Few-shot object detection
Implementation part

Ines BELHAJ MESSAOUD
Jesser BEN HOURIA

National Engineering School of Tunis

19 January 2022

Plan

1. Data preparation and visualization
2. Implementing YOLOv4
   - Why choosing YOLOv4
3. Few-shot implementation
   - MAML

Preparation of our dataset

- The YOLO implementation is applied on a grapes-dataset that contains images of 5 different classes.
- The classes represent 5 grape varieties which are the following

Prefix	Variety
CDY	*Chardonnay*
CFR	*Cabernet Franc*
CSV	*Cabernet Sauvignon*
SVB	*Sauvignon Blanc*
SYH	*Syrah*

Figure 1: The 5 classes of our dataset

Data preparation

Figure 2: CDY

Figure 3: CSV

Data preparation

Figure 4: CFR

Figure 5: SVB

Data preparation

Initially we had 3 file types in our database :

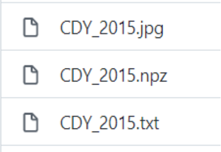

Figure 6: File types

- jpg files representing images of grapes.
- txt files representing the bouding boxes coordinates (class - center-x - center-y - width - height).
- npz files representing a format by numpy that provides storage of array data using gzip compression.

Data preparation

We arranged the dataset into two main folders images and labels :

- Images folder contains the .jpg files
- Labels folder contains the .txt files

Figure 7: Data Structure

Data preparation

- It is important to know what each text file represents

```
0 0.1116 0.3026 0.0845 0.2271
0 0.5168 0.5084 0.0415 0.0645
0 0.6697 0.4912 0.0386 0.0945
0 0.0779 0.4835 0.1040 0.1289
0 0.0588 0.5813 0.0493 0.0696
0 0.0193 0.6216 0.0386 0.1341
0 0.1721 0.6066 0.0405 0.0938
0 0.1572 0.4187 0.0381 0.0901
0 0.1714 0.3502 0.0498 0.0454
0 0.2073 0.2916 0.0825 0.0718
0 0.4072 0.5806 0.0664 0.1502
```

Figure 8: Example of a CDY image text file

Data visualization

We have observed the number of images in each category (class) :

```
[ ] n_vimages = {v: len(inst_v) for v, inst_v in instances.items()}
    n_vimages

    {'CDY': 65, 'CFR': 65, 'CSV': 57, 'SVB': 65, 'SYH': 48}
```

Figure 9: Number of images per class

Data visualization

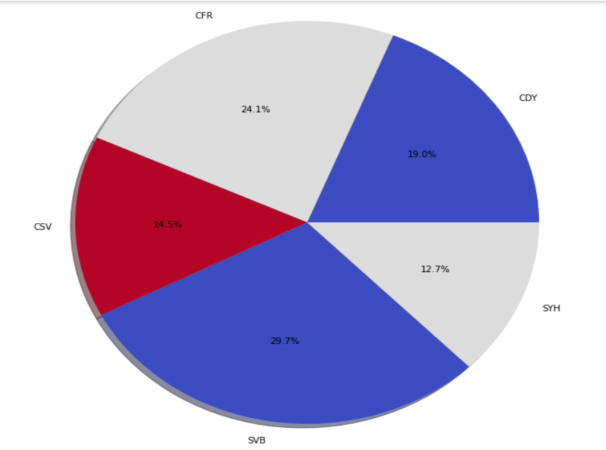

Figure 10: Image distribution per class by percentage

Data visualization

We have also observed the number of bounding boxes in each category (class) :

```
n_vboxes = {v: np.array([n for ii, n in n_iboxes[v].items()]).sum() for v in varietals}
n_vboxes
```

```
{'CDY': 840, 'CFR': 1069, 'CSV': 643, 'SVB': 1316, 'SYH': 563}
```

Figure 11: Number of bounding boxes per class

Data visualization

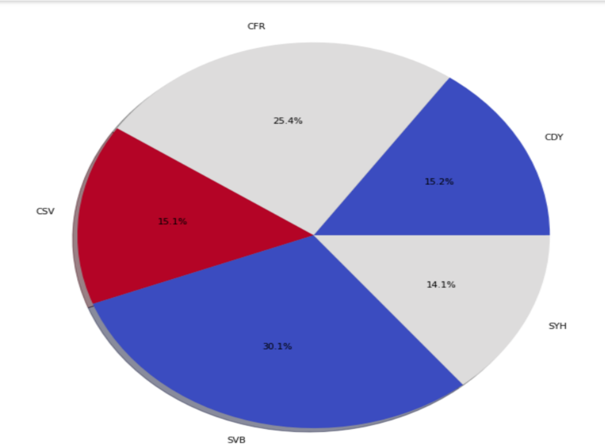

Figure 12: Boxes distribution per class by percentage

YOLOv4

We chose to implement the 4th YOLO version using both Pytorch and Tensorflow.

Why choosing Pytorch for the training

- In PyTorch, tools and codes are way more imperative and dynamic.
- The framework is more tightly integrated with Python language.

Why choosing Tensorflow for the visualization

- Tensorboard is awesome when it comes to visualization.
- Useful for debugging and comparison of different training runs.
- Visualizes the differences between runs

Why choosing YOLOv4

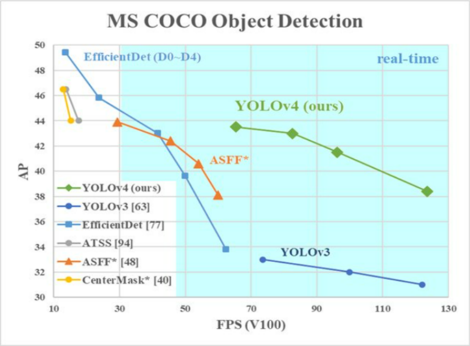

Figure 13: YoloV4 and YoloV3

Results of y

- Yolov4 is an improvement on the Yolov3 algorithm. The mean average precision(mAP) improved by 10% and the number of frames per second improved by 12%.
- The Yolov4 architecture has 4 different blocks : The backbone, the neck, the dense prediction, and the sparse prediction.

Evaluation Metrics

- We focused on evaluationg our yolov4 model with 3 main metrics :
- mAP : AP (Average precision) is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. Average precision computes the average precision value for recall value over 0 to 1.
- Precision
- recall

YoloV3 results

- With YoloV3, with 300 epochs, we have obtained a **precision** $= 0.67$ and a **recall** $= 0.47$.
- The average **IoU** is equal to 47.8.
- With 300 epochs, we have obtained a mAp $= 59.39$ %

Results

- All the training results and the test results are downloaded in the inference folder.
- The tensorboard also provides all the train and test metrics obtained with every Epoch.

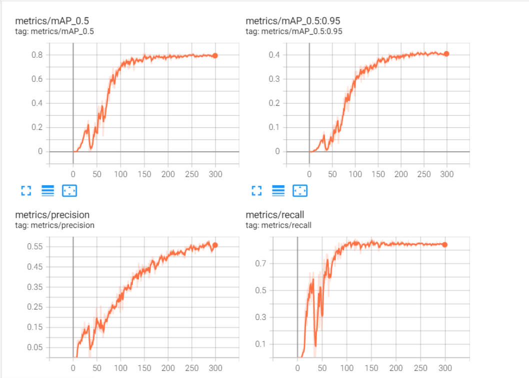

Figure 14: Result example with 300 epochs

Results

YOLO loss function is broken into three parts:

- The one responsible for finding the bounding-box coordinates (cls-loss : cross entropy loss)
- The bounding-box score prediction (giou-loss)
- The class-score prediction (obj-loss) : Mean squared error loss

Train Results

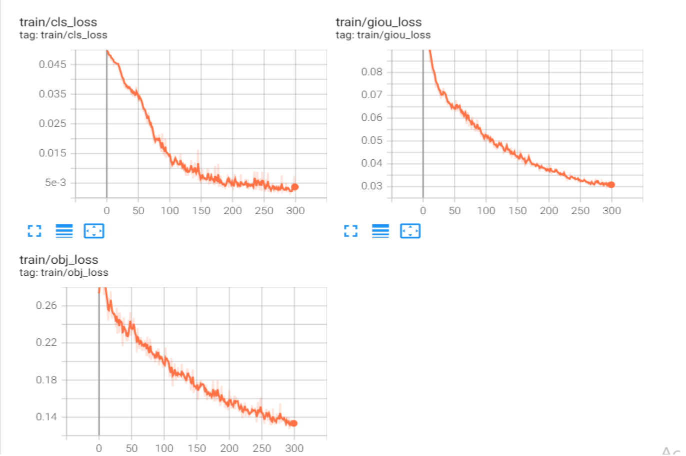

Figure 15: Train Result example with 300 epochs

Test Results

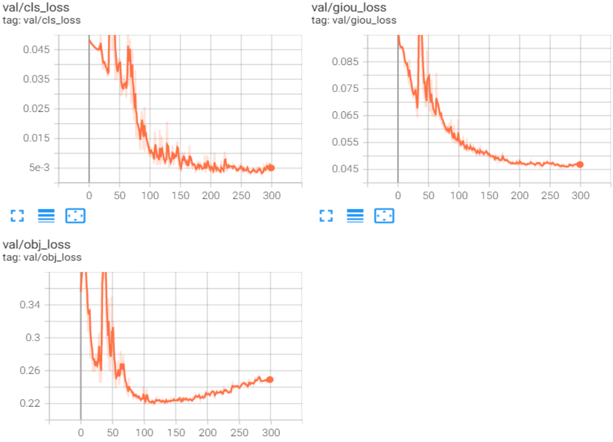

Figure 16: Test Result example with 300 epochs

Interpretation

- With 100 epochs we have noticed an underfitting : the map of the training set was remarkably low
- When we increased the number of epochs to 300 we obtained a map of 0.8.

FEW-SHOT implementation

First step of Few-shot implementation

- For the Few-shot training part, we chose to implement the MAML algorithm.
- MAML's main goal is optimizing the model parameters so that a small number of gradient steps would produce a maximum effective behaviour on a new task.

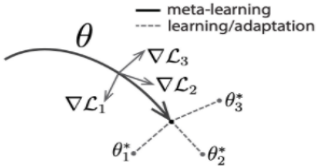

Figure 1. Diagram of our model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks.

First step of Few-shot implementation

- The meta-optimization across tasks is performed with stochastic gradient descent.
- The model parameters are noted as theta θ and β is the meta step size

Algorithm 2 MAML for Few-Shot Supervised Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (2) or (3)
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 2 or 3
11: **end while**

Figure 17: Model-Agnostic Meta-Learning

First step of Few-shot implementation

Algorithm 2 MAML for Few-Shot Supervised Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters

1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K datapoints $\mathcal{D} = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation (2) or (3)
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample datapoints $\mathcal{D}_i' = \{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$ from $\mathcal{T}_i$ for the meta-update
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 2 or 3
11: **end while**

Figure 18: MAML algorithm

```python
#In our case the model is a yolov4 model
model=ConvolutionalNeuralNetwork(out_features=7) #we propose a 7-way setting
meta_optimizer=torch.optim.Adam(model.parameters(), lr=1e-3)
#Sample batch of tasks/ Line_4
for task in batch:

  #Ligne_5 and Line_6
  train_inputs, train_targets=task['support']
  test_inputs, test_targets=task['query']

  train_logit=model(train_input) #Evaluate the model
  inner_loss=F.cross_entropy(train_logit, train_target)
  model.zero_grad()

  #Line_7
  #estimate the gradients on the inner loss on the model.meta_params
  grads=torch.autograd.grad(inner, model.meta_params(), create_graph=True)
  params=OrderedDict()  #define a dictionary of params

  #iterate on our model.meta_params and we translate the equation
  #(do a gradient step with a time step size)
  for (name, param) , grad in zip (model.meta_named_pars(), grads):
    params[name]=param - step_size * grad

  #Line_8 to line_10
  test_logit=model(test_input, params=params)

  #the sum of all the losses over all the tasks
  outer_loss+=F.cross_entropy(test_logit, test_target)
outer_loss.backward() #apply the backward pass over all the tasks
meta_optimizer.step() #apply the optimizer with the specific step size
```

Figure 19: Pytorch implementation of MAML algorithm

THANK YOU FOR YOUR ATTENTION