ФФ-21, Максакова Дар’я

Лабораторна робота №6: «Робота з відкритим програмним кодом»

Мета роботи: отримати навички використання існуючого програмного коду.

Хід виконання роботи:

1. Вибір наукової статті та відкритого коду.

Обрана наукова стаття (англійською, не старіша 2016 р.): Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, Sergey Zagoruyko. «End-to-End Object Detection with Transformers (DETR)», 2020.

Посилання на статтю:

- arXiv (сторінка): https://arxiv.org/abs/2005.12872
- PDF (arXiv): https://arxiv.org/pdf/2005.12872

Відповідний відкритий код на GitHub:

- Official repository: https://github.com/facebookresearch/detr

Демо для запуску:

- Colab notebook: https://colab.research.google.com/github/facebookresearch/detr/blob/colab/notebooks/detr_demo.ipynb#scrollTo=RSnU5JFxGeDe

2. Короткий опис результатів статті, запропонованого алгоритму/ідеї, способу досягнення та новизни.

Стаття «End-to-End Object Detection with Transformers (DETR)» присвячена задачі детекції об’єктів на зображенні (object detection), тобто автоматичному пошуку об’єктів на фотографії/кадрі. Для кожного знайденого об’єкта потрібно визначити його клас (class) і побудувати обмежувальну рамку

(bounding box або скорочено bbox) – прямокутник, який обводить об’єкт так, щоб він був всередині цієї рамки.

Основний результат роботи полягає в тому, що автори пропонують детектор, який працює від початку до кінця однією моделлю (end-to-end). Це означає, що замість складного ланцюжка окремих блоків і правил, які підбирають вручну, вся задача розв’язується однією узгодженою нейромережею, яка навчається одразу видавати готові рамки та класи. У багатьох класичних підходах детекція будується як кілька етапів: спочатку створюються кандидатні області/рамки (proposals – тобто припущення «можливо, тут є об’єкт»), або використовуються якорі (anchors – заздалегідь задані шаблонні рамки різних розмірів і форм), а потім застосовується пост-обробка NMS (non-maximum suppression – процедура, яка прибирає дублікати рамок: якщо модель намалювала багато майже однакових рамок на один об’єкт, NMS залишає одну найкращу, а інші видаляє). У DETR задум інший: прибрати ці ручні частини й навчити модель одразу давати чистий результат.

Головна ідея DETR: детекцію формулюють як задачу передбачення множини об’єктів (set prediction). Множина тут означає, що модель повертає не послідовність кроків і не тисячі кандидатів, а одразу список з фіксованої кількості прогнозів, наприклад 100. Кожен прогноз у цьому списку має вигляд: (клас, рамка). Частина прогнозів відповідає реальним об’єктам, а решта прогнозів спеціально позначається як «об’єкта немає» (no object), тобто модель вчиться, що далеко не всі 100 місць обов’язково зайняті.

Як це досягається (будова алгоритму). DETR складається з двох великих частин:

Перша частина – блок виділення ознак із зображення (CNN / convolutional neural network, згорткова нейромережа). Її ще називають backbone (хребет моделі), бо це базовий модуль, який перетворює зображення на карту ознак (feature map). Карта ознак – це не звичайна картинка, а набір чисел, які

описують що і де є на зображенні: наприклад, у яких областях видно контури, текстури, характерні фрагменти об’єктів. Як backbone часто беруть ResNet (це популярна архітектура згорткових мереж для роботи із зображеннями).

Друга частина – трансформер (Transformer). Це архітектура, яка вміє обробляти інформацію так, щоб кожна частина даних могла взаємодіяти з кожною іншою через механізм уваги (attention). Механізм уваги можна пояснити як модель, що обчислює, на які області потрібно звернути більше уваги, тобто, які області важливіші для поточного рішення, і як різні частини зображення пов’язані між собою. У DETR використовується структура енкодер-декодер (encoder-decoder): енкодер отримує карту ознак з backbone і формує узагальнене представлення зображення з урахуванням всього контексту, декодер, використовуючи це представлення, формує кінцеві прогнози.

Щоб трансформер міг розуміти, де саме на зображенні знаходяться ознаки, додають позиційне кодування (positional encoding). Це додаткові числа, які підказують моделі координатну інформацію, бо сам по собі трансформер не знає, де верх/низ або ліво/право, якщо йому не додати позиційні ознаки.

Найхарактерніша деталь DETR – object queries (запити об’єктів). Це набір навчуваних векторів, які можна уявляти як 100 віртуальних пошукачів об’єктів. Кожен такий запит звертається до представлення зображення в трансформері і намагається знайти один об’єкт. Якщо об’єкт знаходиться, то запит повертає його клас і рамку. Якщо ні – повертає no object. Завдяки цьому модель не мусить перебирати тисячі кандидатних рамок: вона працює зі сталою кількістю місць під об’єкти.

Як модель навчається, щоб не було дублікатів рамок. У класичних детекторах дублікати прибирають пост-обробкою NMS. У DETR це робиться інакше: під час навчання використовується парне зіставлення (bipartite

matching), яке будується угорським алгоритмом (Hungarian algorithm). Суть така: на кожному зображенні є справжні об’єкти з розмітки (ground truth – правильні відповіді) і є набір прогнозів моделі (наприклад, 100 прогнозів). Потрібно знайти відповідність, який прогноз відповідає якому реальному об’єкту так, щоб сумарна помилка була мінімальною, і щоб один реальний об’єкт не був призначений одразу кільком прогнозам. Угорський алгоритм якраз і знаходить найкраще таке зіставлення. Після того як відповідності знайдені, модель штрафується (через функцію втрат, loss) за дві речі: неправильний клас і неправильну рамку. Функція втрат – це число, яке показує, наскільки модель помиляється, навчання зводиться до мінімізації цього числа.

Результати, яких досягли автори. У роботі демонструється, що DETR дає конкурентну якість на стандартному датасеті COCO (великий публічний набір зображень з розміткою об’єктів, який часто використовують для порівняння детекторів). Якість вимірюють метрикою AP (average precision – узагальнений показник точності/повноти для детекції). У офіційному репозиторії наведено приклади результатів та цифри AP для базових конфігурацій моделі (наприклад, з backbone ResNet-50 чи ResNet-101).

У чому новизна і чому підхід кращий за типові ідеї. Новизна DETR у тому, що детекція представлена як передбачення множини і модель навчається уникати дублікатів не через ручну пост-обробку, а через оптимальне зіставлення прогнозів із реальними об’єктами під час навчання. Це робить весь алгоритм більш узгодженим: модель сама вчиться видавати один прогноз на один об’єкт. Додатково трансформер із механізмом уваги дозволяє враховувати глобальний контекст зображення (інформацію з усієї сцени одразу), що може бути важливо у складних сценах, де об’єкти частково перекриваються.

3. Розбір коду, запуск та демонстрація демо DETR.

У якості демо було використано Standalone Colab Notebook з репозиторію DETR (minimal implementation), яке було взято у https://github.com/facebookresearch/detr з README.md. Демо запускає попередньо натреновану модель DETR і візуалізує результати детекції об'єктів (рамки та класи) на тестовому зображенні.

```
from PIL import Image
import requests
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'retina'

import torch
from torch import nn
from torchvision.models import resnet50
import torchvision.transforms as T
torch.set_grad_enabled(False);
```

- PIL.Image і requests тут потрібні, щоб завантажити/відкрити картинку (з URL або локально) і подати її в модель.
- matplotlib.pyplot – щоб потім намалювати результат (bbox, підписи класів).
- torch, torch.nn – база для моделі та шарів.
- resnet50 – CNN "backbone" (витягує ознаки з картинки).
- torchvision.transforms as T – перетворення зображення в тензор (resize/normalize тощо).
- torch.set_grad_enabled(False) вимикає градієнти (це демо для inference, швидше й менше пам'яті).

```
class DETRdemo(nn.Module):
    """
    Demo DETR implementation.

    Demo implementation of DETR in minimal number of lines, with the
    following differences wrt DETR in the paper:
    * learned positional encoding (instead of sine)
    * positional encoding is passed at input (instead of attention)
    * fc bbox predictor (instead of MLP)
    The model achieves ~40 AP on COCO val5k and runs at ~28 FPS on Tesla V100.
    Only batch size 1 supported.
    """
```

- Це клас моделі DETR (Detection Transformer), але у скороченому demo-варіанті.
- Автор одразу попереджає про відмінності від оригінального DETR:
    - позиційні ембедінги навчаються (learned), а не синусоїдальні;
    - позиційний сигнал додають на вході трансформера;
    - bbox-регресор — просто Linear(…,4), а не MLP (в оригіналі 3-шаровий MLP).

```
def __init__(self, num_classes, hidden_dim=256, nheads=8,
             num_encoder_layers=6, num_decoder_layers=6):
    super().__init__()

    # create ResNet-50 backbone
    self.backbone = resnet50()
    del self.backbone.fc

    # create conversion layer
    self.conv = nn.Conv2d(2048, hidden_dim, 1)

    # create a default PyTorch transformer
    self.transformer = nn.Transformer(
        hidden_dim, nheads, num_encoder_layers, num_decoder_layers)
```

```
        # prediction heads, one extra class for predicting non-empty slots
        # note that in baseline DETR linear_bbox layer is 3-layer MLP
        self.linear_class = nn.Linear(hidden_dim, num_classes + 1)
        self.linear_bbox = nn.Linear(hidden_dim, 4)

        # output positional encodings (object queries)
        self.query_pos = nn.Parameter(torch.rand(100, hidden_dim))

        # spatial positional encodings
        # note that in baseline DETR we use sine positional encodings
        self.row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
        self.col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
```

- num_classes – кількість класів об’єктів (без фонового).
- hidden_dim=256 – розмірність ознак, з якою працюватиме трансформер.
- backbone = resnet50(); del backbone.fc:
  - ResNet-50 використовується як екстрактор ознак.
  - fully-connected голова (fc) видаляється, бо класифікація ImageNet тут не потрібна.
- self.conv = Conv2d(2048 → 256, kernel=1):
  - вихід ResNet-50 після layer4 має 2048 каналів,
  - 1×1 conv зменшує канали до hidden_dim, щоб узгодити з трансформером.
- self.transformer = nn.Transformer(...):
  - стандартний Transformer з PyTorch: encoder+decoder.
  - nheads=8 – кількість голів уваги.
  - num_encoder_layers/num_decoder_layers – кількість шарів.
- self.linear_class: 256 → (num_classes + 1): “+1” – додатковий клас “no object” (порожній слот), це типова ідея DETR.
- self.linear_bbox: 256 → 4: 4 числа – параметри bbox (у DETR зазвичай нормовані координати).

- self.query_pos = Parameter(100, 256): 100 “object queries” – це фіксована кількість слотів, які питають трансформер: знайди мені об’єкти.
- row_embed/col_embed:
  - позиційні ембедінги по рядках і колонках (для H×W фіч-мапи),
  - розбито навпіл (hidden_dim//2 + hidden_dim//2 = hidden_dim).

```
def forward(self, inputs):
    # propagate inputs through ResNet-50 up to avg-pool layer
    x = self.backbone.conv1(inputs)
    x = self.backbone.bn1(x)
    x = self.backbone.relu(x)
    x = self.backbone.maxpool(x)

    x = self.backbone.layer1(x)
    x = self.backbone.layer2(x)
    x = self.backbone.layer3(x)
    x = self.backbone.layer4(x)
```

- inputs – тензор зображення форми приблизно [B, 3, H, W]. В цьому demo очікується batch size 1.
- Блоки conv1/bn1/relu/maxpool + layer1..layer4 – це стандартний forward ResNet-50 до останнього convolutional блоку.
- На виході x має форму [1, 2048, H/32, W/32] (типово, бо ResNet дає downsample ×32).

```
    # convert from 2048 to 256 feature planes for the transformer
    h = self.conv(x)
```

- Тут h стає [1, 256, H/32, W/32].
- Це “token features” для трансформера (після позиційного додавання їх перетворять у послідовність).

```
    # construct positional encodings
    H, W = h.shape[-2:]
    pos = torch.cat([
        self.col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
        self.row_embed[:H].unsqueeze(1).repeat(1, W, 1),
    ], dim=-1).flatten(0, 1).unsqueeze(1)
```

- H, W – просторові розміри фіч-мапи.
- self.col_embed[:W] дає W векторів для колонок (кожен розмірності 128).
    - unsqueeze(0) робить форму [1, W, 128]
    - repeat(H,1,1) → [H, W, 128] (однакове для кожного рядка)
- self.row_embed[:H] аналогічно робить [H, W, 128] (однакове для кожної колонки)
- torch.cat(..., dim=-1) → [H, W, 256] – повний позиційний вектор.
- flatten(0,1) перетворює сітку H×W у послідовність довжини H*W: [H*W, 256]
- unsqueeze(1) → [H*W, 1, 256] (де “1” – batch dimension для nn.Transformer, бо він очікує [S, N, E]).

```
# propagate through the transformer
h = self.transformer(pos + 0.1 * h.flatten(2).permute(2, 0, 1),
                     self.query_pos.unsqueeze(1)).transpose(0, 1)
```

- h.flatten(2) з [1,256,H,W] робить [1,256,H*W].
- permute(2,0,1) → [H*W, 1, 256] – рівно як pos.
- pos + 0.1 * features:
    - додають позиційний сигнал до фіч.
    - множник 0.1 просто масштабує внесок ознак (так зробили в цьому demo).
- self.query_pos.unsqueeze(1) з [100,256] → [100,1,256]: це “target sequence” для decoder (100 запитів).
- self.transformer(src, tgt):
    - src = (pos + features) довжини H*W,
    - tgt = object queries довжини 100,
    - вихід має форму [100,1,256] (decoder outputs).
- transpose(0,1) → [1,100,256] (зручніше далі: batch first).

```
        # finally project transformer outputs to class labels and bounding boxes
        return {'pred_logits': self.linear_class(h),
                'pred_boxes': self.linear_bbox(h).sigmoid()}
```

- h тут [1, 100, 256] – по 100 слотів (кандидатів на об’єкти).
- pred_logits = Linear(256 → num_classes+1): вихід [1, 100, num_classes+1], це сирі логіти класів.
- pred_boxes = Linear(256 → 4), потім sigmoid():
  - [1, 100, 4] у діапазоні (0,1),
  - зазвичай інтерпретують як нормовані bbox-параметри (часто cx, cy, w, h), щоб легко масштабувати під розмір зображення.

```
detr = DETRdemo(num_classes=91)
state_dict = torch.hub.load_state_dict_from_url(
    url='https://dl.fbaipublicfiles.com/detr/detr_demo-da2a99e9.pth',
    map_location='cpu', check_hash=True)
detr.load_state_dict(state_dict)
detr.eval();
```

- detr = DETRdemo(num_classes=91)
- Тут створюється порожня модель DETRdemo (архітектура + випадкові початкові ваги).
- Чому саме 91:
  - у COCO є 80 реальних класів об’єктів, але їхні ID в COCO йдуть не суцільно від 1 до 80, а можуть мати дірки і доходять до 90.
  - тому в списку CLASSES є 'N/A' як заглушки для пропущених ID, і загальна довжина виходить 91 (від 0 до 90).

- state_dict = torch.hub.load_state_dict_from_url(...)
- state_dict – це словник ваг моделі (параметри всіх шарів), який скачують по URL.
- map_location='cpu' означає: навіть якщо у тебе є GPU, ваги спочатку завантажаться на CPU (так безпечніше і простіше для демо в Colab).

- check_hash=True: Colab перевіряє, що файл скачався правильно (щоб не було битих ваг).
- detr.load_state_dict(state_dict) – підставляє скачані ваги у створену модель (тобто модель стає навченою, а не випадковою).
- detr.eval()
- переводить модель в режим оцінювання (inference):
  - вимикається dropout (якщо він є),
  - batchnorm (якщо є) використовує накопичені статистики, а не рахує їх заново.
- у демо це важливо, бо ми хочемо стабільні передбачення, а не “трошки випадкові”.

```
# COCO classes
CLASSES = [
    'N/A', 'person', 'bicycle', 'car', 'motorcycle', 'airplane', 'bus',
    'train', 'truck', 'boat', 'traffic light', 'fire hydrant', 'N/A',
    'stop sign', 'parking meter', 'bench', 'bird', 'cat', 'dog', 'horse',
    'sheep', 'cow', 'elephant', 'bear', 'zebra', 'giraffe', 'N/A', 'backpack',
    'umbrella', 'N/A', 'N/A', 'handbag', 'tie', 'suitcase', 'frisbee', 'skis',
    'snowboard', 'sports ball', 'kite', 'baseball bat', 'baseball glove',
    'skateboard', 'surfboard', 'tennis racket', 'bottle', 'N/A', 'wine glass',
    'cup', 'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple', 'sandwich',
    'orange', 'broccoli', 'carrot', 'hot dog', 'pizza', 'donut', 'cake',
    'chair', 'couch', 'potted plant', 'bed', 'N/A', 'dining table', 'N/A',
    'N/A', 'toilet', 'N/A', 'tv', 'laptop', 'mouse', 'remote', 'keyboard',
    'cell phone', 'microwave', 'oven', 'toaster', 'sink', 'refrigerator', 'N/A',
    'book', 'clock', 'vase', 'scissors', 'teddy bear', 'hair drier',
    'toothbrush'
]

# colors for visualization
COLORS = [[0.000, 0.447, 0.741], [0.850, 0.325, 0.098], [0.929, 0.694, 0.125],
          [0.494, 0.184, 0.556], [0.466, 0.674, 0.188], [0.301, 0.745, 0.933]]
```

- CLASSES – список назв класів датасету COCO, щоб модель могла підписати знайдені об’єкти “cat”, “remote”, “couch” тощо.
- 'N/A' тут означає “немає класу/порожнє місце” (службові індекси в списку).
- COLORS – набір кольорів для малювання рамок (bounding boxes) на картинці.

```
# standard PyTorch mean-std input image normalization
transform = T.Compose([
    T.Resize(800),
    T.ToTensor(),
    T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])
```

- Тут автор текстом пояснює формат рамок, який видає DETR:
    - [x_center, y_center, w, h] – центр рамки та її ширина/висота.
    - “relative coordinates” = координати нормовані в діапазоні 0…1 відносно розміру зображення.
    - Для малювання треба перевести в [x0, y0, x1, y1] (лівий верхній і правий нижній кути) і в пікселі.
- transform = T.Compose([...]) – передобробка зображення перед подачею в модель:
    - T.Resize(800) – змінює розмір (довша/коротша сторона приводиться до типового масштабу).
    - T.ToTensor() – робить тензор PyTorch (значення пікселів у числах).
    - T.Normalize(mean, std) – нормалізація під ImageNet (так навчали ResNet-50, тому треба так само на вході).

```
# for output bounding box post-processing
def box_cxcywh_to_xyxy(x):
    x_c, y_c, w, h = x.unbind(1)
    b = [(x_c - 0.5 * w), (y_c - 0.5 * h),
         (x_c + 0.5 * w), (y_c + 0.5 * h)]
    return torch.stack(b, dim=1)
```

- x – це набір рамок у форматі [x_center, y_center, w, h] (і все це ще нормоване 0…1).
- unbind(1) розпаковує по стовпцях: окремо центри та розміри.
- Формула переводу в кути:

- x0 = x_center − w/2
- y0 = y_center − h/2
- x1 = x_center + w/2
- y1 = y_center + h/2

- torch.stack(..., dim=1) збирає назад у тензор форми [N, 4].

```
def rescale_bboxes(out_bbox, size):
    img_w, img_h = size
    b = box_cxcywh_to_xyxy(out_bbox)
    b = b * torch.tensor([img_w, img_h, img_w, img_h], dtype=torch.float32)
    return b
```

- out_bbox – рамки у відносних координатах (0…1).
- size – розмір вихідного зображення (ширина, висота) у пікселях.
- Після box_cxcywh_to_xyxy ми маємо [x0,y0,x1,y1], але ще в 0…1.
- Множення на [img_w, img_h, img_w, img_h] переводить координати у пікселі, щоб рамки правильно лягли на реальну картинку.

```
def detect(im, model, transform):
    # mean-std normalize the input image (batch-size: 1)
    img = transform(im).unsqueeze(0)
```

```
# demo model only support by default images with aspect ratio between 0.5 and 2
# if you want to use images with an aspect ratio outside this range
# rescale your image so that the maximum size is at most 1333 for best results
assert img.shape[-2] <= 1600 and img.shape[-1] <= 1600, 'demo model only supports images up to 1600 pixels on each side'
```

```
    # propagate through the model
    outputs = model(img)

    # keep only predictions with 0.7+ confidence
    probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]
    keep = probas.max(-1).values > 0.7

    # convert boxes from [0; 1] to image scales
    bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size)
    return probas[keep], bboxes_scaled
```

- Це головна функція демо: вона бере картинку → ганяє через модель → відбирає хороші детекції → повертає ймовірності та рамки.

- img = transform(im).unsqueeze(0):
  - transform(im) робить тензор [3, H, W].
  - unsqueeze(0) додає batch-вісь → [1, 3, H, W], бо модель очікує батч.
- assert ... <= 1600:
  - перевірка: demo не підтримує дуже великі картинки (інакше може впасти по пам’яті/часу).
- outputs = model(img):
  - outputs – словник з двома ключами:
    - 'pred_logits' – логіти класів для кожного запиту (100 слотів).
    - 'pred_boxes' – рамки для кожного слоту (100 штук).
- probas = outputs['pred_logits'].softmax(-1)[0, :, :-1]:
  - softmax(-1) переводить логіти у ймовірності по класах.
  - [0, :, :-1] означає:
    - беремо перший елемент батчу,
    - беремо всі 100 слотів,
    - відкидаємо останній клас “no object” (бо він не об’єкт).
- keep = probas.max(-1).values > 0.7:
  - для кожного слоту беремо максимальну ймовірність серед класів,
  - залишаємо тільки ті, де впевненість > 0.7 (поріг).
- bboxes_scaled = rescale_bboxes(outputs['pred_boxes'][0, keep], im.size):
  - беремо тільки рамки тих слотів, які пройшли поріг,
  - масштабуємо в пікселі.
- return probas[keep], bboxes_scaled: повертає оцінки класів і рамки тільки для відібраних об’єктів.

```
url = 'http://images.cocodataset.org/val2017/000000039769.jpg'
im = Image.open(requests.get(url, stream=True).raw)

scores, boxes = detect(im, detr, transform)
```

- Тут береться тестове зображення з COCO по URL.

- Image.open(requests.get(...).raw) – завантажили байти картинки з інтернету і відкрили як PIL.Image.
- scores, boxes = detect(...) – запускаємо весь pipeline:
    - scores – ймовірності класів для знайдених об’єктів,
    - boxes – рамки (в пікселях) для цих об’єктів.

```
def plot_results(pil_img, prob, boxes):
    plt.figure(figsize=(16,10))
    plt.imshow(pil_img)
    ax = plt.gca()
    for p, (xmin, ymin, xmax, ymax), c in zip(prob, boxes.tolist(), COLORS * 100):
        ax.add_patch(plt.Rectangle((xmin, ymin), xmax - xmin, ymax - ymin,
                                   fill=False, color=c, linewidth=3))
        cl = p.argmax()
        text = f'{CLASSES[cl]}: {p[cl]:0.2f}'
        ax.text(xmin, ymin, text, fontsize=15,
                bbox=dict(facecolor='yellow', alpha=0.5))
    plt.axis('off')
    plt.show()

plot_results(im, scores, boxes)
```

- plot_results – це чисто візуалізація (постобробка результатів для демонстраці).
- plt.figure(figsize=(16,10)) – задає великий розмір, щоб добре видно рамки.
- plt.imshow(pil_img) – показує картинку.
- ax = plt.gca() – отримуємо вісь, на яку будемо додавати прямокутники.
- цикл for p, (xmin,ymin,xmax,ymax), c in zip(...):
    - p – вектор ймовірностей по класах для одного об’єкта,
    - (xmin,ymin,xmax,ymax) – координати рамки,
    - c – колір.
- ax.add_patch(Rectangle(...)): малює прямокутник рамки з товщиною linewidth=3.

- cl = p.argmax(): argmax = індекс класу з найбільшою ймовірністю (тобто який клас модель обрали).
- text = f'{CLASSES[cl]}: {p[cl]:0.2f}': підпис типу “cat: 1.00”.
- ax.text(..., bbox=dict(facecolor='yellow', alpha=0.5)): малює текст на жовтому фоні, щоб читалося.
- plot_results(im, scores, boxes) – виклик, який і дає фінальну картинку з рамками.

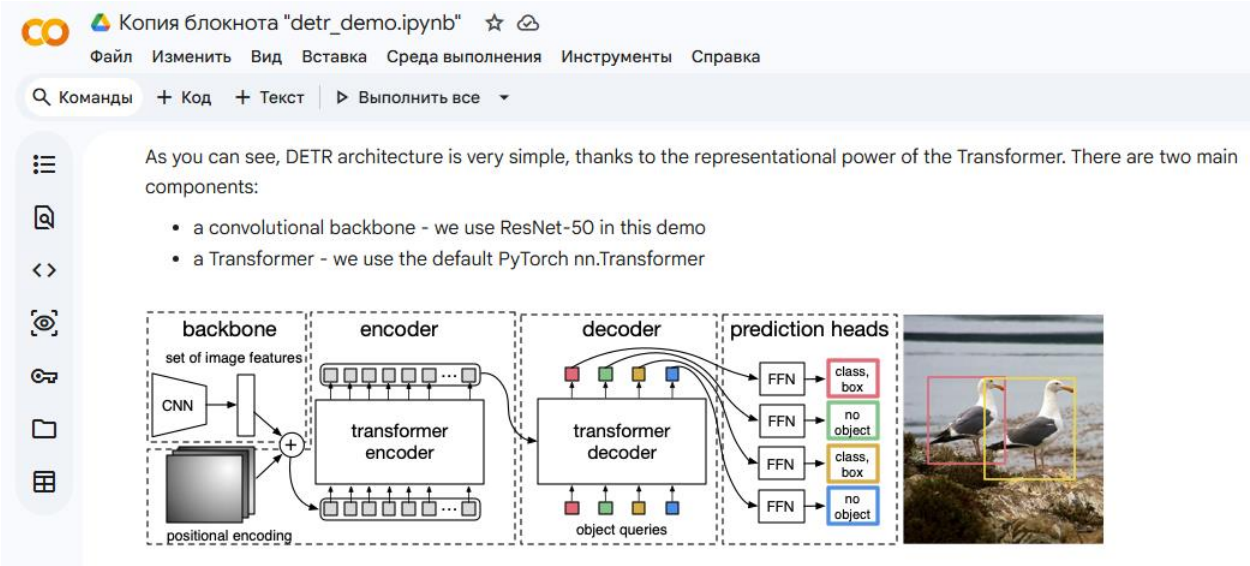


Рис. Схема архітектури DETR (demo з GitHub)

На рисунку показано принцип роботи DETR: зображення спочатку проходить через згортковий backbone (у демо використано ResNet-50) і перетворюється на карту ознак. Далі ознаки разом із позиційним кодуванням подаються в Transformer encoder/decoder. Decoder працює з фіксованою кількістю object queries (запитів об’єктів), а на виході prediction heads формують для кожного запиту прогноз класу та координат рамки або варіант no object (тобто слот без об’єкта). Ця схема пояснює, яку саме модель запускає ноутбук.

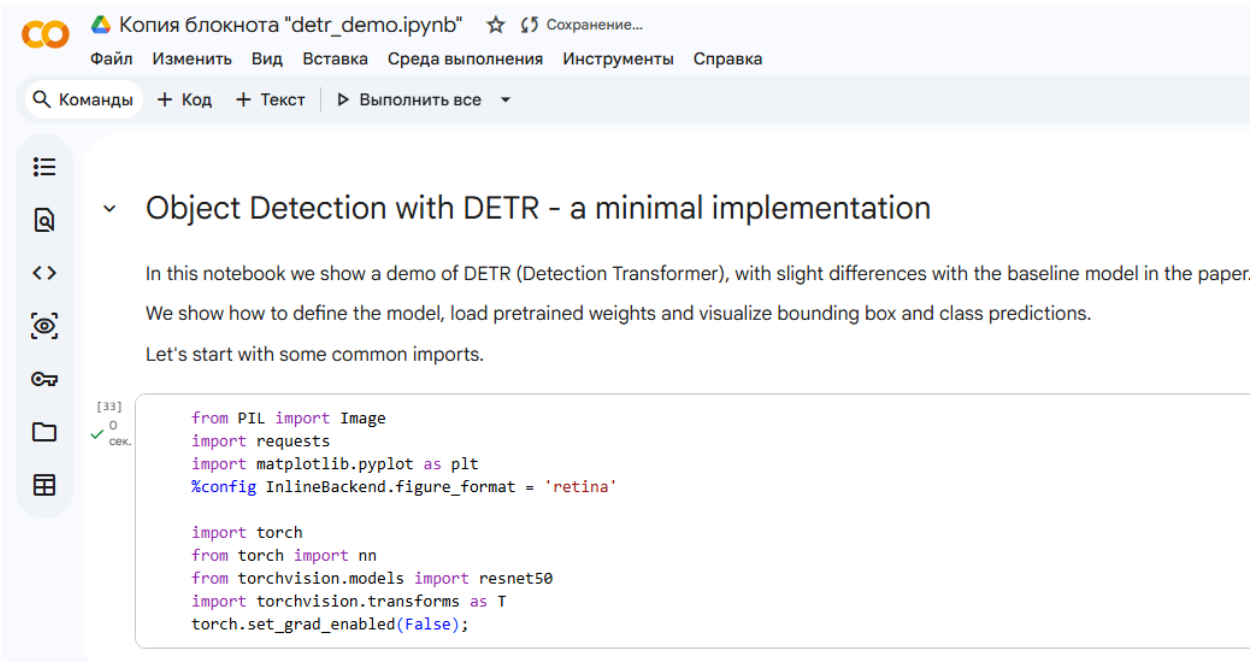


Рис. Запуск Standalone Colab Notebook «Object Detection with DETR – a minimal implementation»

Показано інтерфейс Google Colab із відкритим демо-ноутбуком з репозиторію DETR. Видно початкову комірку з імпортами бібліотек (PIL, requests, matplotlib, PyTorch, torchvision) та позначення виконання комірки (зелена відмітка/номер). Це підтверджує, що демо дійсно було запущене й середовище готове до подальших обчислень (завантаження моделі, інференс, візуалізація).

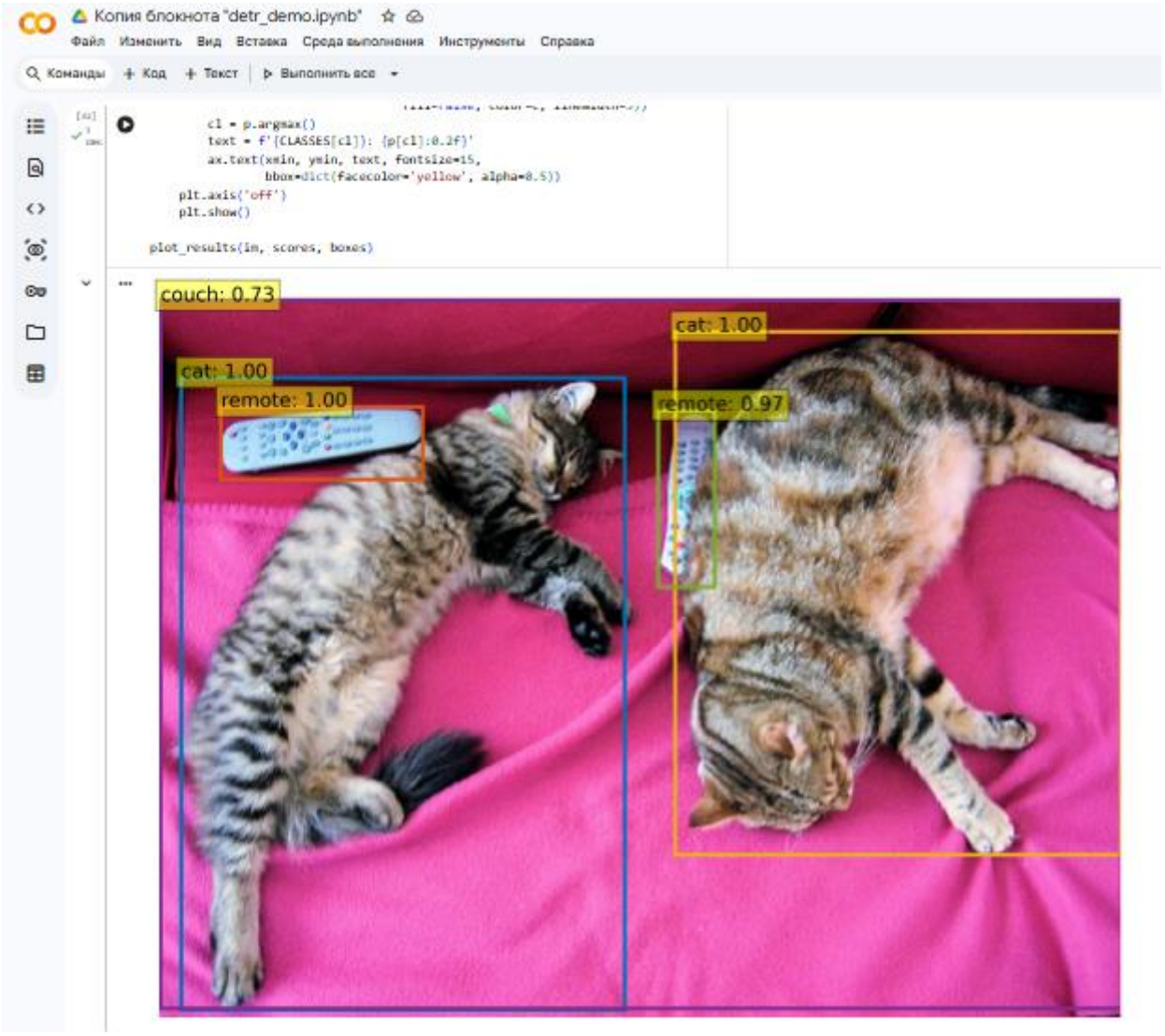


Рис. Результат роботи демо DETR: детекція об’єктів на зображенні

Показано фінальний вихід демо після виконання інференсу: модель знайшла об’єкти на зображенні та намалювала bounding boxes (прямокутні рамки) з підписами класів і значеннями впевненості (confidence). На прикладі виявлено об’єкти класів “cat”, “remote” та “couch” із відповідними оцінками (наприклад, cat: 1.00, remote: 0.97, couch: 0.73). Це і є демонстрація запущеного демо з GitHub – видно, що модель не просто завантажилась, а реально сформувала прогноз і візуалізувала результат.

4. Можливі шляхи покращення отриманих результатів у статті

1. Multi-scale Test-Time Augmentation (TTA) з клас-орієнтованим Non-Maximum Suppression (NMS) для об’єднання детекцій

```
def detect(im, model, transform, threshold=0.7, use_nms=True, iou_threshold=0.5, scales=(800, 1000, 1200)):
    """
    im: PIL image
    threshold: поріг впевненості (чим вище, тим менше рамок)
    scales: multi-scale TTA (різні розміри Resize)
    """
```

- Розширено detect так, щоб вона вміла:
    1. запускатися на кількох масштабах (TTA) через scales
    2. прибирати дублікати рамок через NMS (use_nms=True)
- threshold – фільтр «беремо тільки впевнені детекції».
- iou_threshold – наскільки рамки можуть перекриватись, щоб одну прибрати як дубль.

```
device = next(model.parameters()).device
```

- Визначає, де зараз модель: CPU або GPU.
- Потім ми будемо відправляти туди ж і картинку (to(device)), щоб не було помилок “tensor on cpu / model on cuda”.

```
all_probs = []
all_boxes = []
```

Сюди збираємо результати з кожного масштабу. Після циклу об’єднаємо їх у великий список детекцій.

```
for s in scales:
    t = T.Compose([
        T.Resize(s),
        T.ToTensor(),
        T.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ])

    img = t(im).unsqueeze(0).to(device)
```

- Це і є multi-scale TTA: ми змінюємо тільки Resize(s) (800, 1000, 1200).
- Normalize(...) – стандарт ImageNet, як і було в оригінальному демо.
- t(im) робить тензор [3, H, W].
- unsqueeze(0) додає batch → [1, 3, H, W].
- .to(device) переносить на CPU/GPU разом з моделлю.

```
assert img.shape[-2] <= 1600 and img.shape[-1] <= 1600, "Image too large for this demo"
```

Захист, щоб демо не впало, якщо після Resize картинка стала занадто великою. Це те, що було логікою в оригіналі: модель/демо розраховані на певні розміри.

```
outputs = model(img)

prob = outputs['pred_logits'].softmax(-1)[0, :, :-1]  # [num_queries, num_classes]
scores, labels = prob.max(-1)                         # [num_queries], [num_queries]

keep = scores > threshold
scores = scores[keep]
labels = labels[keep]

boxes = rescale_bboxes(outputs['pred_boxes'][0, keep].detach().cpu(), im.size)
scores = scores.detach().cpu()
labels = labels.detach().cpu()
```

- pred_logits -> softmax дає ймовірності по класах для кожного query.
- :-1 відкидає клас no-object.
- scores, labels = prob.max(-1) – для кожного query беремо найкращий клас і його впевненість.
- threshold – залишаємо тільки впевнені детекції.
- rescale_bboxes(...) переводить bbox з нормованого формату DETR у піксельні координати хуху для малювання.

```
        probs_keep = torch.zeros((scores.shape[0], len(CLASSES)))
        probs_keep[torch.arange(scores.shape[0]), labels] = scores

        all_probs.append(probs_keep)
        all_boxes.append(boxes)

    probs_keep = torch.cat(all_probs, dim=0)
    bboxes_scaled = torch.cat(all_boxes, dim=0)
```

- plot_results очікує, що для кожної рамки є вектор ймовірностей по класах.
- Але після max у нас є лише (label, score), тому робимо “one-hot” вектор: все 0, а в одному класі — score.
- Потім зліплюємо результати з усіх масштабів: тепер у нас супер-набір рамок.

```
if use_nms and bboxes_scaled.shape[0] > 0:
    final_keep = []
    labels_all = probs_keep.argmax(dim=1)
    scores_all = probs_keep.max(dim=1).values

    for c in labels_all.unique():
        mask = labels_all == c
        inds = torch.where(mask)[0]
        keep_inds = nms(bboxes_scaled[inds], scores_all[inds], iou_threshold=iou_threshold)
        final_keep.append(inds[keep_inds])


    final_keep = torch.cat(final_keep) if len(final_keep) > 0 else torch.empty((0,), dtype=torch.long)

    final_scores = probs_keep[final_keep].max(dim=1).values
    order = final_scores.argsort(descending=True)
    final_keep = final_keep[order]

    probs_keep = probs_keep[final_keep]
    bboxes_scaled = bboxes_scaled[final_keep]

return probs_keep, bboxes_scaled
```

- Після multi-scale TTA часто з’являються дублікати рамок (той самий об’єкт кілька разів).
- NMS прибирає дублікати, залишаючи найвпевненіші.
- Дуже правильно, що робиться окремо по кожному класу: рамка “cat” не приб’є рамку “remote”.

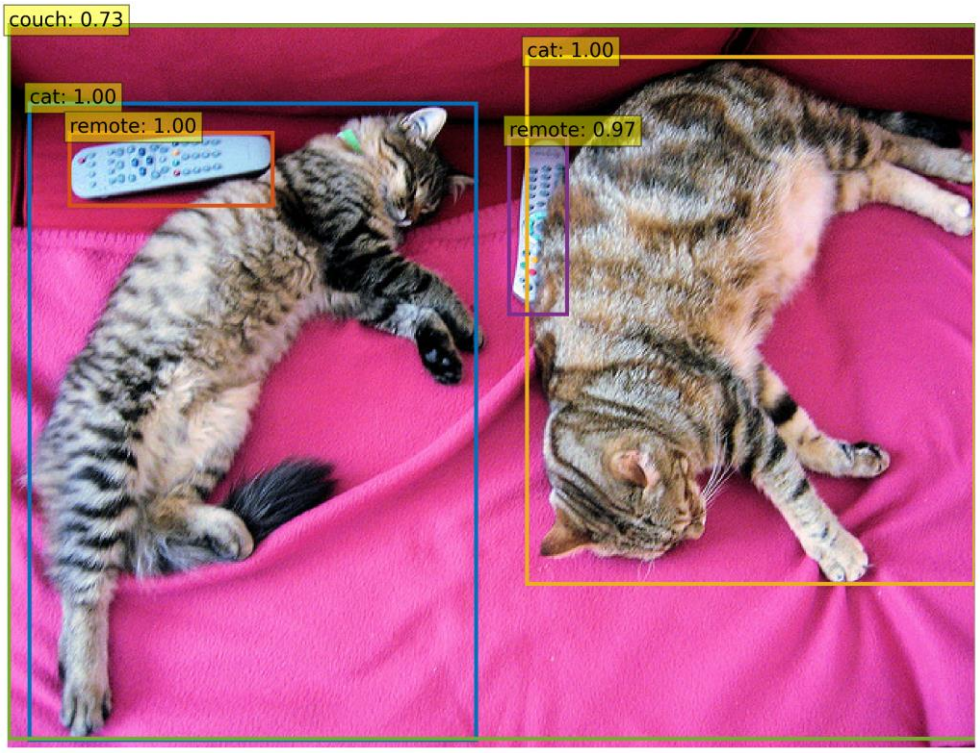
```
scores_base, boxes_base = detect(im, detr, transform, threshold=0.7, use_nms=True, iou_threshold=0.5, scales=(800,))
scores_tta, boxes_tta = detect(im, detr, transform, threshold=0.7, use_nms=True, iou_threshold=0.5, scales=(800, 1000, 1200))

print("Base detections:", len(boxes_base))
print("TTA  detections:", len(boxes_tta))

def get_best_score_for(label_name, probs):
    if probs.numel() == 0:
        return None
    idx = CLASSES.index(label_name)
    vals = probs[:, idx]
    m = vals.max().item() if (vals > 0).any() else None
    return m

print("couch score (base):", get_best_score_for("couch", scores_base))
print("couch score (tta) :", get_best_score_for("couch", scores_tta))


plot_results(im, scores_base, boxes_base)
plot_results(im, scores_tta, boxes_tta)
```

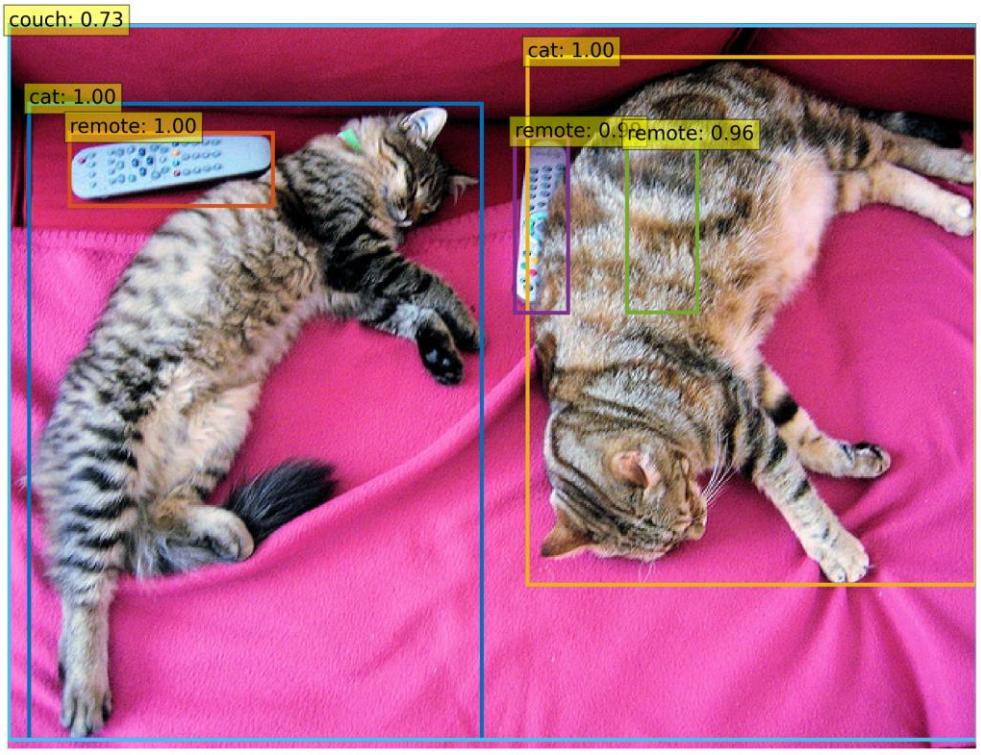
- Base – один масштаб (як у стандартному демо).
- TTA – кілька масштабів.
- Ми порівнюємо:

    1. скільки рамок стало
    2. чи змінився найкращий score для вибраного класу
    3. візуально, чи з’явилися корисні детекції / чи прибрались дублікати.

Результати:

```
Base detections: 5
TTA  detections: 6
couch score (base): 0.7264321446418762
couch score (tta) : 0.7264321446418762
```





У базовому запуску (один масштаб 800) модель DETR дала 5 детекцій на зображенні: дві кішки, диван (couch) і пульт (remote) (плюс ще одна рамка залежно від того, як саме пройшов фільтр по порогу). Після додавання multi-scale TTA (прогони на 800, 1000, 1200) кількість детекцій зросла до 6. Це означає, що при додаткових масштабах модель знайшла ще одну рамку, яка

пройшла поріг впевненості threshold=0.7 і не була відкинута після постобробки.

Найважливіше, що зміна проявилась не на великих об’єктах, а на дрібніших/складніших. Для класу “couch” максимальна впевненість залишилася однаковою в base та tta (значення score для “couch” не змінилося). Це нормальна ситуація: диван займає велику частину кадру, добре читається вже на базовому масштабі, і додаткові Resize не дають суттєвого приросту впевненості або якості рамки. Аналогічно для “cat” модель і так видає майже максимальні значення, тому простору для покращення по score майже немає.

Зміни найбільше торкнулися “remote”: у варіанті з TTA з’явилася додаткова рамка remote з близькою, але нижчою впевненістю (типу remote: 0.96 поряд із remote: 0.97). Це означає, що на одному з додаткових масштабів модель згенерувала ще одну правдоподібну гіпотезу для пульта, але ця рамка не була повністю пригнічена NMS. Причина в тому, що ці дві рамки можуть мати недостатньо велике перекриття (IoU нижче порогу iou_threshold=0.5), або їхні координати відрізняються так, що алгоритм не вважає їх дублікатами. У результаті отримуємо не чисте покращення, а змішаний ефект: з одного боку, TTA підвищує чутливість до дрібних об’єктів і може додавати корисні знахідки; з іншого боку, вона підвищує ймовірність появи зайвих повторних рамок (фактично дублювання одного й того ж об’єкта).

Отже, multi-scale TTA допомагає моделі стабільніше знаходити об’єкти різного масштабу (особливо дрібні), але водночас вимагає більш строгої постобробки, щоб прибрати повтори. Покращення проявилося саме як додаткова детекція (6 замість 5) і поява другої рамки для remote, тоді як для couch приросту по score не видно, бо базова детекція і так була достатньо впевненою. Це показує, що TTA корисна насамперед для складних або малих об’єктів, а не для великих і очевидних, і що критерій покращення тут краще

оцінювати не тільки по числу рамок, а по тому, чи з’являються нові коректні об’єкти без зайвих дублювань.

Наступні пункти покращення розберемо без змін у коді, тому що демо-ноутбук з GitHub призначений для інференсу (показу роботи вже навчених ваг): завантажується pretrained модель, подається одне зображення, і одразу будуються рамки. У такому режимі параметри моделі не змінюються, градієнти не рахуються, датасет і цикл навчання відсутні. Через це більшість серйозних покращень, які реально підвищують якість детекції, неможливо коректно реалізувати саме в демо: вони потребують або донавчання моделі, або зміни навчального датасету/розмітки, або зміни training pipeline і повторного отримання ваг. У демо можна лише змінювати постобробку та стратегії інференсу, але це не замінює покращення моделі, а інколи дає побічні ефекти на кшталт дублювання детекцій.

2. Покращення попередньої обробки даних і аугментацій під час навчання
Ідея: зробити так, щоб модель під час тренування бачила більше реальних варіацій зображень, які потім зустрінуться на тесті. Якщо під час навчання показувати тільки ідеальні картинки, модель гірше переноситься на інші камери, інше освітлення і інші масштаби об’єктів.
Що саме можна покращити:

- multi-scale training: під час навчання випадково змінювати розмір зображення перед подачею в модель (різні масштаби). Це допомагає краще знаходити об’єкти як великі, так і малі.
- випадкові геометричні перетворення: невеликі повороти, зсуви, обрізання (random crop), дзеркальне відображення. Це робить модель більш стійкою до того, що об’єкт може бути не по центру, частково обрізаний або під іншим кутом.

- фотометричні аугментації: зміна яскравості, контрасту, насиченості, легкий шум або розмиття. Це корисно, коли в реальних фото бувають тіні, засвітки, погана якість кадру.

Який очікуваний ефект: менше пропусків об’єктів, більш стабільні рамки, менша залежність від ідеального освітлення та масштабу.
Чому не робиться в демо: аугментації для тренування мають сенс лише тоді, коли після них модель донавчається і перебудовує ваги. У демо ваги фіксовані, тому сильні аугментації можуть навіть погіршити результат на конкретному зображенні.

3. Використання додаткових джерел даних і донавчання під потрібний домен
Ідея: pretrained DETR навчений на COCO, але реальні задачі часто відрізняються від COCO (інші ракурси, інші класи, інше середовище, інші типи об’єктів). Щоб модель працювала краще саме в потрібних умовах, її донавчають на даних, схожих на цільові.
Що саме можна зробити:

- зібрати невеликий власний набір зображень з потрібного домену і розмітити bounding boxes. Навіть кілька сотень якісно розмічених зображень можуть дати відчутний приріст.
- використати додаткові відкриті датасети зі схожими класами або умовами, поєднати їх з основним набором (важливо привести класи та формат розмітки до єдиного вигляду).
- застосувати fine-tuning: стартувати з pretrained ваг і донавчити на нових даних. Це швидше й дешевше, ніж навчати з нуля, і майже завжди дає кращий результат для конкретної задачі.

Який очікуваний ефект: підвищення точності саме на цільових зображеннях, менше хибних спрацьовувань, краща локалізація рамок в нестандартних сценах.
Чому не робиться в демо: потрібні розмічені дані, завантаження датасету,

training loop і збереження нових ваг. Демо цього не містить, воно розраховане на швидку демонстрацію інференсу.

4. Удосконалення моделі та інференсу: більш сильний DETR-варіант, краща постобробка, стабілізація детекцій

Ідея: базовий DETR з оригінальної реалізації простий, але має обмеження (особливо по швидкості збіжності навчання та по малих об’єктах). Тому на практиці часто беруть покращені архітектури або додають сильніші інференсні механізми.

Що саме можна зробити на рівні моделі:

- перейти на модифікації, які краще працюють у складних сценах (наприклад, Deformable DETR або інші сучасні DETR-похідні). Вони краще фокусуються на важливих ділянках зображення і частіше дають кращу локалізацію.
- використати сильніший backbone (наприклад, більш сучасні CNN або transformer-backbone). Це підвищує якість витягування ознак, особливо для дрібних деталей.

  Що саме можна зробити на рівні інференсу:
- test-time augmentation, multi-scale: проганяти зображення в кількох масштабах і об’єднувати результати. Це може додати детекції, які зриваються на одному масштабі.
- коректна постобробка: клас-специфічний NMS або інші методи подавлення дублювань, а також акуратний вибір порога впевненості. Це зменшує кількість повторних рамок і підвищує читабельність та стабільність фінального результату.

Який очікуваний ефект: краща якість на складних прикладах (особливо дрібні/частково перекриті об’єкти), менше дублікатів рамок, більш надійний результат при зміні масштабу.

Чому повністю не робиться в демо: заміна архітектури або backbone означає

інший код моделі і нові ваги, які треба отримати тренуванням. У демо можна показати лише інференсні покращення, але архітектурні та навчальні зміни належать до основного репозиторію з тренуванням.

Висновки: у ході роботи було обрано та запущено демо з GitHub для алгоритму DETR у середовищі Google Colab, перевірено коректність виконання всіх комірок і отримано результат у вигляді візуалізації детекцій на тестовому зображенні (класи, ймовірності та обмежувальні рамки). Код демо було розібрано по логічних частинах: підготовка середовища та залежностей, завантаження попередньо навчених ваг, підготовка зображення, виконання інференсу, післяобробка та побудова результатів.

Додатково продемонстровано один із практичних способів покращення якості інференсу без перенавчання моделі – test-time augmentation (multi-scale), тобто запуск моделі на кількох масштабах зображення та об'єднання результатів. Порівняння базового режиму та режиму з TTA показало, що кількість детекцій і набір знайдених об'єктів можуть змінюватися, а також можуть з'являтися додаткові рамки для складних або частково перекритих об'єктів. Це підтверджує, що результати детекції чутливі до попередньої обробки та стратегії інференсу, і що навіть прості зміни пайплайна можуть впливати на фінальну якість.

Таким чином, вимогу роботи щодо розуміння коду алгоритму, запуску та демонстрації запущеного демо з GitHub виконано, а також показано приклад обґрунтованого покращення результатів на рівні інференсу та післяобробки.

Посилання на GitHub: https://github.com/avokaskam/ml-lab6-detr-demo/tree/main