3. Fine-Tuning Model Process

1. Introduction

In [1], We train the Image Segmentation and Classification model from `resnet-50` backbone, but we re-write a whole model for it to adapt to our dataset keys. And in that case, we re-write all heads and even a whole model. Which complicated the problem.

Refer to forward method In our model, we receive following parameters :

```python
def forward(self,
            pixel_values: torch.Tensor,
            pixel_mask: torch.Tensor,
            class_ids: Optional[torch.Tensor] = None,
            segmentation_mask: Optional[torch.Tensor] = None,
            bboxes: Optional[torch.Tensor] = None,
            **kwargs,  # meta info (not used here)
            ) -> FishSegmentationModelOutput:
```

(1) Model Construction

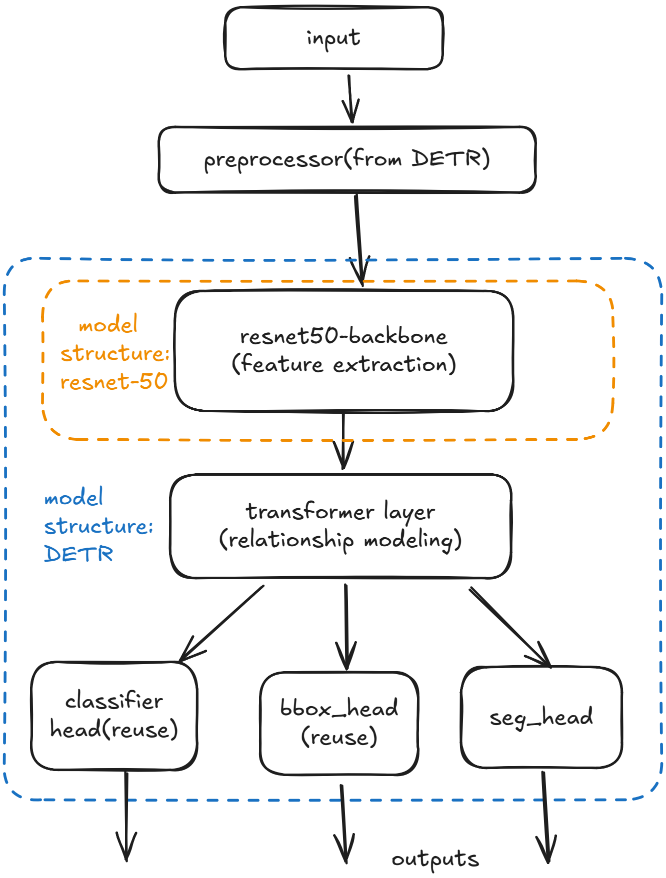
Actually, we can firstly simply modify `num_labels`,

```python
from transformers import DetrForObjectDetection

model = DetrForObjectDetection.from_pretrained(
    "facebook/detr-resnet-50-panoptic",
    num_labels=9,  # number of fish species
    ignore_mismatched_sizes=True,  # avoid mismatched weighting size error
)
# or use following (use num_labels in config)
config = DetrConfig(num_labels=9, )
model = DetrForObjectDetection.from_pretrained(
    "facebook/detr-resnet-50-panoptic", # number of fish species
    ignore_mismatched_sizes=True,
    config=config
)
```

Then jump to the source code of `DetrForObjectDetection` to see what it needs :

```python
class DetrForObjectDetection(DetrPreTrainedModel):
    def __init__(self, config: DetrConfig):...

    @auto_docstring
    def forward(
        self,
        pixel_values: torch.FloatTensor,
        pixel_mask: Optional[torch.LongTensor] = None,
        decoder_attention_mask: Optional[torch.FloatTensor] = None,
        encoder_outputs: Optional[torch.FloatTensor] = None,
        inputs_embeds: Optional[torch.FloatTensor] = None,
        decoder_inputs_embeds: Optional[torch.FloatTensor] = None,
        labels: Optional[list[dict]] = None,
        output_attentions: Optional[bool] = None,
        output_hidden_states: Optional[bool] = None,
        return_dict: Optional[bool] = None,
    ) → Union[tuple[torch.FloatTensor], DetrObjectDetectionOutput]:...
```

Compare to ResNet50-Strucutre-Model, actual DETR model has a transformer layer

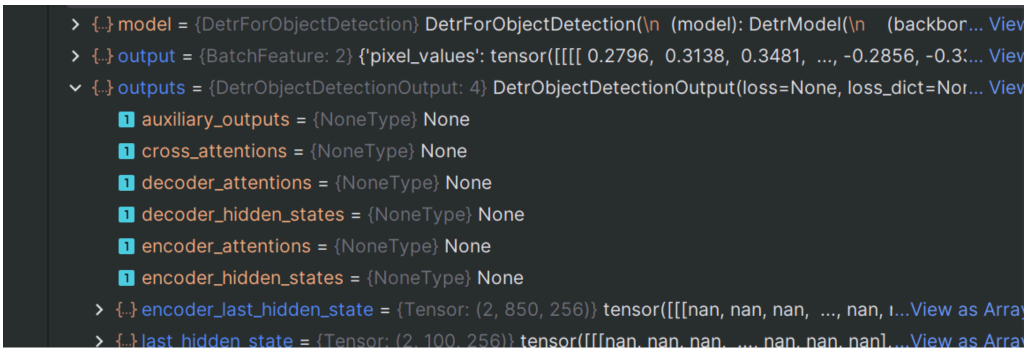


So if we just want to fine-tune, we **only need image as the input** :

```python
def test_model():
    # just use pretrained DETR model and fine-tune it
    model = DetrForObjectDetection.from_pretrained(
        "facebook/detr-resnet-50-panoptic",
        num_labels=9,  # number of fish species
        ignore_mismatched_sizes=True,
    )
    processor = DetrImageProcessor.from_pretrained("facebook/detr-resnet-50")

    url = "http://images.cocodataset.org/val2017/000000039769.jpg"
    image = Image.open(requests.get(url, stream=True).raw)
    inputs = processor(images=[image, image], return_tensors="pt")
    inputs.data.keys()  # Out[25]: dict_keys(['pixel_values', 'pixel_mask'])
    outputs = model(**inputs)
    outputs.keys()  # odict_keys(['logits', 'pred_boxes', 'last_hidden_state', 'encoder_last_hidden_state'])
    outputs["logits"].shape  # Out[22]: torch.Size([2, 100, 10])

    # we can also control the output by `output_attention` parameter and `output_hidden_states` parameter
    outputs2 = model(**inputs, output_attentions=True, output_hidden_states=True)
    outputs2.keys() # Out[23]: odict_keys(['logits', 'pred_boxes', 'last_hidden_state', 'decoder_hidden_states', 'decoder_attentions', 'cross_attentions', 'encoder_last_hidden_state', 'encoder_hidden_states', 'encoder_attentions'])

    results = image_processor.post_process_object_detection(outputs, target_sizes=torch.tensor([image.size[::-1]]), threshold=0.3)
```

We note that **there's no labels in the inputs**, this is **because of what processor does**.

Then the output should be like this :



(2) Re-implement of dataloader function

Our new data function will return raw image (not convert RGB) and also, `mask`, `class_name` and `class_id`, It's relatively simpler than what we did before :

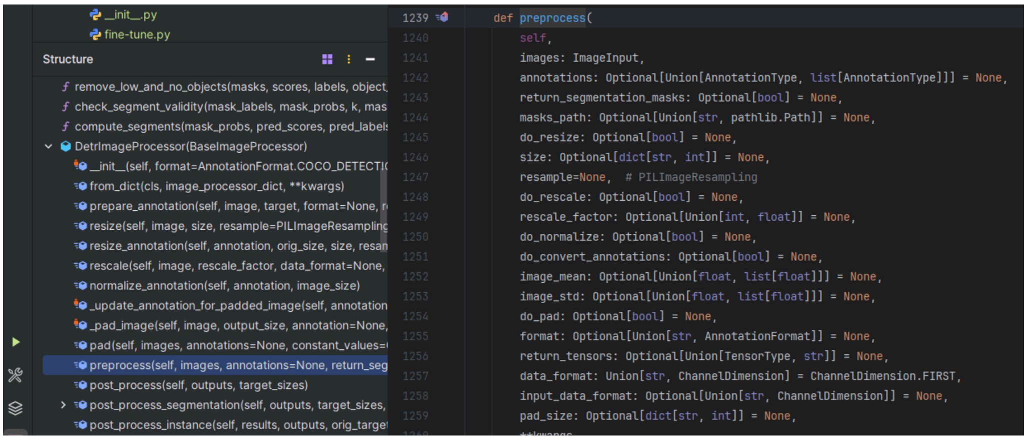
```python
class FishSegmentDataSet(Dataset):
    def __init__(self,
                 img_dirs: Union[str, List[str]],
                 mask_dirs: Union[str, List[str]]):
        super().__init__()
        self.img_dirs: List[str] = [img_dirs] if isinstance(img_dirs, str) else img_dirs
        self.mask_dirs: List[str] = [mask_dirs] if isinstance(mask_dirs, str) else mask_dirs
        if len(self.img_dirs) != len(self.mask_dirs):
            raise ValueError("Number of image and mask directories must match")

        self.items = []
        for idx, (img_dir, mask_dir) in enumerate(zip(self.img_dirs, self.mask_dirs)):
            img_paths = sorted(os.listdir(img_dir))
            mask_paths = sorted(os.listdir(mask_dir))
            class_name = os.path.basename(img_dir)  # assuming img_dir is like .../class_name
            if len(img_paths) != len(mask_paths):
                raise ValueError(f"Number of images and masks must match in {img_dir} and {mask_dir}")
            for img_path, mask_path in zip(img_paths, mask_paths):
                img_id = os.path.splitext(img_path)[0]
                mask_id = os.path.splitext(mask_path)[0]
                if img_id != mask_id:
                    raise ValueError(f"Image and mask file names must match: {img_id} vs {mask_id}")
                self.items.append(dict(img_path=img_path,
                                       mask_path=mask_path,
                                       class_name=class_name,
                                       class_id=idx))
    def __len__(self):
        return len(self.items)

    def __getitem__(self, item_id):
        record = self.items[item_id]
        img = Image.open(os.path.join(self.img_dirs[record['class_id']], record['img_path']))
        mask = Image.open(os.path.join(self.mask_dirs[record['class_id']], record['mask_path'])).convert("L")
        # here we also return image_id for tracking
        return dict(image_id = item_id,
                    image=img,
                    mask=torch.tensor(np.array(mask)).long(),
                    class_id=record['class_id'],
                    class_name=record['class_name'])
```

Then, we can use a simple `load_dataset()` method, just call:

```python
def load_dataset():
    dataset_url = "crowww/a-large-scale-fish-dataset"
    base_path = os.path.join(kagglehub.dataset_download(dataset_url), "Fish_Dataset",
"Fish_Dataset")
    class_names = get_fish_classes()
    img_dirs = [os.path.join(base_path, cls, cls) for cls in class_names]
    mask_dirs = [os.path.join(base_path, cls, cls + " GT") for cls in class_names]
    dataset = FishSegmentDataSet(img_dirs=img_dirs, mask_dirs=mask_dirs)
    return dataset
```

(3) Additional Parameters in Processor

For more precise parameters, we can refer to `DetrImageProcessor` [2], the parameters are listed in `preprocess` method of `DetrImageProcessor` .



In the above picture, preprocessor receive those parameters, in previous chapter, we only use `images` :

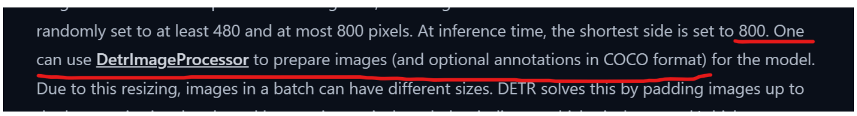
```python
    def preprocess(
        self,
        images: ImageInput,
        annotations: Optional[Union[AnnotationType, list[AnnotationType]]] = None,
        return_segmentation_masks: Optional[bool] = None,
        masks_path: Optional[Union[str, pathlib.Path]] = None,
        do_resize: Optional[bool] = None,
        size: Optional[dict[str, int]] = None,
        resample=None,  # PILImageResampling
        do_rescale: Optional[bool] = None,
        rescale_factor: Optional[Union[int, float]] = None,
        do_normalize: Optional[bool] = None,
        do_convert_annotations: Optional[bool] = None,
        image_mean: Optional[Union[float, list[float]]] = None,
        image_std: Optional[Union[float, list[float]]] = None,
        do_pad: Optional[bool] = None,
        format: Optional[Union[str, AnnotationFormat]] = None,
        return_tensors: Optional[Union[TensorType, str]] = None,
        data_format: Union[str, ChannelDimension] = ChannelDimension.FIRST,
        input_data_format: Optional[Union[str, ChannelDimension]] = None,
        pad_size: Optional[dict[str, int]] = None,
        **kwargs,
    ) -> BatchFeatur
```

Now the `annotations` parameter is what we need. That's what should we input :

```python
"""
annotations (`AnnotationType` or `list[AnnotationType]`, *optional*):
        List of annotations associated with the image or batch of images. If annotation is for object
        detection, the annotations should be a dictionary with the following keys:
        - "image_id" (`int`): The image id.
        - "annotations" (`list[Dict]`): List of annotations for an image. Each annotation should be a
          dictionary. An image can have no annotations, in which case the list should be empty.
        If annotation is for segmentation, the annotations should be a dictionary with the following keys:
        - "image_id" (`int`): The image id.
        - "segments_info" (`list[Dict]`): List of segments for an image. Each segment should be a dictionary.
          An image can have no segments, in which case the list should be empty.
        - "file_name" (`str`): The file name of the image.
"""
```

1) COCO-format annotations

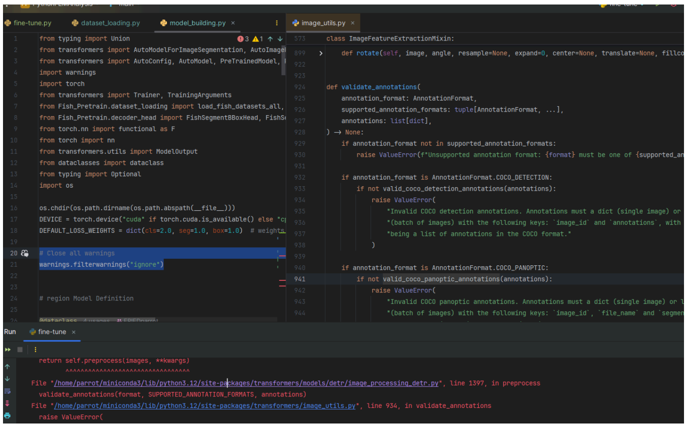
The DETR model use **standard COCO dataset format** for annotations[3], which is


randomly set to at least 480 and at most 800 pixels. At inference time, the shortest side is set to 800. One
can use DetrImageProcessor to prepare images (and optional annotations in COCO format) for the model.
Due to this resizing, images in a batch can have different sizes. DETR solves this by padding images up to

Standard Coco dataset format annotations are given as :

```python
# Construct annotations for object detection
"annotations": [
    {
        "id": 0,
        "image_id": 0,
        "category_id": 2,
        "bbox": [
            45,
            2,
            85,
            85
        ],
        "area": 7225,
        "segmentation": [],
        "iscrowd": 0
    },
    {
        "id": 1,
        "image_id": 0,
        "category_id": 2,
        "bbox": [
            324,
            29,
            72,
            81
        ],
        "area": 5832,
        "segmentation": [],
        "iscrowd": 0
    }
]
```

But we have a easier way to find the things out, we can trace back and jump to `valid_coco_detection_annotations` and `valid_coco_panoptic_annotations` to check how it really is :



The **source of checking coco-format functions are pasted as follows** :

Python

```python
# **Detection-only model** → ignores segmentation
def is_valid_annotation_coco_detection(annotation: dict[str, Union[list, tuple]]) -> bool:
    if (
        isinstance(annotation, dict)
        and "image_id" in annotation
        and "annotations" in annotation
        and isinstance(annotation["annotations"], (list, tuple))
        and (
            # an image can have no annotations
            len(annotation["annotations"]) == 0 or isinstance(annotation["annotations"][0], dict)
        )
    ):
        return True
    return False

def is_valid_annotation_coco_panoptic(annotation: dict[str, Union[list, tuple]]) -> bool:
    if (
        isinstance(annotation, dict)
        and "image_id" in annotation
        and "segments_info" in annotation
        and "file_name" in annotation
        and isinstance(annotation["segments_info"], (list, tuple))
        and (
            # an image can have no segments
            len(annotation["segments_info"]) == 0 or isinstance(annotation["segments_info"][0], dict)
        )
    ):
        return True
    return False
```

2) How to use in our Model

Our goal is to predict both class, bounding box and binary segmentation mask, but in Processor **We use Object detection annotation format**, so **the processor information** includes **classification and bounding boxes**.

This is because Processor just define what data to keep, but **actually the Model defines what loss to calculate**, so for 1-object with mask prediction, we can use `detection processor + segmentation model`:

✓ **Model Inputs**

- `DetrForObjectDetection`
  - Only looks at `"annotations"` with `bbox`, `category_id`, `area`, `iscrowd`.
  - Ignores any `segmentation` field.
  - Loss = classification + box (L1 + GIoU).
- `DetrForSegmentation` (`facebook/detr-resnet-50-panoptic`):
  - Extends the object detection setup with a **mask head**.

  - When you call the model with `labels` (from processor), it expects:
    - `labels["class_labels"]` (categories),
    - `labels["boxes"]` (bboxes),
    - `labels["mask_labels"]` **(segmentation masks)**.
  - Loss = classification + box + segmentation loss.

We also need to set `return_segmentation_masks` to `True`

```python
"""
return_segmentation_masks (`bool`, *optional*, defaults to self.return_segmentation_masks):
    Whether to return segmentation masks.
"""
```

Note there's another important parameter `format`, which is defined as :

```python
from transformers.image_utils import AnnotationFormat
class AnnotationFormat(ExplicitEnum):
    COCO_DETECTION = "coco_detection"
    COCO_PANOPTIC = "coco_panoptic"
```

Also, note the `segmentation` should be `formated` into **COCO detection style (which is [x1,y1, x2, y2, ...]).** So when we use `numpy` to input, we got `Exception: input type is not supported.`

Firstly, we use a coco sample for test

```python
"annotations": [
    {
        "image_id": torch.tensor([f["class_id"]]),
        "category_id": torch.tensor([f["class_id"]]),
        "bbox": torch.tensor(bbox),   # only gives bbox and area
        "area": torch.tensor([area]),
        # just make a square for test
        "segmentation": [[bbox[0], bbox[1], bbox[0] + bbox[2], bbox[1], bbox[0] + bbox[2], bbox[1] + bbox[3], bbox[0], bbox[1] + bbox[3]]],  # COCO polygon format
        "iscrowd": torch.tensor([0]),
    }
]
```

The return result shape is :

```python
{
  "pixel_values": Tensor[B, 3, H, W],      # normalized/resized images
  "pixel_mask": Tensor[B, H, W],           # padding mask (1 = valid, 0 = padded)
  "labels": [                              # list of dicts, one per image
      {
          "size": Tensor[2],               # processed (resized) image size
          "orig_size": Tensor[2],          # original H, W
          "image_id": Tensor[1],           # image id
          "class_labels": Tensor[num_obj], # categories for each object
          "boxes": Tensor[num_obj, 4],     # normalized boxes in cxcywh format
          "area": Tensor[num_obj],         # object area
          "iscrowd": Tensor[num_obj],      # crowd flags
          "masks": Tensor[num_obj, H, W],  # binary masks aligned to pixel_values
      },
      ...
  ]
}
```

We prepare to reuse `resize_mask` function in [4] add `masks` after processing to **simulate the calculation process**

So firstly, we test the shape of output :

```python
for batch in dataloader:
    print(batch)
    print(batch.keys())
    print(batch['labels'][0].keys())
    print(batch.get('labels')[0]["masks"].shape)  # torch.Size([1, 800, 1060])
```

3) Full Implement of data loader

> ⚠ **sort directory name after**
>
> `os.listdir()` in Python **does not sort folders**
> `os.listdir()` **in Python** does not sort folders or files automatically**.
> It returns the list of filenames in arbitrary order, which depends on the underlying filesystem and OS implementation.
> So, the order is not guaranteed and should be considered "random"!

We get images from dataloader,

Python

```python
class FishCollator:  # (DataCollator)
    def __init__(self, processor: DetrImageProcessor):
        self.processor = processor

    def __call__(self, features: List[Dict[str, Any]]) -> BatchFeature:
        images = [f["image"] for f in features]
        annotations = []

        # we may have multiple objects in one image, but here in dataset we use only one object per image
        for f in features:
            # prepare annotations from masks
            bbox = _compute_bbox_from_mask(np.array(f["mask"]))
            area = int(torch.prod(torch.tensor(bbox[2:])))
            ann: AnnotationType = {
                "image_id": torch.tensor(f["image_id"]),
                "annotations": [
                    {
                        "image_id": torch.tensor(f["image_id"]),  # not set to class_id, It breaks the matching logic inside the loss computation (Hungarian matching).
                        "category_id": torch.tensor(f["class_id"]),
                        "bbox": torch.tensor(bbox),  # only gives bbox and area
                        "area": torch.tensor(area),
                        # for COCO polygon format,
                        # "segmentation": [[bbox[0], bbox[1], bbox[0] + bbox[2], bbox[1], bbox[0] + bbox[2], bbox[1] + bbox[3], bbox[0], bbox[1] + bbox[3]]],
                        "iscrowd": torch.tensor([0]),
                    }
                ]
            }
            annotations.append(ann)
        # it is a dict, but here we use batch feature for convenience
        out = self.processor(images,
                             annotations=annotations,
                             return_segmentation_masks=False, # no mask info
                             format=AnnotationFormat.COCO_DETECTION,
                             return_tensors="pt")
        # since out now have no masks, we need to add them back
        masks =  [f["mask"] for f in features]

        for i, mask in enumerate(masks):
            # resize mask to the size of out['labels'][i]['size']
            target_size = tuple(out['labels'][i]['size'])
            mask_resize = resize_mask(mask, size=target_size)
            out['labels'][i]["masks"] = torch.tensor(mask_resize, dtype=torch.float32).unsqueeze(0)  # add channel dim
        return out
```

Hint

At first I didn't discover that code completion and accidentally made `image_id :`

> `f["class_id"]`, This is a very critical issue. This will affects Hungarian Matching

Here is an reference :

🧩 1️⃣ How `image_id` affects Hungarian Matching

DETR's training loss is built around **set-based bipartite matching** — this is what makes DETR unique.

For every **image**, it:

1. Takes the **model predictions** (e.g., 100 queries per image → 100 predicted boxes).
2. Takes the **ground-truth annotations** for that same image.
3. Runs a **Hungarian algorithm** to find the *best matching* between predicted boxes and real boxes — minimizing a cost that combines classification, bounding box, and mask loss.

Then, it computes the losses **per image**.

So actually the image

In that case,

```python
print(batch.get('labels')[0]["masks"].shape)  # it also returns  torch.Size([1, 800, 1060])
```

Then we can test the output by using following :

```python
print(model(**batch))      # also, we can notice "pred_masks" is in the prediction, so the mask loss is correctly calculated
```

2. Train the fine-tune pretrained model

(1) Train Code Implement

After implementing the above data collator code, the Process of fine-tuning model on Fish Dataset would become much simple than before, **all the backbones, transformers, heads, and loss functions** are reused :

> **Note**
>
> Be careful of `remove_unused_column` parameter, It should be set as `False` in order to keep the data transferred from the `Collator`, or it will lose columns.

Python

```python
def train_model():
    # just use pretrained DETR model and fine-tune it
    config = DetrConfig(
        model_name="Fish_Segmentation_Model_Fine_Tuning_DETR",
        num_labels=9
    )
    model = DetrForSegmentation.from_pretrained(
        "facebook/detr-resnet-50-panoptic",  # number of fish species
        ignore_mismatched_sizes=True,
        config=config,
    )
    processor = DetrImageProcessor.from_pretrained("facebook/detr-resnet-50")

    train_dataset, val_dataset = load_dataset(train_size=0.8, seed=42)
    collator = FishCollator(processor=processor)

    training_args = TrainingArguments(
        output_dir="./model",
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        num_train_epochs=20,
        logging_steps=10,
        save_steps=50,
        save_total_limit=2,
        remove_unused_columns=False,
        eval_strategy="steps",
        learning_rate=5e-5,
        weight_decay=0.01,
        warmup_steps=100,
    )

    trainer = Trainer(
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        data_collator=collator,
    )

    # trainer.train()
    trainer.save_model('./models/final_model')
    processor.save_pretrained('./models/final_model')
    config.save_pretrained('./models/final_model')

    repo_id = "FriedParrot/fish-segmentation-simple"
    # push model to hub
    backup_model_to_hub(repo_id, config, model, processor)

    print("model training finished")
```

Training Parameter Settings

> Since this is a very large pretrained model, 1e-4 learning rate is too large for this model to converge. So we can use smaller learning rate (5e-5) and slightly increase the epoch num (to 20). And It's better for model training.

Here how to implement `backup_model_to_hub` has been given in [4-1].

(2) Query-based

And example for how to evaluate model are also given in this chapter. But note for `detr` model, its return data includes :

Plain text

```
'loss', 'loss_dict', 'logits', 'pred_boxes', 'pred_masks', 'last_hidden_state', 'encoder_last_hidden_state'
```

Since the DETR infers multiple predictions (N as is said in essay), Where the `logits` is of following shape (default `num_queries` is `100`) :

Python

```python
(batch_size, num_queries, num_classes + 1)
```

so actually, when we get `logits` from the network output, we got an output shape of `(batch_size, 100, 10)`

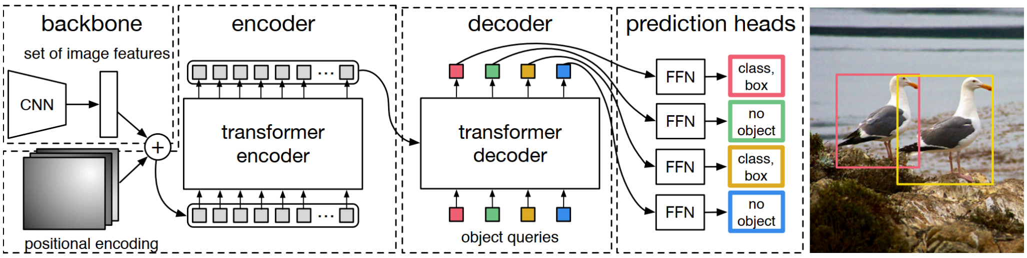
We need to know more about the output information of DERT. So for a brief view of the prediction principles of DETR network, we can refer to [5].

DETR use a direct set prediction approach to bypass the surrogate tasks. Which adopts the sequence-prediction effective transformers architecture. It simplifies the detection pipeline by dropping multiple hand-designed components that encode prior knowledge.

For multi-object detectors, DETR use Bipartite Matching Loss[6]. with following loss function :

$$\hat{\sigma} = \arg\min_{\sigma \in \Sigma} \sum_{i=1}^{n} \mathcal{L}_{match}(y_i, \hat{y}_{\sigma(i)}) \tag{2.2.1}$$

The ensemble structure of a DETR is as follows :



So the result is N predictions, and for `facebook/detr-resnet-50-panoptic` model, it also has its built-in mask head to predict the mask of each prediction.

Also, to predict the `backgroud`, we needn't assign the background by ourselves. And when DETR predicts classes, it **automatically appends** a special **"no-object" (background)** class.

so we use following config :

```python
config = DetrConfig(
    model_name="Fish_Segmentation_Model_Fine_Tuning_DETR",
    num_labels=9  # use number of Fish species (no need to +1 for background in DETR segmentation)
)
```

Then DETR internally reserves one more class :

> total classes = `num_classes + 1`

(3) Problems in loading model

When we load model, we may encounter following problem :

```shell
File "/lib/python3.12/site-packages/safetensors/torch.py", line 381, in load_file with safe_open(filename, framework="pt", device=device) as f: ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ safetensors_rust.SafetensorError: Error while deserializing header: header too small
```

This is because hugging face will cache the model weights, and it may load failed. So we can execute like following command to clear cache.

```shell
rm -rf ~/.cache/huggingface/hub
```

3. Evaluation

Evaluation Code are given as follows :

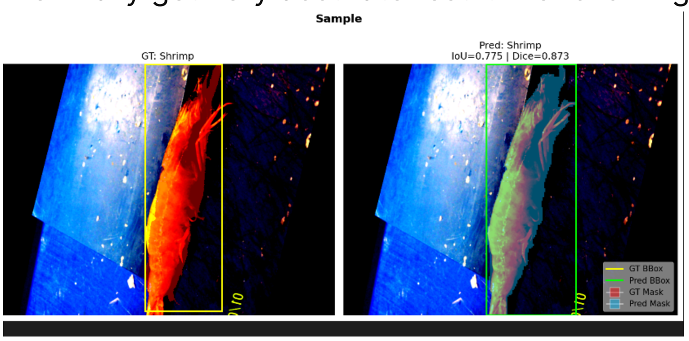
What to care :

- Since the input mask is

```python
def evaluate_model(
        model, processor, batch_size: int = 1, max_vis: int = 2, save_dir: str = None
    ):
    _, test_set = load_dataset(train_size=0.95)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False, num_workers=2,
                             collate_fn=FishCollator(processor), pin_memory=True if torch.cuda.is_available() else False)
    classes = get_fish_classes()
    model.eval()
    model.to(device)

    with torch.no_grad():
        show_num = 0
        correct = 0
        for index, batch in enumerate(test_loader):
            # since there is nested structure in inputs, only pin_memory for tensors is not enough
            inputs = move_batch_to_device(batch, device)
            outputs = model(**inputs)

            images = inputs['pixel_values']
            labels = [label for label in inputs['labels']]

            # ========   predict result (written by GPT) ===============
            class_pred = outputs["logits"].argmax(-1)  # (B, num_queries)
            pred_masks = outputs["pred_masks"].sigmoid()  # (B, num_queries, H, W)
            pred_bboxes = outputs["pred_boxes"]  # (B, num_queries, 4)
            if show_num >= max_vis:
                print("calculating accuracy ...", "batch: ", index, "/", len(test_loader))

            batch_size = images.shape[0]   # the bbox is predicted from mask
            # calculate the classification accuracy -> gt : ground truth
            for i in range(batch_size):
                img = images[i]  # shape [3, H, W]
                label = labels[i] # predicted labels
                # --- Handle predicted class, masks and bboxes (they should all be 1)---
                class_ids = label['class_labels']  # input class ids
                masks = label['masks']  # [B, 800, 1060]
                bboxes = label['boxes']  # NOTE : the bbox is normalized
                if len(class_ids) > 1:
                    print(f"Warning: Multiple classes detected in sample {i}, using the first.")
                if len(masks) > 1:
                    print(f"Warning: Multiple masks detected in sample {i}, using the first.")
                if len(bboxes) > 1:
                    print(f"Warning: Multiple boxes detected in sample {i}, using the first.")

                gt_label = class_ids[0] if len(class_ids) > 0 else None
                gt_mask = masks[0] if len(masks) > 0 else None
```

```
                gt_box = bboxes[0] if len(bboxes) > 0 else None

                # --- Prediction Extraction ---
                valid_idx = class_pred[i] != model.config.num_labels
                filtered_classes = class_pred[i][valid_idx]
                filtered_masks = (pred_masks[i][valid_idx] > 0.5).long()   # for sigmoid function, we need to compare to 0.5
                filtered_boxes = pred_bboxes[i][valid_idx]

                # collate predicted results
                pred_label = filtered_classes[0] if filtered_classes.nelement() > 0 else None
                if pred_label == gt_label:
                    correct += 1

                if show_num >= max_vis:
                    continue
                else:
                    show_num += 1
                pred_mask = filtered_masks[0] if filtered_masks.nelement() > 0 else None
                pred_box = filtered_boxes[0] if filtered_boxes.nelement() > 0 else None
                # get label names
                gt_label_name = classes[gt_label] if gt_label is not None else "N/A"
                pred_label_name = classes[pred_label] if pred_label else "N/A"
                visualize_sample_comparison(
                    img.cpu(),
                    gt_mask.cpu(), pred_mask.cpu(),
                    gt_label_name, pred_label_name,
                    gt_box.cpu(), pred_box.cpu(),
                    save_path=save_dir,
                    bbox_mode="center",
                )
        # compute accuracy
        total = len(test_set)
        accuracy = correct / total # total number of samples in the test set
        print(f"Classification Accuracy on Test Set: {accuracy*100:.2f}% ({correct}/{total})")
```

We finally get very accurate result like following images :



---

1. 1. Entrance of Hugging Face and pre-trained model ↩
2. https://huggingface.co/docs/transformers/v4.56.2/en/model_doc/detr ↩

3. https://www.v7labs.com/blog/coco-dataset-guide ↩

4. 1. Entrance of Hugging Face and pre-trained model ↩ ↩

5. End-to-End Object Detection with Transformers ↩

6. Bipartite Matching Loss ↩