamiento de modelos de detección de objetos, ya que proporciona información sobre la ubicación y la clase de los objetos de interés en las imágenes. Además, el **análisis exploratorio** realizado durante la primera semana ayudará a validar y explicar los resultados obtenidos tras el entrenamiento y la evaluación de los modelos.

Durante esta semana, los estudiantes aprenderán a cargar la base de datos anotada, preparar los datos para el entrenamiento, manejar arquitecturas de redes neuronales, y entrenar, validar y evaluar los modelos. Para facilitar la tarea, partiremos de modelos potentes preentrenados para otras tareas de detección. Con esto se consigue aprovechar el conocimiento aprendido previamente por la red para acelerar el proceso de entrenamiento y mejorar la precisión de las detecciones.

2 JupyterHub

Utilizaremos el JupyterHub montado en el laboratorio como framework para entrenar nuestras redes. Para acceder a él, abra un navegador y escriba la siguiente dirección: `https://k8ssr.labdoc.ssr.upm.es:30000`.

Asegúrese de entrar al perfil de JupyterHub con acceso a GPU. Si hemos seguido bien los pasos de la semana pasada, nuestro perfil debería tener la siguiente estructura de directorios:

```
claim-grupoX
├─ dataset
│  ├─ Secuencia 1
│  │  ├─ images
│  │  └─ labels
│  ├─ Secuencia 2
│  │  ├─ images
│  │  └─ labels
│  └─ ...
└─ notebooks
   ├─ week1
   └─ week2
      ├─ test.ipynb
      └─ train.ipynb
```

```python
def torchvision_model(model_name: str, pretrained: bool = False, num_classes: int = 2) -> Any:
    """
    Return a model from a list of Torchvision models.
    :param model_name: name of the Torchvision model that you want to load.
    :param pretrained: whether pretrained weights are going to be loaded or not.
    :param num_classes: number of classes. Minimum is 2: 0 = background, 1 = object.
    :return:
        model: Torchvision model.
    """

    # Torchvision models
    model_dict = {
        'faster_rcnn_v1': detection.fasterrcnn_resnet50_fpn,
        'faster_rcnn_v2': detection.fasterrcnn_resnet50_fpn_v2,
        'faster_rcnn_v3': detection.fasterrcnn_mobilenet_v3_large_fpn,
        # 'faster_rcnn_v4': detection.fasterrcnn_mobilenet_v3_large_320_fpn,
        # 'fcos_v1': detection.fcos_resnet50_fpn,
        'retinanet_v1': detection.retinanet_resnet50_fpn,
        'retinanet_v2': detection.retinanet_resnet50_fpn_v2,
        'ssd_v1': detection.ssd300_vgg16,
        'ssd_v2': detection.ssdlite320_mobilenet_v3_large,
    }
```

Figure 1: Torchvision models.

3 PyTorch

3.1 Modelos

Diríjase al notebook *notebooks/week2/train.ipynb*, y ejecute las primeras celdas para importar las librerías necesarias. Después, en la sección *Utils functions* podrá encontrar varias funciones que usaremos para entrenar nuestra red. No es necesario (aunque sí recomendable) entender todas las funciones. Nos centraremos en la función *torchvision_model* (Figura 1). Dicha función contiene un diccionario de modelos de Torchvision (una sub-librería de PyTorch con modelos pre-entrenados). Utilice las claves de los modelos listados cuando vaya a entrenar para indicar qué red desea entrenar. Note que las claves son solo nombres elegidos para facilitar la tarea de llamar a los distintos modelos, el nombre real de cada modelo es el que aparece en los valores del diccionario (No queremos ver en la memoria *faster_rcnn_v1*, sino *Faster R-CNN con un backbone ResNet50-FPN*).

No se espera que los alumnos sean capaces de comprender cómo funcionan estos modelos por detrás de forma exacta, ya que contienen módulos y estructuras muy complejas. Sin embargo, se valorará positivamente a aquellos grupos de estudiantes capaces de investigar y explicar cómo funcionan y qué características tienen. Ignore además los 2 modelos que aparecen comentados en la Figura 1, ya que no han sido probados para esta asignatura y es complicado conseguir un buen funcionamiento con ellos (se valorará positivamente a los grupos que consigan un buen rendimiento con ellos).

3.2 Generador de datos

Los generadores de datos se emplean para alimentar a la red neuronal. Cuando tenemos bases de datos datos de cientos, miles o incluso millones de imágenes, es inviable que cualquier ordenador sea capaz de retener en memoria todas esas muestras y procesarlas simultáneamente a través de una red neuronal. Como alternativa, se emplean generadores de datos para extraer lo que se conoce como *batches* de imágenes (con sus respectivas anotaciones o "ground-truth"), que no es más que un subconjunto de imágenes anotadas. Estos batches van alimentando a la red de forma iterativa

```python
class CustomDataset(Dataset):

    def __init__(self, path_dataset: str, resize_shape: tuple = None, transform: transforms.Compose = None) -> None:
        """
        Custom dataset that feeds the network during train, validation, and test.
        :param path_dataset: path to the dataset.
        :param resize_shape: tuple indicating height and width to resize images (for faster performance).
        :param transform: list of transforms to apply to the images.
        """
        print("Loading annotations...")
        self.annotations = parse_annotations(path_dataset)
        self.resize_shape = resize_shape
        self.transform = transform

    def __getitem__(self, idx: int) -> Tuple[torch.Tensor, dict]:
        """
        Get an index corresponding to one of the images and return the image and its annotation.
        :param idx: index of image to load.
        :return:
            image: Torch tensor containing the image with shape (Channels, Height, Width).
            targets: dictionary with the bounding boxes (boxes) and class labels (labels) of each annotated object.
        """
        # Get one annotation for the current index
        annotation = self.annotations[idx]

        # Load image
        image = Image.open(annotation['path_image']).convert("RGB")

        # Load bounding boxes
        boxes = self.annotations[idx]['boxes']
        boxes = torch.Tensor(boxes)

        # Load labels (class of the object)
        labels = self.annotations[idx]['labels']
        labels = torch.Tensor(labels).type(torch.int64)  # Specify that labels are int

        # Apply transforms
        if self.transform:
            w, h = image.size
            image, boxes, class_labels = self.transform(image, boxes, labels)
            if self.resize_shape:
                boxes = resize_boxes(boxes, self.resize_shape, (h, w))

        # Torchvision models use this structure for boxes and labels
        targets = {'boxes': boxes, 'labels': torch.Tensor(labels)}
        return image, targets
```

Figure 2: Generador de datos.

hasta que se hayan procesado todas las imágenes de la base de datos. En ese momento se dice que la red ha entrenado durante un *epoch*. Normalmente, conseguir un buen aprendizaje requiere de decenas o centenas de epochs, dependiendo de la red neuronal y la base de datos empleadas.

Se recomienda a los alumnos entender el funcionamiento del generador de datos de PyTorch. Para ello, se recomienda empezar por la clase *CustomDataset* (ver Figura 2) y sus respectivas funciones. Analice el formato de los datos a la salida de la función *__getitem__*, ya que debe coincidir con el formato de los datos a la entrada de la red neuronal. Preste atención también al método *self.transform* y explique qué preprocesado se aplica a las imágenes y por qué.

3.3 Hiperparámetros

La Figura 3 muestra los hiperparámetros que utilizaremos para entrenar la red neuronal elegida. Los hemos clasificado en tres tipos:

- **Hiperparámetros de la base de datos:** Incluyen información relativa a la carga de la base de datos y la división en subconjuntos de entrenamiento, validación y evaluación.

```
local_path = '../..'

# Seed (for reproducibility) -> Do NOT change
seed = 0
set_seed(seed)

# Data parameters
path_dataset = f'{local_path}/dataset'                    # Path to dataset
train_val_test_split = [0.7, 0.1, 0.2]                    # Train-Val-Test split of the dataset

# Model parameters
model_name = 'faster_rcnn_v1'                             # Torchvision model
model_path = f'{local_path}/models/{model_name}/model.pt' # Path to save trained model
num_classes = 2                                           # Number of classes (ALWAYS 2: 0=background, 1=robot)
resize_shape = None                                       # Resize images for faster performance. None to avoid resizing
pretrained = True                                         # Use weights pre-trained on COCO dataset

# Train parameters
use_gpu = True                                            # Use GPU (True) or CPU (False)
num_workers = 0                                           # Number of workers (CPU cores loading data)

# Other train parameters (explain them and play with them)
num_epochs = ...
batch_size = ...
lr = ...
lr_momentum = ...
lr_decay = ...
lr_factor = ...
lr_patience = ...
lr_threshold = ...
lr_min = ...
models_to_clip = ['<model-1>', '<model-2>']
max_grad_norm = 1.0 if model_name in models_to_clip else 0
```

Figure 3: Hiperparámetros.

Se debe utilizar la base de datos nombrada como *dataset* en la sección 2. El parámetro *train_valid_test_split* permitirá dividir las imágenes del dataset en conjuntos de entrenamiento (80%) validación (20%) y evaluación (0%). Opcionalmente, juegue con los 2 primeros valores para comprobar cómo varía el rendimiento del modelo. Deje el último a 0 porque la evaluación se hará con otro dataset (ver sección 3.5).

- **Hiperparámetros del modelo:** Los parámetros relativos al modelo a entrenar. Note que la variable *model_path* no solo indica dónde se debe guardar el modelo entrenado. También permite retomar entrenamientos.

> **IMORTANTE**: Suponga que, por cualquier motivo, su notebook falla mientrás está entrenando un modelo. Para evitar tener que entrenar desde cero, puede retomar el entrenamiento por el mismo punto en el que lo dejó simplemente cargando el archivo que indique la variable *model_path*. Esto **SOBRESCRIBIRÁ** el archivo .pt indicado. Téngalo siempre en cuenta antes de empezar a entrenar para evitar sobreescribir modelos por error. Se recomienda encarecidamente poner un nombre único a cada modelo (a cada archivo .pt) antes de comenzar cada entrenamiento.

- **Hiperparámetros de entrenamiento:** valores que debe comprender y con los que puede jugar para comprobar como varía el entrenamiento. Deje fijo el uso de GPU (*use_gpu*) y el número de CPU cores (*num_workers*) e intente rellenar el resto de valores. Vea la sección 4 para entender bien lo que se espera que haga.

```python
# Training loop
best_val, hist = np.inf, {'train_loss': [], 'val_loss': []}
for epoch in range(initial_epoch, num_epochs):

    # Set model in train mode (calculate gradients)
    model.train()

    # Iterate over each batch
    loss_avg = []
    print(f"Epoch {epoch + 1}/{num_epochs}")
    for images, targets in tqdm(train_loader, desc='Train'):

        # Move data to device
        images = torch.stack(images, dim=0).to(device)
        targets = [{k: v.to(device) for k, v in targets[t].items()} for t in range(len(images))]

        # Predict and get loss value
        loss_dict = model(images, targets)
        losses = sum(loss for loss in loss_dict.values()).mean()
        loss_avg.append(losses)

        # Reset gradients
        ...

        # Apply backpropagation
        ...

        # Clip gradients (to avoid gradient exploding)
        clip_grad_norms(optimizer.param_groups, max_norm=max_grad_norm)

        # Update model's weights
        ...

    # Update scheduler
    lr_scheduler.step(losses)

    # Validation (gradients are not necessary)
```

Figure 4: Bucle de entrenamiento.

3.4 Entrenamiento

Las tres primeras celdas de la sección "Train" crearán los generadores de datos de entrenamiento y validación, el modelo, el optimizador y el "learning rate scheduler" (explique qué es y para qué se usa). La cuarta celda (ver Figura 4) muestra el bucle de entrenamiento. Normalmente, en PyTorch, por cada epoch se debe:

1. Fijar el modelo en modo de entrenamiento para habilitar el cálculo de gradientes.
2. Crear un bucle interno para extraer batches del generador de datos de entrenamiento.
3. Mover los datos a la GPU o CPU (generalmente, usaremos GPU).
4. Realizar la predicción y calcular la función de coste. Los modelos de Torchvision en modo de entrenamiento devuelven el valor de la función de coste directamente sin tener que calcularlo a mano con otra función adicional.
5. Resetear los gradientes guardados en el optimizador.
6. Aplicar "backpropagation" para calcular los gradientes.
7. Limitar la norma de los gradientes (solo para alguna red neuronal concreta).

8. Permitir al optimizador actualizar los pesos de la red a partir de los gradientes calculados previamente.

9. Repetir los pasos 3-8 hasta procesar todas las imágenes de la base de datos.

10. Actualizar el "learning rate scheduler".

11. Realizar la validación y guardar el modelo (nosotros guardaremos el modelo solo si mejora la función de coste de validación, pero se pueden explorar otras estrategias).

Por último, ejecute la última celda para obtener las curvas de entrenamiento y validación. Explíque su utilidad y qué significado tiene su forma (creciente, decreciente, ruidosa, etc.). Explíque también todos los pasos para entrenar las redes neuronales. Preste especial atención a los pasos 5, 6 y 8, ya que deben ser completados por los alumnos.

IMORTANTE: Juegue con el notebook *train.ipynb* hasta que tenga algo funcional. Cuando lo consiga, puede ejecutar todas las celdas y dejar el navegador abierto mientras el notebook completa el entrenamiento. Note que los entrenamientos pueden llevarle cierto tiempo (aproximadamente 1 hora) y que el proceso es algo inestable, ya que si cierra el navegador o se le cae la conexión de red, el entrenamiento se detendrá (no es trágico ya que guardamos el modelo en cada epoch y puede retomarlo por dónde iba como se explica en la sección 3.3).

Para facilitar la tarea (no es obligatorio, pero si recomendable para ahorrar tiempo), pulse Ctrl+Shift+L, abra un Terminal y escriba el siguiente comando (ponga el nombre que quiera al nuevo notebook, preferiblemente algo que explique la prueba que está usted realizando):

papermill train.ipynb {new-notebook-name}.ipynb

Esto lanzará de fondo el entrenamiento. Abra regularmente el archivo generado y compruebe desde ahí por cuál epoch va y cuánto le queda para terminar. **NO CIERRE** el Terminal. Con papermill puede cerrar el navegador y desconectarse de JupyterHub si quiere, pero no cierre la pestaña específica del Terminal dentro de JupyterHub. Si quiere detener el entrenamiento, pulse Ctrl+C y luego cierre el Terminal.

Opcionalmente, revise la documentación de papermill para realizar varias pruebas de un solo golpe. En especial, échele un vistazo a este y a este enlace.

3.5 Evaluación

Para evaluar el modelo entrenado en el subconjunto de evaluación (test) vamos a utilizar el notebook *test.ipynb*. Este notebook es similar al anterior. Ejecútelo entero indicando el modelo entrenado y el dataset *dataset_test*. Este dataset constará de una serie de secuencias comunes para todos los grupos de alumnos y distintas a las secuencias de entrenamiento y validación. Tras ejecutar el notebook entero, se obtendrá un conjunto de métricas de evaluación cuantitativas. Investigue y explíque cómo funcionan las métricas mostradas y qué miden. Utilícelas para comparar diferentes modelos y determinar cuál será su modelo preferido.

Adicionalmente, las celdas finales del notebook proveeran de herramientas para mostrar las predicciones. Haga un análisis cualitativo de las predicciones, mostrando casos en los que el modelo funcione bien y casos en los que falle. Preste especial atención al parámetro *score_threshold* de la función *plot_predictions*. Determine la utilidad de este parámetro. Opcionalmente, puede hacer un análisis para indicar el valor ideal para este parámetro.

4 Objetivos

Con la base de datos de la semana pasada, debe conseguir entrenar un modelo robusto de detección de RoboMasters y presentar un informe con las observaciones y conclusiones extraídas durante el proceso. Los pasos a seguir son los siguientes:

- Complete el código del bucle de entrenamiento y explique bien todos los pasos.
- Elija un primer modelo para entrenar (recomendación: "faster_rcnn_v1").
- Juegue con los hiperparámetros hasta dar con un buen modelo. Investigue cuál puede ser una buena selección inicial de parámetros (pistas por aquí y por aquí) y céntrese en tunear (a base de prueba y error), principalmente, los siguientes:
  - **Número de epochs**: ¿Hasta cuándo debe entrenar su modelo? No malgaste su tiempo entrenando hasta el infinito.
  - **Batch size**: ¿Qué ocurre con valores muy bajos? ¿Y con valores muy altos? ¿Aparece algún error de ejecución? Documéntelo todo.
  - **Learning rate**: ¿Qué ocurre con valores muy bajos? ¿Y con valores muy altos? ¿Afecta en algo al número de epochs que debo utilizar? ¿Qué hace el "learning rate scheduler"? Pista: Intente imprimir por pantalla (o a hacer una gráfica) el learning rate actualizado por el learning rate scheduler en cada epoch. Le puede dar información adicional.
- Cuando considere que tiene los mejores hiperparámetros, déjelos fijos y pruebe a comparar diferentes redes neuronales. Elija la que mejor funcione.
  - Si alguna red le da problemas, preste atención a los parámetros "models_to_clip" y "max_grad_norm". Intente explicar qué sucede y para qué sirven tanto estos parámetros como la función "clip_grad_norms" utilizada en el bucle de entrenamiento.
- Analice y explique las curvas de entrenamiento y las métricas de evaluación. Investigue cómo se calcula el "mean Average Precision" (mAP) y sus variantes (mAP@0.5, mAP@Small, ...). Utilice estas métricas para comparar sus modelos entrenados.
- Decida en qué casos debe utilizar "data augmentation" para entrenar y/o evaluar sus modelos. Analice qué ventajas y desventajas puede tener su uso. ¿Mejora el rendimiento si utiliza "data augmentation"? (Se recomienda dejar esta prueba para el final).
- Analice cualitativamente el modelo: casos en los que acierte y en los que falle, y por qué.
- (Opcional) Pruebe, no solo a evaluar la precisión de los modelos (con el mAP), si no también a medir los tiempos de inferencia (tiempo de ejecución del modelo en test). Esto le permitirá comparar la velocidad de computación de cada modelo.
- (Opcional) Para entrenar, se dividen aleatoriamente todas las imágenes del dataset en 80% para entrenar y 20% para validar (parámetro *train_valid_test_split*). Lo más apropiado sería designar unas secuencias de entrenamiento y otras de validación. Haga los cambios pertinentes en el generador de datos para poder separar secuencias de entrenamiento y validación.
- Presentación de un informe con las observaciones y conclusiones extraídas. Este informe será incremental (se irá expandiendo a lo largo del proyecto). Se debe inluir una introducción al problema a resolver y un análisis del resto de puntos de esta sección. Se valorará la limpieza y presentación del documento, así como el hacer pruebas con sentido y fundamentadas.