1. Entrance of Hugging Face and pre-trained model

1. Introduction

(1) Installation

Many useful examples can be found at [1].
For torch installation

```
pip install torch==2.8.0 torchvision==0.23.0 torchaudio==2.8.0 --index-url
https://download.pytorch.org/whl/cu126
```

Mostly famous models in hugging face are used for NLP and image processing.

```
pip install transformers datasets accelerate evaluate safetensors sentencepiece
huggingface_hub diffusers timm
```

We can set the model source by following command to set mirror to `hf-mirror.com` :

```
echo 'export HF_ENDPOINT="https://hf-mirror.com"' >> ~/.bashrc
source ~/.bashrc
echo $HF_ENDPOINT
```

or `MaxRetryError("HTTPSConnectionPool(host='huggingface.co', port=443` will happens.

For `pycharm` , the **environmental variable must be added in run configuration** (or use in `.env` file and then specify it in run configuration):

```
HF_ENDPOINT = https://hf-mirror.com
```

then give it a check :

```
import os
print(os.getenv("HF_ENDPOINT"))  #  https://hf-mirror.com
```

For login and access account content, we can use :

```
hf auth login
# or in jupyter notebook
from huggingface_hub import notebook_login
notebook_login()
```

2. Pretrained Model

(1) Introduction on base classes

`transformers` are the most important part of hugging face, for more about that, see [2]. Used for fine-tune, extend, and use models.

Main features of transformers includes :

- Pipeline: Simple and optimized **inference class for many machine learning tasks** like text generation, image segmentation, automatic speech recognition, document question answering, and more.
- Trainer: A comprehensive trainer that supports features such as `mixed precision`, `torch.compile`, and `FlashAttention` for training and distributed training for `PyTorch` models.
- generate: Fast text generation with large language models (LLMs) and vision language models (VLMs), including support for streaming and multiple decoding strategies.

For all the basic classes, the following method are all shared, which is `instantiated from pretrained instances, saved locally, and shared on the Hub` :

- `from_pretrained()` lets you **instantiate a model, configuration, and preprocessing class** from a pretrained version either provided by the library itself (the supported models can be found on the Model Hub) or stored locally (or on a server) by the user.
- `save_pretrained()` lets you **save a model, configuration, and preprocessing class** locally so that it can be reloaded using `from_pretrained()`.
- `push_to_hub()` lets you **share a model, configuration, and a preprocessing class to the Hub**, so it is easily accessible to everyone.

Hint

The 3 standard classes require to use each model are `config, model, preprocessing`, Each pretrained model all inherits from those 3 base classes.

```python
# config class
from transformers import PretrainedConfig
# model class
from transformers import PreTrainedModel
#  preprocessing class : preprocessor is A class for converting raw inputs (text, images, audio, multimodal) into numerical inputs to the model. For example, stl_to_sdf can convert a stl to sdf tensor as input for model.
from transformers import PreTrainedTokenizer, AutoImageProcessor, AutoFeatureExtractor, AutoProcessor
```

Note

In actual case, for a specific *model class*, It's just the raw hidden states of the model.

We should use appropriate **model head (on top of the base transformer)** such as `LlamaForCausalLM` or `LlamaForSequenceClassification` (for `LlamaModel`)

All `config, model and preprocessor` have `from_pretrained()` method, this will load the weights and configuration file from the Hub into the model and preprocessor class.

(2) Load pretrained model

Firstly, we introduce `AutoClasses` [3], which contains `AutoConfig, AutoModel` and `AutoTokenizer`. (This is very important since it automatically infers the most suitable class).

```python
model = AutoModel.from_pretrained("google-bert/bert-base-cased")  # will create a model that is an instance of BertModel
```

```python
# all configurations of the model can be loaded as follows :
print(model.config)
```

Each of the auto classes has a method to be extended with your custom classes.
For example, For adding a config for the new model, we can custom define a config class like following :

```python
from transformers import AutoConfig, AutoModel, PretrainedConfig, PretrainedModel

class NewModelConfig(PretrainedConfig):
    model_type = "new-model"  # This must match the registration key
    def __init__(self, my_param=42, **kwargs):
        super().__init__(**kwargs)
        self.my_param = my_param

class NewModel(PretrainedModel):
    pass

AutoConfig.register("new-model", NewModelConfig)  # use register key to register new model
and config with transformers library.
AutoModel.register(NewModelConfig, NewModel)
```

Then we comes to **how to load model and make inputs** :

- `device_map="auto"` automatically allocates the model weights to your fastest device first.
- `dtype="auto"` directly initializes the model weights in the data type they're stored in, which can help avoid loading the weights twice (PyTorch loads weights in `torch.float32` by default).

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("meta-llama/Llama-2-7b-hf", dtype="auto",
device_map="auto")
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")

# Then, model and tokenizer are reday for inference and training
model_inputs = tokenizer(["The secret to baking "], return_tensors="pt").to(model.device)
generated_ids = model.generate(**model_inputs, max_length=30)
tokenizer.batch_decode(generated_ids)[0] '<s> The secret to baking a good cake is 100% in
the preparation. There are so many recipes out there,'
```

For example, we here provide another option is to use `AutoModelForSeq2SeqLM, AutoTokenizer` for decode at the example of (2) Direct Use of pretrained model by pipeline.

for inference, we can use `model.generate` , and then use `tokenizer.batch_decode(generated_ids)[0]` or `tokenizer.decode` for result.

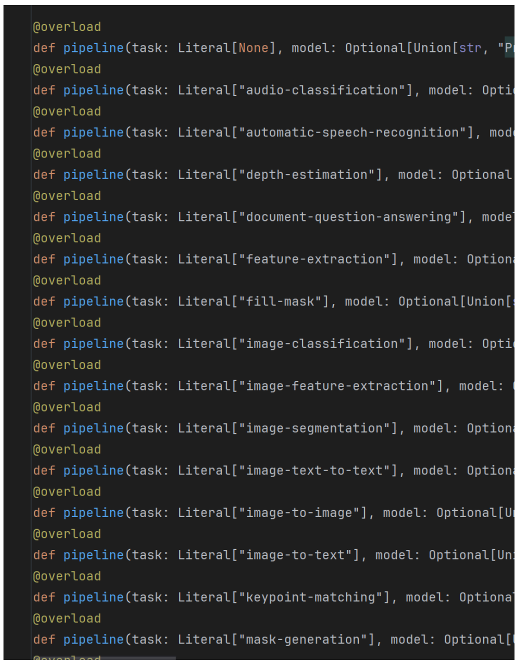
```python
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer

# load a translation model from the Hugging Face model hub
model = AutoModelForSeq2SeqLM.from_pretrained("JamesH/Translation_en_to_fr_project",
dtype="auto")
tokenizer = AutoTokenizer.from_pretrained("JamesH/Translation_en_to_fr_project")

# use model for inference
input_str = "Comment allez-vous?"
inputs = tokenizer(input_str, return_tensors="pt")
```

```python
outputs = model.generate(**inputs, max_length=100)

trans = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(trans)
```

(3) Usage of pretrained model by pipeline

The Pipeline class is the most convenient way to inference with a pretrained model. It supports many tasks such as text generation, image segmentation, automatic speech recognition, document question answering, and more.

For `transformer.pipeline` , it provides a lot of functions, and just using "task", we can do many tasks from pre-trained model.



```python
@overload
def pipeline(task: Literal[None], model: Optional[Union[str, "P
@overload
def pipeline(task: Literal["audio-classification"], model: Opti
@overload
def pipeline(task: Literal["automatic-speech-recognition"], mod
@overload
def pipeline(task: Literal["depth-estimation"], model: Optional
@overload
def pipeline(task: Literal["document-question-answering"], mode
@overload
def pipeline(task: Literal["feature-extraction"], model: Option
@overload
def pipeline(task: Literal["fill-mask"], model: Optional[Union[
@overload
def pipeline(task: Literal["image-classification"], model: Opti
@overload
def pipeline(task: Literal["image-feature-extraction"], model:
@overload
def pipeline(task: Literal["image-segmentation"], model: Option
@overload
def pipeline(task: Literal["image-text-to-text"], model: Option
@overload
def pipeline(task: Literal["image-to-image"], model: Optional[U
@overload
def pipeline(task: Literal["image-to-text"], model: Optional[Un
@overload
def pipeline(task: Literal["keypoint-matching"], model: Optiona
@overload
def pipeline(task: Literal["mask-generation"], model: Optional[
```

We use only 5 line for using a pretrained model (using `Helsinki-NLP/opus-mt-en-fr` as standard `en-fr` model) :

```python
from transformers import pipeline

pipe = pipeline("translation", model="Helsinki-NLP/opus-mt-en-fr")
input_str = "Hello, my dog is cute"
print(pipe(input_str)[0].get('translation_text'))
# [{'translation_text': 'Bonjour, mon chien est mignon.'}]

# This is actually an fr-en translator !!!
pipe2 = pipeline("translation", model="JamesH/Translation_en_to_fr_project")
print(pipe2("Bonjour, mon chien est mignon.")[0].get('translation_text'))
# [{'translation_text': 'Hello, my dog is cute.'}]
```

For a simple figure segmentation task, we can use :

```python
from transformers import pipeline
import warnings
# Close all warnings
warnings.filterwarnings("ignore")
pipeline = pipeline("image-segmentation",
                    model="facebook/detr-resnet-50-panoptic",
                    device_map="auto", dtype="auto",
                      use_fast=True)

# not contain "huggingface.com"!
img_url = "https://hf-mirror.com/datasets/Narsil/image_dummy/raw/main/parrots.png"
result = pipeline(img_url)
# [{'score': 0.999439, 'label': 'bird', 'mask': <PIL.Image.Image image mode=L size=768x512
at 0x72FB0C12CE30>}, {'score': 0.998788, 'label': 'bird', 'mask': <PIL.Image.Image image
mode=L size=768x512 at 0x72FB0C19D4F0>}]
print(result)
```

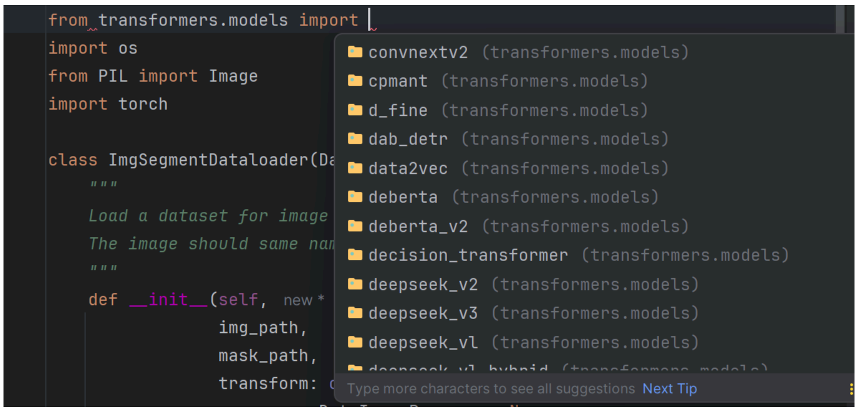
For very quick inference, we can use `InferenceClient` to just for infer using model in hub conveniently.

```
import os
from huggingface_hub import InferenceClient
client = InferenceClient(
    provider="hf-inference",
    api_key=os.environ["HF_TOKEN"],
)
output = client.image_classification("cats.jpg", model="Falconsai/nsfw_image_detection")
```

(4) Cross-Architecture usage for Models and Processors

1. Models

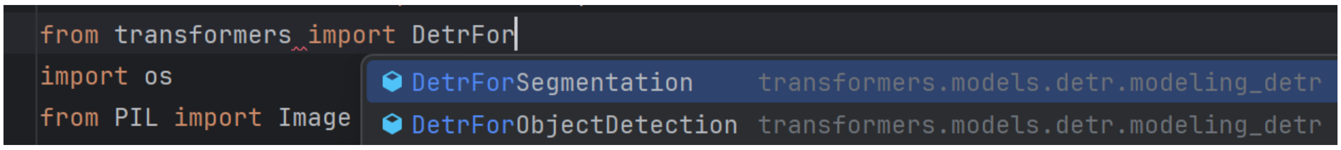
We can use any model and use correct Model class.
In `AutoModel`, it includes very extensive models, which can be accessed by `transformers.models`, and it includes extensive models :



```
from transformers.models import 
import os
from PIL import Image
import torch

class ImgSegmentDataloader(Da
    """
    Load a dataset for image
    The image should same na
    """
    def __init__(self,  new *
                 img_path,
                 mask_path,
                 transform:
```

```
convnextv2  (transformers.models)
cpmant  (transformers.models)
d_fine  (transformers.models)
dab_detr  (transformers.models)
data2vec  (transformers.models)
deberta  (transformers.models)
deberta_v2  (transformers.models)
decision_transformer  (transformers.models)
deepseek_v2  (transformers.models)
deepseek_v3  (transformers.models)
deepseek_vl  (transformers.models)
```

Often, we can first load model use auto and check its type. then we can just look for the task it can handle, then it can be used across following task (Segmentation, Object Detection).

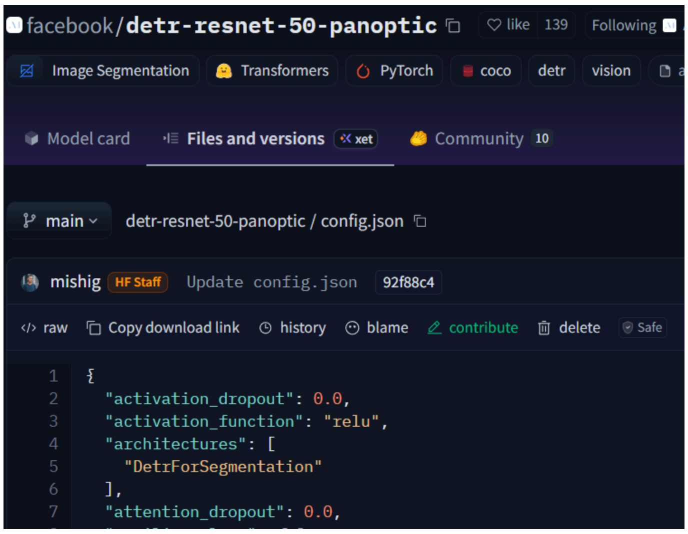


```
from transformers import DetrFor
import os
from PIL import Image
```

```
DetrForSegmentation        transformers.models.detr.modeling_detr
DetrForObjectDetection     transformers.models.detr.modeling_detr
```

```
detr = AutoModel.from_pretrained("facebook/detr-resnet-50-panoptic")

# use Model for different task
detr_detection = AutoModelForObjectDetection.from_pretrained("facebook/detr-resnet-50")
detr_segmentation = AutoModelForImageSegmentation.from_pretrained("facebook/detr-resnet-50-
panoptic")
```

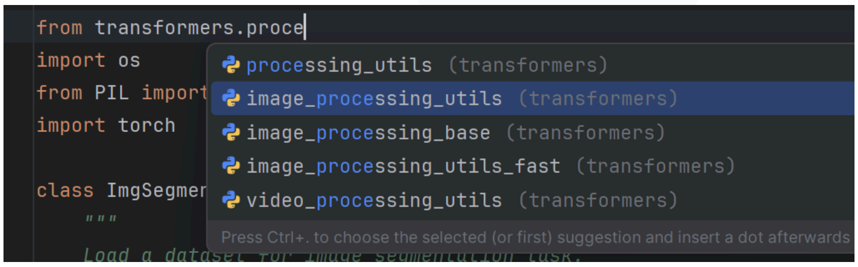
But for precise architecture type, see `configs.json` under the project.



```
{
  "activation_dropout": 0.0,
  "activation_function": "relu",
  "architectures": [
    "DetrForSegmentation"
  ],
  "attention_dropout": 0.0,
```

2. Processor

Similarly, processor are located in `...processing_utils`, but it may return any, just from models folder.



```
from transformers.proce
import os
from PIL import
import torch

class ImgSegmen
    """
    Load a dataset for image segmentation task.
```

processing_utils (transformers)
image_processing_utils (transformers)
image_processing_base (transformers)
image_processing_utils_fast (transformers)
video_processing_utils (transformers)
Press Ctrl+. to choose the selected (or first) suggestion and insert a dot afterwards

All processor are inherited from `BaseProcessor`, Most important includes `BaseImageProcessor`, `BaseVideoProcessor`. And

For image or video, the processor are not compatible :

```
AutoProcessor() # if even not know which processor type is
pr = AutoVideoProcessor.from_pretrained("facebook/detr-resnet-50-panoptic") # it's a image processor, so it will raise error
```

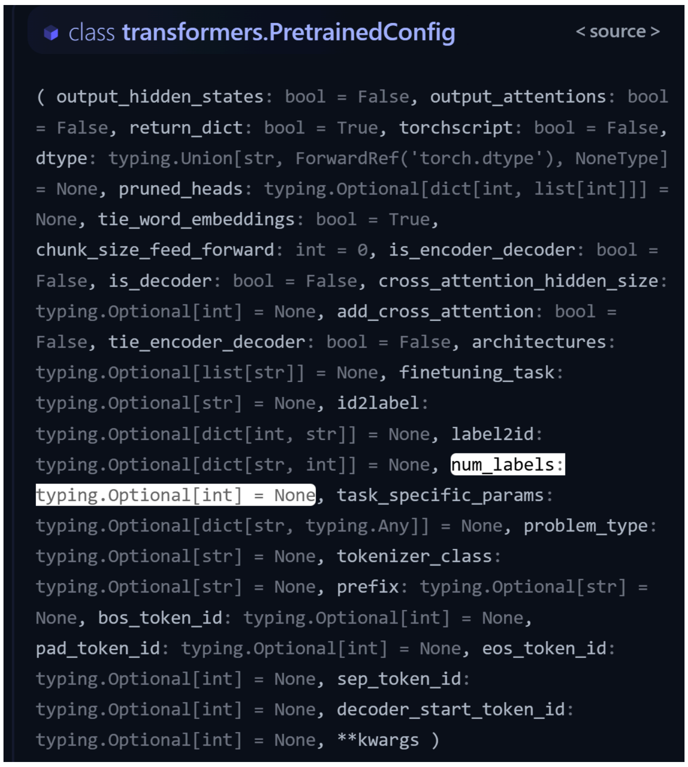
But **for the different task, the model and processor are usually universal** (for example, in `ImageProcessor`, we can use for image classification)

```
# recommended : auto processor
processor = AutoImageProcessor.from_pretrained("facebook/detr-resnet-50-panoptic", use_fast=True)

# can specify
processor = DetrImageProcessor.from_pretrained("facebook/detr-resnet-50-panoptic")
model = DetrForSegmentation.from_pretrained("facebook/detr-resnet-50-panoptic")
```

(5) PretrainedConfig Class

Each config has a set of attributes (like `num_labels`, `id2label`, `label2id`) stored as instance variables. when calling `config.to_dict()`, Hugging Face converts only the **“official” fields** that are explicitly registered in the config class. Some attributes are defined dynamically or inherited, and they don’t always get serialized.

In actual training process, this can used for modify some configurations for pipeline pretrained model, but no-use if we want to fine-tune or something.



```
( output_hidden_states: bool = False, output_attentions: bool
= False, return_dict: bool = True, torchscript: bool = False,
dtype: typing.Union[str, ForwardRef('torch.dtype'), NoneType]
= None, pruned_heads: typing.Optional[dict[int, list[int]]] =
None, tie_word_embeddings: bool = True,
chunk_size_feed_forward: int = 0, is_encoder_decoder: bool =
False, is_decoder: bool = False, cross_attention_hidden_size:
typing.Optional[int] = None, add_cross_attention: bool =
False, tie_encoder_decoder: bool = False, architectures:
typing.Optional[list[str]] = None, finetuning_task:
typing.Optional[str] = None, id2label:
typing.Optional[dict[int, str]] = None, label2id:
typing.Optional[dict[str, int]] = None, num_labels:
typing.Optional[int] = None, task_specific_params:
typing.Optional[dict[str, typing.Any]] = None, problem_type:
typing.Optional[str] = None, tokenizer_class:
typing.Optional[str] = None, prefix: typing.Optional[str] =
None, bos_token_id: typing.Optional[int] = None,
pad_token_id: typing.Optional[int] = None, eos_token_id:
typing.Optional[int] = None, sep_token_id:
typing.Optional[int] = None, decoder_start_token_id:
typing.Optional[int] = None, **kwargs )
```

⚠ **Warning**

Even if some of properties may not given, or not exist when we use json to serialize the config and check it. we can still access it.

3. Trainer

`Trainer` is a **complete training and evaluation loop** for `pytorch` models. for more argument settings, see [4]

import Trainer from the transformers :

```
from transformers import Trainer
```

We need a `model` , `dataset` , a `preprocessor` , and a `data collator` to build batches of data from the dataset.

(1) Usage cases

Trainer can be used in the following cases :

1. The model **return tuples or subclasses of** `transformers.utils.ModelOutput` **(convert to tuple by** `to_tuple()` **)**
2. model can `compute the loss` in case the `labels` argument is provided (in `forward()` method) and `loss` is **returned as the first element of the tuple** (if your model returns tuples). i.e.,

```
# Forward pass with labels - this will compute the loss
outputs = model(
    input_ids=input_ids,
    attention_mask=attention_mask,
    labels=labels  # Ground truth label
)
loss
```

3. model can accept **multiple label arguments** (use `label_names` in TrainingArguments to indicate their name to the Trainer) but none of them should be named `"label"` (see 3. Model Structure)

To summary, the Trainer can do this :

```
outputs = model(**inputs)         # your model forward
loss = outputs.loss if return_outputs else outputs[0]
```

Note there is a **compute_loss** parameter, **this parameter can be used to override the loss function**, For example, if we want to modify loss function, we can define a Trainer and override the `compute_loss` function :

```
class MyTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False):
        outputs = model(**inputs)
        # here you can use inputs["class_ids"], inputs["segmentation_mask"], ...
        cls_logits = outputs.class_logits
        seg_logits = outputs.seg_logits
        bbox_preds = outputs.bbox_preds

        # custom loss calculation
        loss = ...

        return (loss, outputs) if return_outputs else loss
```

But if we doesn't need so, we can not provide `compute_loss` function.

(2) Model Output

Note

Model Output : Model output is inherited from `BaseModelOutput` , Which must have following `properties` :

1. `last_hidden_state`
2. `hidden_states` (Optional)
3. `attentions` (Optional)

Refer from `Model Output` [5], we can simply get an output example by following :

```
from transformers import BertTokenizer, BertForSequenceClassification
import torch

tokenizer = BertTokenizer.from_pretrained("google-bert/bert-base-uncased")
model = BertForSequenceClassification.from_pretrained("google-bert/bert-base-uncased")

inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
labels = torch.tensor([1]).unsqueeze(0)  # Batch size 1
outputs = model(**inputs, labels=labels)
```

here, `output` is a `ModelOutput` object. and calling `to_tuple` , it returns a tuple of loss and logits.

```
(tensor(0.5465, grad_fn=<NllLossBackward0>),  # loss
 tensor([[-0.1871,  0.1315]], grad_fn=<AddmmBackward0>)) # logits
```

For simply classifier, the forward method of model should return like a tuple, also **when labels are passed, the Trainer will see the loss** (for a detailed example method of forward method, see 3. Model Structure)

To construct a `Model Output`, we only need to make a class that inherits from `ModelOutput` like following, we can define the output really easy :

```python
import torch
from transformers.utils import ModelOutput
from dataclasses import dataclass
from typing import Optional

@dataclass
class MultiTaskModelOutput(ModelOutput):
    loss: Optional[torch.Tensor] = None
    logits_classify: Optional[torch.Tensor] = None
    logits_segmentation: Optional[torch.Tensor] = none

def main():
    output = MultiTaskModelOutput(
        loss=0.5,
        logits_classify=torch.randn(2, 10),
        logits_segmentation=torch.randn(2, 3, 224, 224),
    )
    print(output)
    print(output.to_tuple())
```

Note

1. use `torch.Tensor` as much as possible
2. if `out` is a `ModelOutput` not use `out.loss=loss`, this will not add props correctly.

(3) Training Arguments

For customize the training process, we can use `TrainingArguments` [6] for settings of model training process. e.g., batch size, learning rate, mixed precision, `torch.compile`.

```python
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="distilbert-rotten-tomatoes",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=2,
    push_to_hub=True,
)
```

Then, we can train the model by another custom datasets like specified by us :

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=dataset["train"],
    eval_dataset=dataset["test"],
    tokenizer=tokenizer,
    data_collator=data_collator,
)
trainer.train()
# to upload it to hub, using
model.push_to_hub("my-awesome-model")  # with the model name specified.
```

4. A Complete HF Model Structure Tutorial

(1) Process illustration

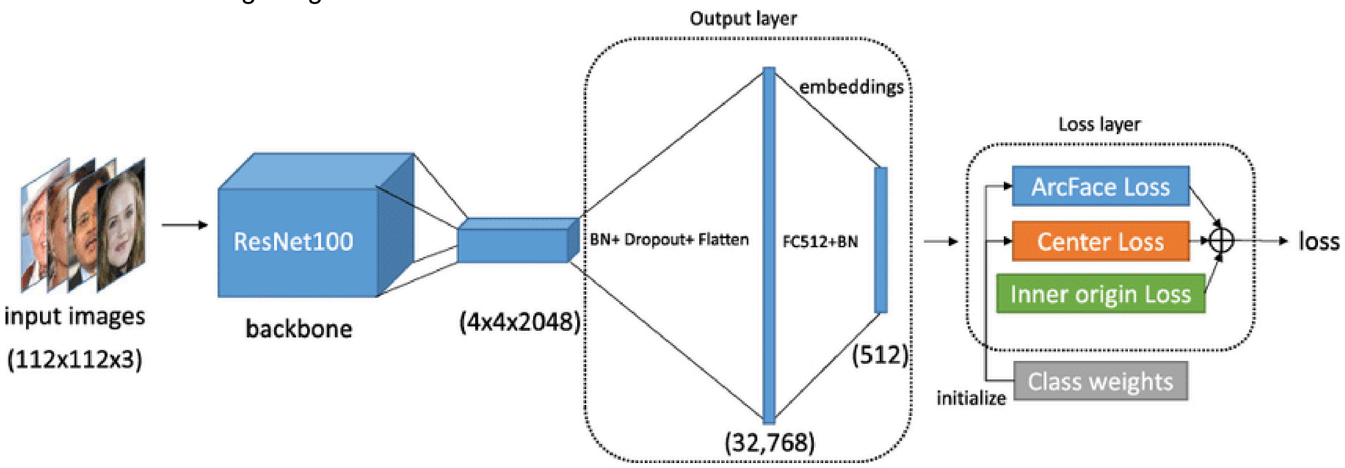
To use pretrained model to **handle tasks in different datasets, which may contains** , In that case, we should :

1. keep the pretrained backbone (reuses ResNet50 pretrained weights).
2. **Build a small segmentation decoder** (simple upsampling / conv decoder or FPN) that outputs per-pixel logits for **9 classes** (plus optional background).
3. Train the network

(2) Backbone of a model

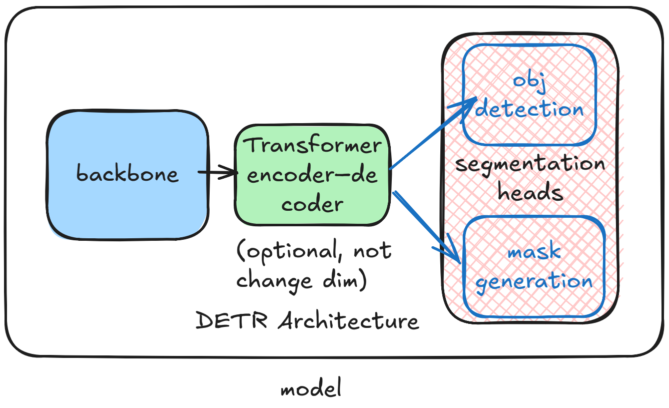
Every model is built from backbone and header. **"backbone" refers to the feature extracting network which is used within the model**. Which is used to **encode the network's input into a certain feature representation.**

We use the following image for illustration :

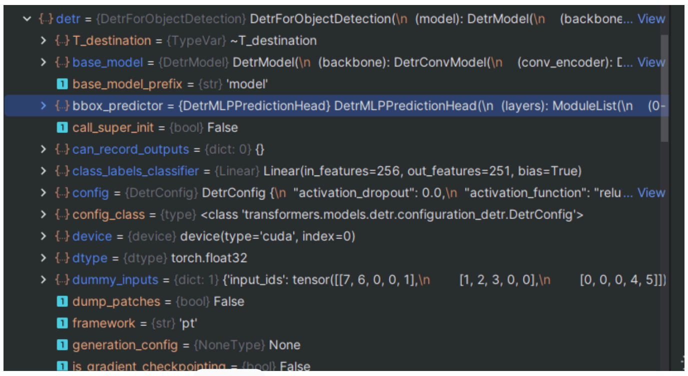


For many models, we can directly use `AutoBackbone` if the model is backbone only. For All models that support backbone, see[7] for details .

```
from transformers import AutoBackbone
# also following are common :
transformers.TimmBackbone
```



We still use the `"facebook/detr-resnet-50-panoptic"` image segmentation model,
This model has following properties :



```
⌄ {..} detr = {DetrForObjectDetection} DetrForObjectDetection(\n  (model): DetrModel(\n    (backbone... View
  > {..} T_destination = {TypeVar} ~T_destination
  > {..} base_model = {DetrModel} DetrModel(\n  (backbone): DetrConvModel(\n    (conv_encoder): [... View
    base_model_prefix = {str} 'model'
  > {..} bbox_predictor = {DetrMLPPredictionHead} DetrMLPPredictionHead(\n  (layers): ModuleList(\n    (0-
    call_super_init = {bool} False
  > {..} can_record_outputs = {dict: 0} {}
  > {..} class_labels_classifier = {Linear} Linear(in_features=256, out_features=251, bias=True)
  > {..} config = {DetrConfig} DetrConfig {\n  "activation_dropout": 0.0,\n  "activation_function": "relu... View
  > {..} config_class = {type} <class 'transformers.models.detr.configuration_detr.DetrConfig'>
  > {..} device = {device} device(type='cuda', index=0)
  > {..} dtype = {dtype} torch.float32
  > {..} dummy_inputs = {dict: 1} {'input_ids': tensor([[7, 6, 0, 0, 1],\n        [1, 2, 3, 0, 0],\n        [0, 0, 0, 4, 5]])
    dump_patches = {bool} False
    framework = {str} 'pt'
    generation_config = {NoneType} None
    is_gradient_checkpointing = {bool} False
```

We note actual `detr` structure is :

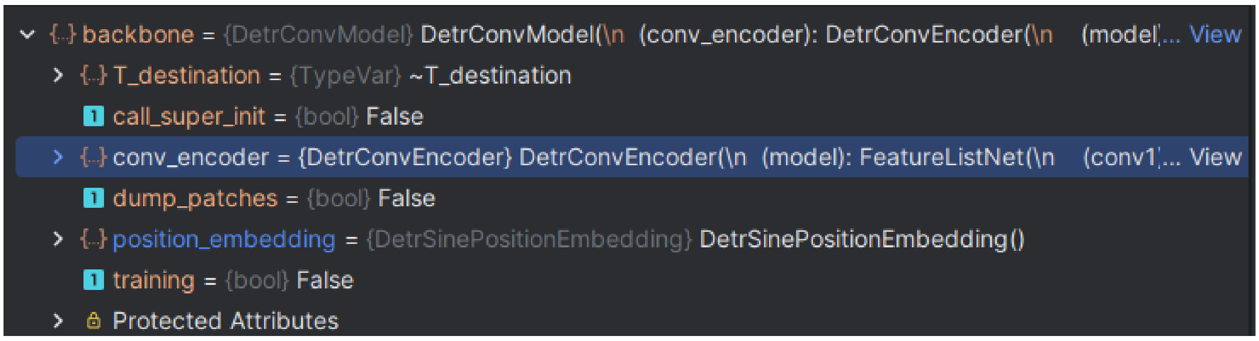
```
Image ──→ CNN backbone (ResNet-50)
        └─→ Transformer Encoder
                 └─→ Transformer Decoder
                        └─→ Prediction Heads
                               ├─→ Class head (object class logits)
                               └─→ Box head (bounding boxes)
                               └─→ Mask head (for panoptic/segmentation variant)
```

So we can get it's `resnet50` backbone by following(check [8]), and those structure are all correspond to :

```python
from transformers import AutoModelForImageSegmentation, AutoImageProcessor
model = AutoModelForImageSegmentation.from_pretrained(
    "facebook/detr-resnet-50-panoptic", device_map="auto", dtype="auto")
# For the DetrForSegmentation model, we can access its
model_detr = model.detr.model
model_detr.backbone
model_detr.encoder   # transformer encoder, in_features=256,
model_detr.decoder   # transformer decoder  in_features=256, out_features = 256
model.bbox_attention    # attention layer before bbox

model.mask_head              # Mask head
model.detr.bbox_predictor    # BBox head
model.detr.class_labels_classifier # Class Head,   Linear(in_features=256,
out_features=251, bias=True)
```

Note if we access `model.detr.model.backbone` with the model weight trained (using `AutoModelForImageSegmentation.from_pretrained`), This backbone **already has pretrained weights** —

they were **loaded together with the full DETR checkpoint.**



```
⌄ {...} backbone = {DetrConvModel} DetrConvModel(\n  (conv_encoder): DetrConvEncoder(\n    (model)... View
  > {...} T_destination = {TypeVar} ~T_destination
    call_super_init = {bool} False
  > {...} conv_encoder = {DetrConvEncoder} DetrConvEncoder(\n  (model): FeatureListNet(\n    (conv1)... View
    dump_patches = {bool} False
  > {...} position_embedding = {DetrSinePositionEmbedding} DetrSinePositionEmbedding()
    training = {bool} False
  > Protected Attributes
```

In that case, **we can can then**:

- **Freeze it and use it as a feature extractor for your own dataset (but output would like same in original network)**.
- **Fine-tune it with a new head (classification, segmentation, etc.).**

We use a large scale fish dataset[9] as segmentation example, the load process of dataset can be refer from [10].

For use `Train` method for fine-tuning, we need following `self-defined parameters`, including :

1. data collator [11],
2. Compute metrics

(3) Data Collator

1. Creation for data collator

For full document, see [12]. We need to **tell our `Trainer` how to form batches from the pre-processed inputs**. this is what Data Collator should do. So Data Collator is used for return `dictionary_like` data. which [10-1] has given a rich loader (use this may not need collator since processor has been applied).

Note

Often we add data collator if dataset return information is not enough, this also do :

1. apply processor
2. other things, like computing bounding boxes.

It can be just **self-define a data collator (the `__call__` method should be implemented**). An example for that copy from [13] is given as follows :

Load datasets in jupyter

In another script, use `dataset = datasets.load_dataset("swag", "regular", keep_in_memory=True)` to keep it. then directly use it in jupyter.

Important

Return Rule for DataCollator

1. **Every item the model or loss function uses must be a torch.Tensor** (not a list) with consistent first dimension = batch size.
2. Dtypes:
   - Image / pixel values: `float32`
   - Padding mask (`pixel_mask`): `bool` (or `uint8` convertible to bool)
   - Segmentation targets: `long` (class indices per pixel)
   - Classification targets: `long`
   - Bounding boxes: `float32`
3. return as less list as possible, if needed, we can put it in `meta` part (this is for avoiding moving to other devices).
   We note that the result from processor is `list[np.array]` , so we still need to convert it to a tensor :

Also, `DataCollator` only need `__call__()` method be called correctly, but `DataCollator` is **not a real base class** — it's a `typing.NewType` alias. So we only define it and needn't inherit from `DataCollator` class.

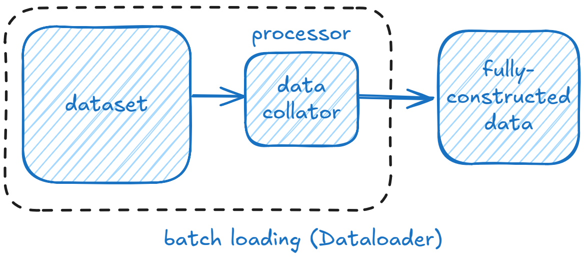
```
from dataclasses import dataclass
from typing import Optional, Union
from transformers import DataCollator
import torch

@dataclass
class FishSegmentDataCollator:  # DataCollator
    processor : any # Replace 'any' with the actual type of your processor
    def __call__(self, batch):
        """
        Features is dataset,
        """
        images = torch.stack([feature["pixel_values"] for feature in batch])  #
        seg_masks = torch.stack([resize_mask(feature["segmentation_mask"], size=(800,
1060)) for feature in batch])

        # resize masks to  [800, 1060]

        class_names = [feature.get("class_name", None) for feature in batch]    # class
name
        class_ids = [feature.get("label", None) for feature in batch]           # class id

        # including : pixel_values, pixel_mask
        encodings = self.processor(images)

        meta = []
        bboxes = []
        for mask, cls_name in zip(seg_masks, class_names):  # masks = list of numpy or
torch 2D arrays
            bbox = _compute_bbox_from_mask(mask.cpu().numpy())
            if bbox is None:
                continue
            bboxes.append(bbox)
            area = bbox[2] * bbox[3]
            x, y, w, h = bbox
            # 4 vertices of the rectangle
            segmentation = [x, y, x + w, y, x + w, y + h, x, y + h]

            meta.append({
                "class_name": cls_name,
                "area": torch.tensor([area]),
                "segmentation": segmentation,
            })

        result = dict(encodings)   # expand storage

        # return tensor for training parameters
        result["pixel_values"] = torch.tensor(np.stack(encodings["pixel_values"]),
dtype=torch.float)
        result["pixel_mask"] = torch.tensor(np.stack(encodings["pixel_mask"]),
dtype=torch.long)  # attention mask
        result["segmentation_mask"] = torch.stack(seg_masks) # caution: not use
"pixel_mask"
        result["class_ids"] = torch.stack(class_ids)
        result["bboxes"] = torch.tensor(bboxes, dtype=torch.float) if bboxes else
torch.empty((0, 4), dtype=torch.float)

        result["meta"] = meta  # meta data list
        return result
```

The above part returns the following structure, which is friendly for `Trainer` later :

```
{
  "pixel_values": FloatTensor(B,3,H,W),
  "pixel_mask": BoolTensor(B,H,W),          # optional if no padding
  "segmentation_mask": LongTensor(B,Hs,Ws), # ground-truth per-pixel
  "class_labels": LongTensor(B,),           # image-level class (or fish class)
  "bboxes": FloatTensor(B,4),               # normalized or absolute
  "meta": {
      "class_names": [...],                 # list length B
      "raw_labels": [...],                  # original list if needed
      "image_ids": [...],
      # anything else you want
  }
}
```

2. Why use data collator

`Data Collator` is a natural built-in system for pre-processing in `DataLoader`, we can just use `collate_fn` parameter for processing in batch data. i.e.,



Note : the output of data collator should be a dictionary with every value a list.

So we firstly define a `data collator` like following, and use processor and input, receive a `batch` and construct the new data batch dictionary like following, note
we **replace the pixel_mask term by the training data pixel mask, but keep the processed image pixel value data** :

⚠ **Mask Processing**

In hugging face DETR preprocessing process :

1. `pixel_values` refers to preprocessed images
2. `pixel_mask` is `boolean mask` marking valid pixels
   So, `pixel_mask` is valid mask, this **shouldn't be override, this should be kept**

```
def resize_mask(mask, size=(800, 1060)):
    """
    Resize segmentation mask to given size using nearest neighbor interpolation.
    Args:        mask (torch.Tensor or PIL.Image): Input mask.            - If
torch.Tensor: shape (H, W) or (1, H, W).            - If PIL.Image: will be converted to
tensor.        size (tuple): target size (height, width).
    Returns:        torch.Tensor: resized mask, shape (H_new, W_new), dtype long    """
if isinstance(mask, Image.Image):
        mask = torch.as_tensor(np.array(mask), dtype=torch.long)

    if mask.ndim == 2:  # (H, W)
        mask = mask.unsqueeze(0).unsqueeze(0)  # -> (1,1,H,W)
    elif mask.ndim == 3 and mask.shape[0] == 1:  # (1,H,W)
        mask = mask.unsqueeze(0)  # -> (1,1,H,W)
    else:
        raise ValueError(f"Unsupported mask shape: {mask.shape}")

    # nearest neighbor interpolation for masks
    mask_resized = F.interpolate(mask.float(), size=size, mode="nearest")
    mask_bin = (mask_resized > 0).long()   # must binarize here !!!
    return mask_bin.squeeze()  # -> (H_new, W_new)
```

Then we can **use the data collator as a embedded function in DataLoader** by following code :

```
processor = AutoImageProcessor.from_pretrained(
    "facebook/detr-resnet-50-panoptic",
    use_fast=True,
) # use original normalization
dataset = load_fish_dataset()
collator = FishSegmentDataCollator(processor=processor) # processor is
data_loader = DataLoader(dataset, batch_size=3, collate_fn=collator)

batch = next(iter(data_loader))
print("Batch keys:", batch.keys())  # Batch keys: dict_keys(['pixel_mask', 'pixel_values',
'segmentation_mask', 'class_ids', 'bboxes', 'meta'])
```

Then we can implement a function named `build_dataloaders` , which will apply processor :

Hint

In actual train process, we needn't use the following `data_loader` . We can specify the parameter in `Trainer`

```
def build_dataloaders(train_size=0.7, test_size=0.2, batch_size=8) \
        -> tuple[DataLoader, DataLoader, DataLoader]:
```

```
    assert train_size + test_size < 1, f"train_size + test_size must be less than 1, but
got {train_size + test_size}"

    # Load the fish dataset -> no need more process since it
    dataset = load_fish_dataset()
    train_len = int(train_size * len(dataset))
    test_len = int(test_size * len(dataset))
    val_len = len(dataset) - train_len - test_len

    # Split the dataset into train, test, and validation sets
    generator = torch.Generator().manual_seed(42)  # for reproducibility of train-test
split
    train_dataset, val_dataset, test_dataset = random_split(
        dataset, [train_len, val_len, test_len], generator=generator
    )
    processor = AutoImageProcessor.from_pretrained(
        "facebook/detr-resnet-50-panoptic",
        use_fast=True,
    )
    collator = FishSegmentDataCollator(processor=processor) # processor
    train_loader = DataLoader(train_dataset,
                              batch_size=batch_size,
                              shuffle=True,
                              pin_memory=True,
                              num_workers=4,
                              prefetch_factor=2,
                              persistent_workers=True,
                              collate_fn=collator)
    ...
    return train_loader, val_loader, test_loader
```

(4) Combine them all together

1. use backbone to get hidden states.

For using the backbone for training, `backbone` of DETR takes two parameters (image and mask, all should be float).

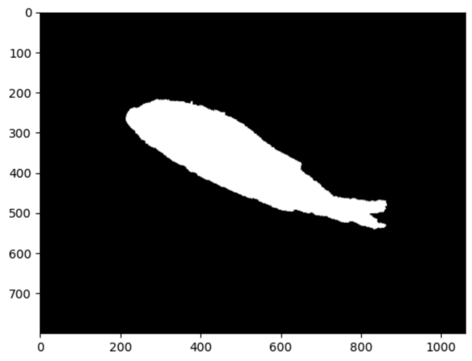
```
from transformers import AutoModelForImageSegmentation
from Fish_Pretrain import build_dataloaders
import torch

model = AutoModelForImageSegmentation.from_pretrained(
        "facebook/detr-resnet-50-panoptic", device_map="auto", dtype="auto"
    )
backbone = model.detr.model.backbone

train_loader, _, _ = build_dataloaders(0.7, 0.2)
sample = next(iter(train_loader))
sample.keys() # dict_keys(['pixel_mask', 'pixel_values', 'labels', 'class_names',
'class_ids'])

# get the
val = sample.get("pixel_values").to("cuda")
mask = sample.get("pixel_mask").to("cuda")
out = backbone(val, mask)    # call the backbone for training.
```

Not only train the backbone

We can also use `out = model(val, mask)`, which is using the full model for train, but actually, since it is a 251 segmentation model, we have :

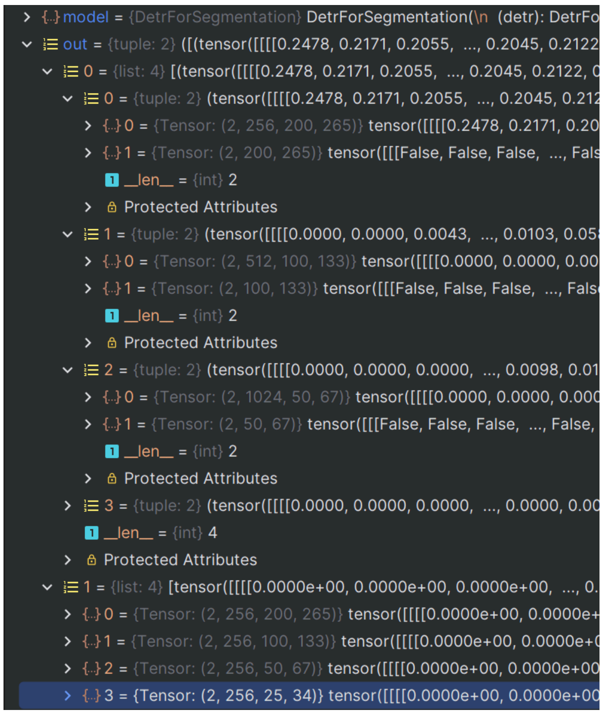
We resize the mask to fit the input of the processor.

```python
m = mask[0].cpu().numpy()
import matplotlib.pyplot as plt
plt.imshow(m, cmap='gray')
plt.show()
```



```python
out = model(val, mask)
out.logits.shape
Out[7]: torch.Size([2, 100, 251])    # 251 classes, but we don't want it here.
```

The output of the `backbone` is an `DetrConvModel` object, which **contains all hidden states of the model (where 0, 1 in the first layer is hidden states)** and the size of `[(src, mask)]` is



```
> {...} model = {DetrForSegmentation} DetrForSegmentation(\n  (detr): DetrFo
v ☰ out = {tuple: 2} ([(tensor([[[[0.2478, 0.2171, 0.2055,  ..., 0.2045, 0.2122
  v ☰ 0 = {list: 4} [(tensor([[[[0.2478, 0.2171, 0.2055,  ..., 0.2045, 0.2122, 0
    v ☰ 0 = {tuple: 2} (tensor([[[[0.2478, 0.2171, 0.2055,  ..., 0.2045, 0.212
      > {...} 0 = {Tensor: (2, 256, 200, 265)} tensor([[[[0.2478, 0.2171, 0.20
      > {...} 1 = {Tensor: (2, 200, 265)} tensor([[[False, False, False,  ..., Fals
        1 __len__ = {int} 2
      > Protected Attributes
    v ☰ 1 = {tuple: 2} (tensor([[[[0.0000, 0.0000, 0.0043,  ..., 0.0103, 0.05
      > {...} 0 = {Tensor: (2, 512, 100, 133)} tensor([[[[0.0000, 0.0000, 0.00
      > {...} 1 = {Tensor: (2, 100, 133)} tensor([[[False, False, False,  ..., Fals
        1 __len__ = {int} 2
      > Protected Attributes
    v ☰ 2 = {tuple: 2} (tensor([[[[0.0000, 0.0000, 0.0000,  ..., 0.0098, 0.01
      > {...} 0 = {Tensor: (2, 1024, 50, 67)} tensor([[[[0.0000, 0.0000, 0.000
      > {...} 1 = {Tensor: (2, 50, 67)} tensor([[[False, False, False,  ..., False,
        1 __len__ = {int} 2
      > Protected Attributes
    > ☰ 3 = {tuple: 2} (tensor([[[[0.0000, 0.0000, 0.0000,  ..., 0.0000, 0.00
      1 __len__ = {int} 4
    > Protected Attributes
  v ☰ 1 = {list: 4} [tensor([[[[0.0000e+00, 0.0000e+00, 0.0000e+00,  ..., 0.
    > {...} 0 = {Tensor: (2, 256, 200, 265)} tensor([[[[0.0000e+00, 0.0000e+
    > {...} 1 = {Tensor: (2, 256, 100, 133)} tensor([[[[0.0000e+00, 0.0000e+0
    > {...} 2 = {Tensor: (2, 256, 50, 67)} tensor([[[[0.0000e+00, 0.0000e+00
    > {...} 3 = {Tensor: (2, 256, 25, 34)} tensor([[[[0.0000e+00, 0.0000e+00
```

Where **the first is features**, and **the second is masks**,

```
# out[0] is a list of 4 feature maps:
(B, 256, 200, 265)
```

```
(B, 512, 100, 133)
(B, 1024, 50, 67)
(B, 2048, 25, 34)
# This matches ResNet50's FPN/stride outputs (downsampling by 4, 8, 16, 32).

# out[1] is a list of 4 boolean masks:
(B, 200, 265)
(B, 100, 133)
(B, 50, 67)
(B, 25, 34)   # mask
```

So what we want is **the last layer of hidden layer and masks**, so :

```
features_list, masks_list = out     # get the feature map and mask

# Note W, H is 25, 34, (original  is 800, 1060)
feature = features_list[-1]  # (B, 2048, W, H), use the last feature map
mask = masks_list[-1]  # (B, 25, 34)
```

2. Segment Head Definition

We need to self-define all the task head we need, When checking

The segment header should give the pixel of each class in the head output, i.e., using an input of `(B, C, H, W)` , the output should be `(B, class +1, H, W)` .

Also, recall the `attention layer` [14][15], we can firstly add an attention layer, which use `hidden_dim` as hidden num. So we have :

Down Sampling in Segment

We know the output of `resnet 50` model includes down sampling, i.e., `backbone` of ResNet50 outputs shape like `(B, C, W/32, H/32)` . Which will leads **the computation of segment loss can't directly use this data**.

A method is `upsampling` , but that may leads to more computation budget and lower accuracy.

Another is down sampling. Which **use down sampling** to reduce the output size of segmentation Head. (this will result in lower resolution, but generally better).

```
import torch
from torch import nn

class FishSegmentationHead(nn.Module):
    """
    FishSegmentationHead predicts pixel-wise segmentation masks from input feature maps
using attention and convolutional decoding layers.

    Input shape:
        x: torch.Tensor of shape (B, C, H, W)
            B: batch size
            C: number of channels (default: 256)
            H, W: spatial dimensions
    Output shape:
        torch.Tensor of shape (B, num_classes + 1, H, W)
```

```
            Each output contains per-pixel logits for each class (including background),
            upsampled to (SEG_H, SEG_W) resolution (default: 512x512).

    Attributes:
        reduce (nn.Conv2d): 1x1 convolution to reduce channel dimension.
        attn (nn.MultiheadAttention): Multi-head self-attention layer operating on reduced
features.
        decode (nn.Sequential): Convolutional decoder for segmentation logits.
    """
    def __init__(self, in_ch=256, hidden=128, num_classes=9):
        super().__init__()
        self.reduce = nn.Conv2d(in_ch, hidden, 1)
        self.attn = nn.MultiheadAttention(embed_dim=hidden, num_heads=8, batch_first=False)
        self.decode = nn.Sequential(
            nn.Conv2d(hidden, 128, 3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 64, 3, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Conv2d(64, num_classes + 1, 1)
        )

    def forward(self, x, seg_output_size: tuple):
        """
        Forward pass for segmentation mask prediction.

        Args:
            x (torch.Tensor): Input tensor of shape (B, C, H, W)
            seg_output_size (tuple): Desired output size (H, W) for segmentation masks. (we
may need down sampling)
        Returns:
            torch.Tensor: Output tensor of shape (B, num_classes + 1, H, W) with per-pixel
logits.
        """
        B, C, H, W = x.shape
        x = self.reduce(x)             # (B,hidden,H,W)
        seq = x.view(B, -1, H * W).permute(2, 0, 1)  # (L,B,hidden)
        attn_out, _ = self.attn(seq, seq, seq)
        attn_out = attn_out.permute(1, 2, 0).view(B, -1, H, W)
        logits = self.decode(attn_out)
        logits = F.interpolate(logits, size=seg_output_size, mode='bilinear',
align_corners=False)
        return logits
```

3. Model Structure

Since we use 256 as input, which is transformer does in BERT model, we can reduce the channel by a convolution layer `self.conv`. So we can Implement the full model like following :

1. Firstly we **get the backbone**
2. Then connect the backbone with each head we define here.

Important

A key point is the input in `forward function`, **We input only raw image from the dataset, which means, the boundary mark, e.g. ground truth(GT), not take part in training process**. So the input is `pixel_values` and `pixel_mask`. And `segmentation_mask` (real mask) shouldn't be transferred to model.

1) Using Labels as model input

✓ **Hugging face model design**

In hugging face, the model is designed to return a `ModelOuput` class (see (2) Model Output), and **besides** `x`, it should receive another keyword `labels` for computing `loss`

When `labels` is not passed, It returns predicted value. And when it is passed, it also return `loss`

The minimal forward structure is as follows :

```python
def forward(self, input_ids=None, labels=None):
    logits = self.fc(input_ids)
    loss = None
    if labels is not None:
        loss = F.cross_entropy(logits, labels)
    # If loss is present, return (loss, logits)
    if loss is not None:
        return (loss, logits)
    else:
        return (logits,)
```

Note `labels` is the general case, i.e. `input_id` as `x` parameter, `labels` as `y` parameter. But for multi-tasks with `y1, y2, ...`, we should use the method below

2) For multiple labels

In the case of multi-task, we often have **multiple lables**, in that case, we can add `label_names` into training Arguments :

```python
training_args = TrainingArguments(
    ...,
    label_names=["class_ids", "segmentation_mask", "bboxes"]
)
```

This will be seen as `kwargs` of forward. which should be :

```python
def forward(self, input_ids=None, class_ids=None, segmentation_mask=None, bboxes=None,
**kwargs):
    ...
```

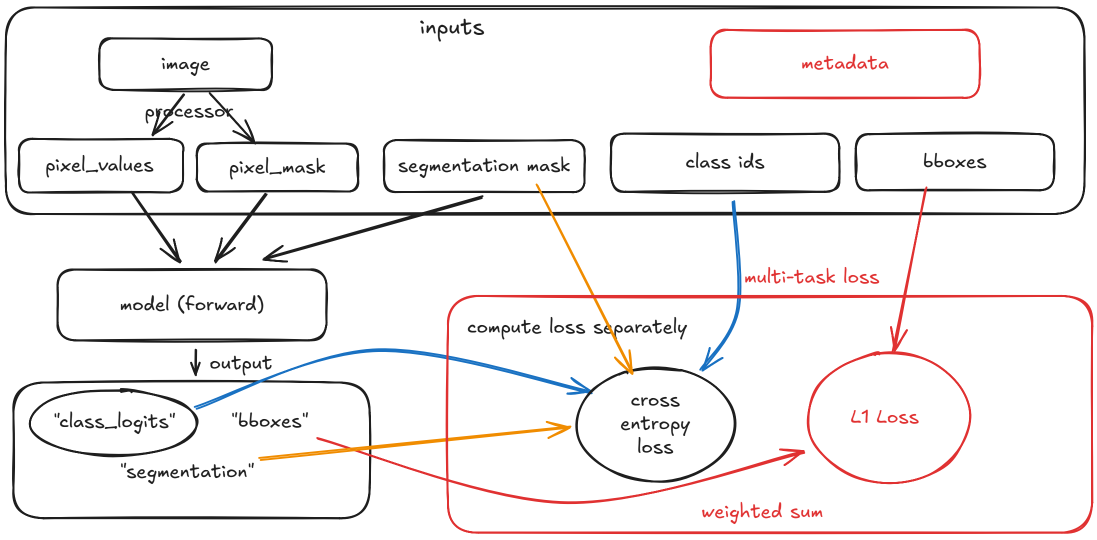
Tip

This design also have a benefit, **when using labels in loss function, we can use `model(**input)` conveniently in a general case** to get loss and output. No need to add **kwargs

3. Loss function definition for multi-heads

Since we **replace the heads of the original transformer**, built-in DETR loss will no longer applies, so we must **define a composite muti-task loss**.

Firstly, recall Our Data Collator Part, it returns a dictionary of `dict_keys(['pixel_mask', 'pixel_values', 'segmentation_mask', 'class_ids', 'bboxes', 'meta'])` , Also, in 3. Model Structure output is `"class_logits", "bboxes"` and `"segmentation"` . So we use that as input :



For Classification and segmentation loss, we **use cross entropy loss for class and logits**, and `l1_loss` for `bbox` .

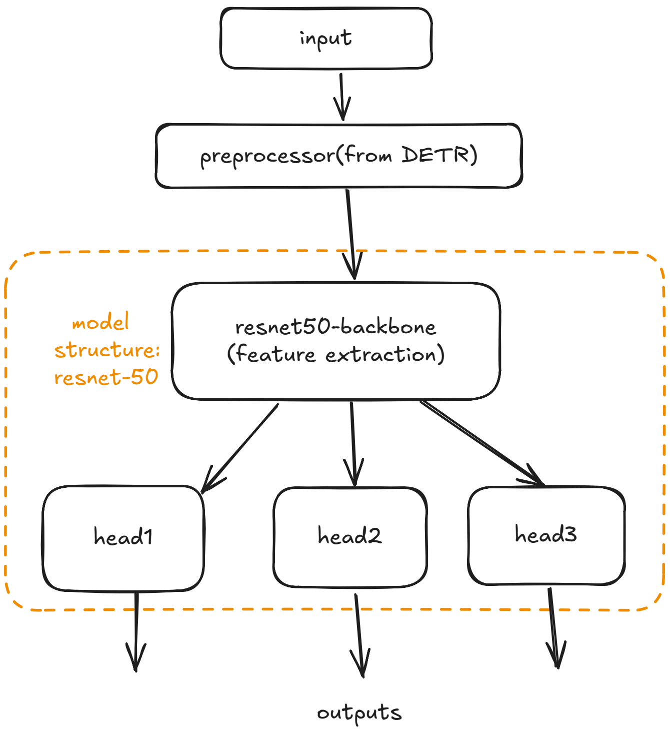
Note that **The final loss should be calculated using weighted sum**.

So in following process we will give it a **multi-task loss implement**.

4. A Complete Example

After all the knowledge above, we got the Model code of `FishSegmentation` :

before that, firstlym we draw the model structure like this :


input
preprocessor(from DETR)
model structure: resnet-50
resnet50-backbone (feature extraction)
head1
head2
head3
outputs

```python
# region Model Definition
@dataclass
class FishSegmentationModelOutput(ModelOutput):
    """
    All members must be Tensor type
    """
    loss: Optional[torch.Tensor] = None
    class_logits: Optional[torch.Tensor] = None
    seg_logits: Optional[torch.Tensor] = None
    bbox_logits: Optional[torch.Tensor] = None


class FishSegmentationModel(torch.nn.Module):
    def __init__(self, model_name: str, hidden_ch=256):
        super(FishSegmentationModel, self).__init__()
        # get base model from huggingface
        model = AutoModelForImageSegmentation.from_pretrained(
            "facebook/detr-resnet-50-panoptic", device_map="auto", dtype="auto"
        )
        classes = get_fish_classes()
        num_classes = len(classes)
        model.config.num_labels = num_classes  # Update this based on your dataset

        # extract resnet50 pretrained backbone, also re-use the bbox head
        backbone = model.detr.model.backbone  # (B, 256, W, H)

        self.model_name = model_name
        self.backbone = backbone

        self.conv = nn.Conv2d(2048, hidden_ch, kernel_size=1)  # reduce channels from 2048
to hidden_ch
        # much smaller resolution is fine for classification & bbox regression, but quite
coarse for pixel segmentation.
        self.classifier = FishSegmentClassifier(in_ch=hidden_ch,
                                                num_classes=num_classes)  # (hidden_ch ->
num_classes) classifier
        self.bbox_head = FishSegmentBBoxHead(hidden_ch)  # (hidden_ch -> 4) bbox head
        self.segmentation_head = FishSegmentationHead(in_ch=hidden_ch,
num_classes=num_classes)

    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:
        """
        Not use "batch" for forward, this is firstly for explicitness, secondly for
compatibility with Trainer
        """
        # all hidden states and positional encodings are computed in the backbone
        features_list, masks_list = self.backbone(pixel_values, pixel_mask)
        # Note W, H is 25, 34, (original  is 800, 1060)
        feat, _ = features_list[-1]  # (B, 2048, W, H), use the last feature map
        feat = self.conv(feat)  # (B, hidden_ch, W, H)
        B, C, W_feat, H_feat = feat.shape
        # calculate output of each head
        class_logits = self.classifier(feat)  # (B, num_classes+1)
        bbox_logits = self.bbox_head(feat)  # (B, 4)
        seg_logits = self.segmentation_head(feat, seg_output_size=(W_feat, H_feat))  # (B,
num_classes+1, W, H)

        out = {
            "class_logits": class_logits,
            "seg_logits": seg_logits,
            "bbox_logits": bbox_logits,
        }
        if class_ids is not None and segmentation_mask is not None and bboxes is not None:
            loss = self.segmentation_loss_func(
                class_ids=class_ids,
                segmentation_mask=segmentation_mask,
                bboxes=bboxes,
                class_logits=class_logits,
                seg_logits=seg_logits,
                bbox_logits=bbox_logits,
                loss_weights=DEFAULT_LOSS_WEIGHTS,
                return_outputs=False,
            )
            # not only 3 construct and use out.loss = loss, this will not be recorded
            out["loss"] = loss
        return FishSegmentationModelOutput(**out)

    @staticmethod
    def segmentation_loss_func(
            class_ids, segmentation_mask, bboxes,
            class_logits, seg_logits, bbox_logits,
            loss_weights=None,
            return_outputs: bool = False,
    ) -> Union[torch.Tensor, tuple[torch.Tensor, FishSegmentationModelOutput]]:
        """
        Custom Multi-task loss function combining classification, segmentation, and
bounding box losses.

        use outputs = model(**inputs) if modified to compute_loss signature
        """
        if loss_weights is None:
            loss_weights = DEFAULT_LOSS_WEIGHTS

        # calculate individual losses
        cls_loss = F.cross_entropy(class_logits, class_ids)  # Classification CE
        box_loss = F.l1_loss(bbox_logits, bboxes)  # Bounding box L1

        # for segmentation, use different resolution
        W_feat, H_feat = seg_logits.shape[2], seg_logits.shape[3]  # (B, num_classes,
W_feat, H_feat)
        seg_targets = torch.stack(
            [resize_mask(mask, (W_feat, H_feat)) for mask in segmentation_mask])  # resize
to match seg_logits
        seg_loss = F.cross_entropy(seg_logits, seg_targets, ignore_index=255)  # no ignore
now; keep for extensibility
        tot_loss = (loss_weights["cls"] * cls_loss +
                    loss_weights["seg"] * seg_loss +
                    loss_weights["box"] * box_loss)  # weighted sum
        outputs = FishSegmentationModelOutput(
            loss=tot_loss,
            class_logits=class_logits,
            seg_logits=seg_logits,
            bbox_logits=bbox_logits,
        )
        return (tot_loss, outputs) if return_outputs else tot_loss
```

5. Process to train the model

(1) Load the dataset

Firstly, using knowledge of [10-2], we give a datasets loading method

```
def load_fish_datasets_all(train_size=0.7, test_size=0.2, random_seed=42):
    """
    Splits the fish segmentation dataset into training, validation, and test sets.
    """
    assert train_size + test_size < 1.0, "train_size and test_size must sum to less than
1.0"
    dataset = load_fish_dataset()
    train_len = int(train_size * len(dataset))
    test_len = int(test_size * len(dataset))
    val_len = len(dataset) - train_len - test_len

    g = torch.Generator().manual_seed(random_seed)
    train_set, test_set, val_set = random_split(
        dataset, [train_len, test_len, val_len],
        generator=g
    )
    return train_set, test_set, val_set
```

(2) Make torch Module into Transformer Pretrained Model

Firstly we can recall Chapter 3. Model Structure, we use `class FishSegmentationModel(torch.nn.Module)` . But its **Trainer-friendly** using the forward function we have provided. So we don't need to modify the forward function, just modify `__init__` :

The hugging face model **should be inherited from** `PretrainedModel` **class**, And `model_type` should be specified :

Also to config the parameter in the model, we will take in a `config` parameter, which is self-defined config. To make all the parameters we need in model train.

1) Fix pytorch model

```
from transformers import AutoConfig, AutoModel, PreTrainedModel, PretrainedConfig

class FishSegmentationModel(PreTrainedModel):
    config_class = FishSegmentModelConfig

    def __init__(self,
                 model_name: str = "fish_segmentation_model",
                 config: Optional[FishSegmentModelConfig] = None):
    if config is None:
        config = FishSegmentModelConfig(model_name=model_name)
    super().__init__(config)
    # ..... (original code)
```

2) Add Model Config

Similarly, we can **define a custom config class** from `PretrainedConfig` :

```
class FishSegmentModelConfig(PretrainedConfig):
    model_type = "fish_segmentation_model"

    def __init__(self, model_name: str = "fish_segmentation_model", hidden_ch=256,
num_labels: Optional[int] = None,
                 **kwargs):
        super().__init__(**kwargs)
        self.model_name = model_name
        self.hidden_ch = hidden_ch
        if num_labels is not None:
            self.num_labels = num_labels
```

3) Register Config and Model

We can add the following code after definition of our Config and Model to register the class, this is necessary if we want to make a backup at hugging face :

```
# Register the model and config to Hugging Face Auto classes
try:
    AutoConfig.register("fish_segmentation_model", FishSegmentModelConfig)
except ValueError:
    pass
try:
    AutoModel.register(FishSegmentModelConfig, FishSegmentationModel)
except ValueError:
    pass
```

After this, we can use `AutoConfig.from_pretrained("model_name")` to load the configuration, and using `AutoModel.from_pretrained("model_name")` to load the config and model.

Note this relies on the `model_type` specified in `config.json` of the model on hub website.

(3) Train!

Finally, we can use those simple code to train the model :

```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")

def train_model():
    train_size, test_size = 0.7, 0.2

    training_args = TrainingArguments(
        output_dir='./models',
        per_device_train_batch_size=8,
        per_device_eval_batch_size=8,
        label_names=["class_ids", "segmentation_mask", "bboxes"],
        do_train=True,  # train_set
        do_eval=True,  # val_set
        do_predict=True,  # test_set
        eval_strategy="steps",
        weight_decay=1e-2,
        num_train_epochs=10, # train epoch
        learning_rate=2e-4,  # specify learning rate
    )

    model = FishSegmentationModel(model_name="FishSegModel").to(DEVICE)
    train_set, eval_set, test_set = load_fish_datasets_all(train_size, test_size)

    processor = AutoImageProcessor.from_pretrained(
        "facebook/detr-resnet-50-panoptic", device_map="auto", dtype="auto")
    # here we specify datasets and data collator
    trainer = Trainer(
        model=model, args=training_args,
        train_dataset=train_set,
        eval_dataset=eval_set,
        data_collator=FishSegmentDataCollator(processor=processor),
        compute_metrics=None,  # can define custom metrics if needed
    )
    trainer.train()
```

Also, to save model locally, we should add following to save model, processor and configs.

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

Now we can train the model :

```
3%|▎         | 500/16880 [10:26<5:46:07,  1.27s/it]{'loss': 286.1008, 'grad_norm':
46.006587982177734, 'learning_rate': 0.00019408767777251185, 'epoch': 0.59}
9%|▉         | 1500/16880 [38:43<5:33:35,  1.30s/it]{'loss': 125.9102, 'grad_norm':
231.83041381835938, 'learning_rate': 0.000182239336492891, 'epoch': 1.78}
....
{'eval_loss': 43.80764389038086, 'eval_runtime': 217.6849, 'eval_samples_per_second':
8.269, 'eval_steps_per_second': 1.034, 'epoch': 19.55}
100%|██████████| 16880/16880 [7:50:26<00:00,  1.04it/s]{'train_runtime': 28226.7622,
'train_samples_per_second': 4.783, 'train_steps_per_second': 0.598, 'train_loss':
67.68818694923726, 'epoch': 20.0}
```

(4) Push to hugging face

For back up and publication, we will push all `configs` , `processor` and `model` to hugging face hub. So it's really simple :

```
# push model to hub
config.push_to_hub(
    repo_id=repo_id,
    private=False,
    commit_message="Add config for FishSegmentationModel"
)

processor.push_to_hub(
    repo_id=repo_id,
    private=False,
    commit_message="Add image processor"
)

model.push_to_hub(
    repo_id=repo_id,
    private=False,
    commit_message="Add trained FishSegmentationModel"
)
print("Model training and saving completed.")
```

Now we finish training 🥳 🎉

6. Test & Post Processing

Note if only for more models, we suggest **access more built-in models from** `tf.keras.applications.ResNet50` (there's a lot other to choose), which can be used more simple. But note theses are dry models, which may need more training epochs to converge;

(1) Load and Summary the Model

Loading model is very simple since `from_pretrained` also support to be load from local files, but since **our model is self-defined**, `AutoModel` and `AutoConfig` will not work. So **we should use our** `model, config` **and** `AutoProcessor` **instead**.

```
from transformers import AutoProcessor
from Fish_Pretrain.model_building import FishSegmentationModel, FishSegmentModelConfig

def load_models():
    os.chdir(os.path.dirname(os.path.abspath(__file__)))
    model_path = "./models/final_model"
    config = FishSegmentModelConfig.from_pretrained(model_path, local_files_only=True)
    model = FishSegmentationModel.from_pretrained(model_path, config=config,
local_files_only=True)
    processor = AutoProcessor.from_pretrained(model_path, local_files_only=True)
    return model, config, processor
```

Then after loading, we can use `torchinfo` and an example input to give model summary :

```
from torchinfo import summary

def print_model_summary(model):
    # Example input sizes
    batch_size = 1
    channels = 3
    height = 590
    width = 445
    pixel_values = torch.randn(batch_size, channels, height, width)
    pixel_mask = torch.ones(batch_size, height, width)
    summary(model,
            input_data={
                "pixel_values": pixel_values,
                "pixel_mask": pixel_mask
            }
    )
```

The model summary would be like this :

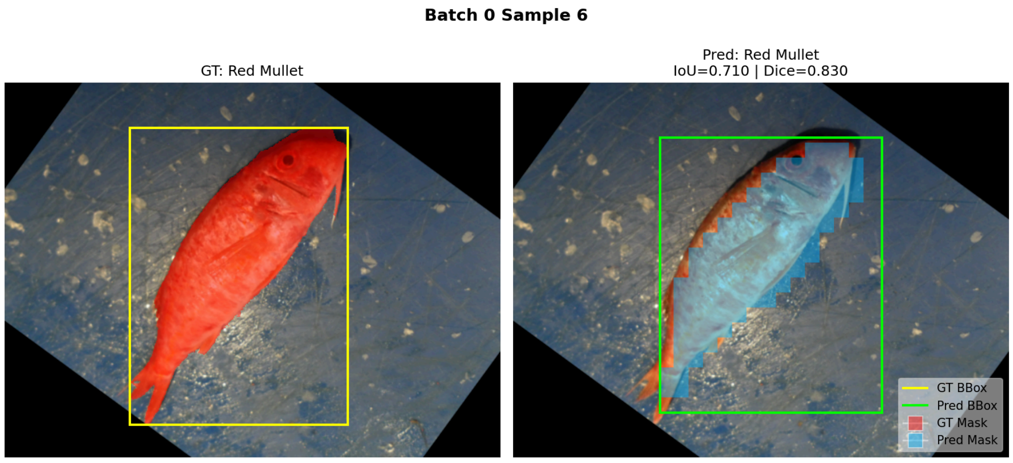
```
============================================================================
Layer (type:depth-idx)                  Output Shape              Param #
============================================================================
FishSegmentationModel                  [1, 4]                    --
├─DetrConvModel: 1-1                   [1, 256, 148, 112]        --
...
├─Conv2d: 1-2                          [1, 256, 19, 14]          524,544
├─FishSegmentClassifier: 1-3           [1, 10]                   --
│    └─AdaptiveAvgPool2d: 2-6          [1, 256, 1, 1]            --
│    └─Linear: 2-7                     [1, 10]                   2,570
├─FishSegmentBBoxHead: 1-4             [1, 4]                    --
...
├─FishSegmentationHead: 1-5            [1, 10, 19, 14]           --
...
============================================================================================
========================
```

(2) Test Model on some images

We can re-use `load_fish_dataset_all` to load some samples for test.

```python
def test_fish_segmentation_model(model: FishSegmentationModel, processor, batch_size: int =
4, limit_batches: int = 1, save_dir: Optional[str] = None):
    """Run evaluation on part of the test set and visualize each sample."""
    _, _, test_set = load_fish_datasets_all()
    collator = FishSegmentDataCollator(processor=processor)
    test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=True, num_workers=2,
collate_fn=collator)
    fish_classes = get_fish_classes()
    model.eval()
    with torch.no_grad():
    for b_idx, inputs in enumerate(test_loader):
        images: torch.Tensor = inputs["pixel_values"]  # (B,3,H,W)
        true_labels: Optional[torch.Tensor] = inputs.get("class_ids")
        true_bboxes: Optional[torch.Tensor] = inputs.get("bboxes")
        true_masks: Optional[torch.Tensor] = inputs.get("segmentation_mask")

        pred_labels: torch.Tensor = outputs.class_logits.argmax(dim=-1)
        pred_bboxes: torch.Tensor = outputs.bbox_logits  # (B,4)
        seg_logits: torch.Tensor = outputs.seg_logits    # (B,C,H,W)
        pred_masks: torch.Tensor = seg_logits.argmax(dim=1)  # (B,H,W)

        # then we can add our customized picture draw function
```

The example output is shown as following :



Now we have trained a segmentation model with something reasonable accuracy 😊 !

(3) Publish A Dataset to Hugging Face

Hugging Face will **automatically converts them into its own efficient Arrow-backed dataset structure**, even if you give it raw files, so uploading the raw files directly is available too.

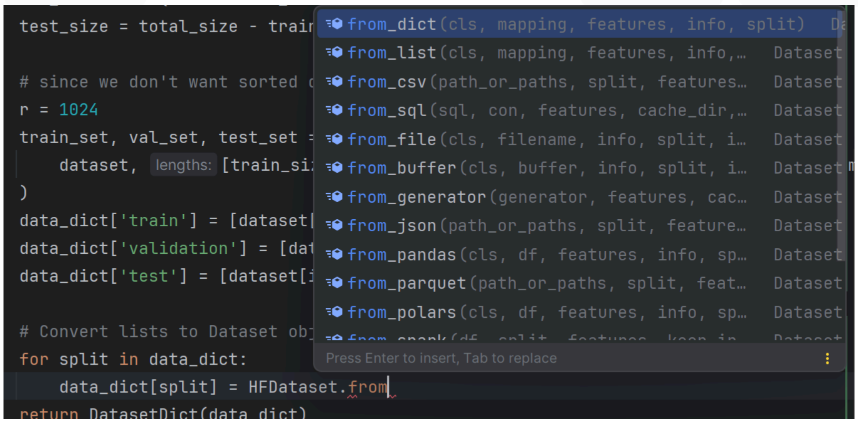
But If we want loading it as our customized structure, We need write a custom script to **upload our dataset into hugging face for others to easily use it**.

1) Convert to Hugging Face Dataset

Firstly, the **dataset in datasets** is `Dataset` , we can import it as `HFDataset` (to distinguish from `torch.utils.data.Dataset`)

```
from datasets import Dataset as HFDataset
```

`HFDataset` has many converting methods, like `from_list` , `from_dict` , `from_generator` ...



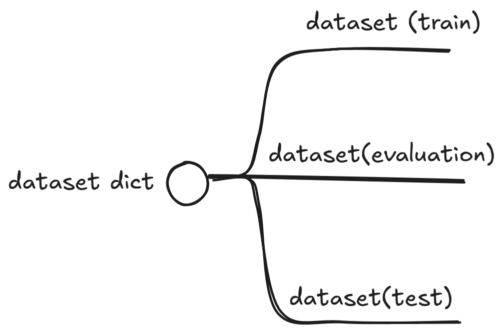
```
test_size = total_size - trai

# since we don't want sorted 
r = 1024
train_set, val_set, test_set =
    dataset, lengths: [train_si
)
data_dict['train'] = [dataset[
data_dict['validation'] = [dat
data_dict['test'] = [dataset[i

# Convert lists to Dataset obj
for split in data_dict:
    data_dict[split] = HFDataset.from
return DatasetDict(data_dict)
```

```
from_dict(cls, mapping, features, info, split)      Da
from_list(cls, mapping, features, info,…            Dataset
from_csv(path_or_paths, split, features…            Dataset
from_sql(sql, con, features, cache_dir,…            Dataset
from_file(cls, filename, info, split, i…            Dataset
from_buffer(cls, buffer, info, split, i…            Dataset
from_generator(generator, features, cac…            Dataset
from_json(path_or_paths, split, feature…            Dataset
from_pandas(cls, df, features, info, sp…            Dataset
from_parquet(path_or_paths, split, feat…            Dataset
from_polars(cls, df, features, info, sp…            Dataset
from_spark(df, split, features, keep_in…            Dataset
Press Enter to insert, Tab to replace
```

Since Hugging Face dataset must return `features` , and **all the columns should specify its individual type**. So we `import the common types from datasets` .

```
features = Features({
        'image_id': Value('int32'),
        'image': Image(),  # HF Image
        'mask': Array2D(dtype='int64', shape=(590, 445)),  # HF Array2D from torch must 
specify shape
        'class_id': Value('int32'),
        'class_name': Value('string'),
    })  # used for converting list of dicts to Dataset
```

If we have train split, evaluation split, what we need is to construct a `DatasetDict`, using following structure:



We want our dataset has following keys, so we should define a `features` to denote it :

```
features = Features({
    'image_id': Value('int32'),
    'image': Image(),  # HF Image
    'mask': Array2D(dtype='int64', shape=(None, None)),
    'class_id': Value('int32'),
    'class_name': Value('string'),
})  # used for converting list of dicts to Dataset


def convert_to_hf_dataset(dataset: FishSegmentDataSet) -> DatasetDict:
    """
    Convert our custom FishSegmentDataSet to HuggingFace DatasetDict
    """
    # Create dataset dict
    data_dict = {
        'train': [],
        'validation': [],
        'test': []
    }

    # Split dataset into train, validation, test (80%, 10%, 10%)
    total_size = len(dataset)
    train_size = int(0.8 * total_size)
    val_size = int(0.1 * total_size)
    test_size = total_size - train_size - val_size

    # since we don't want sorted data, we just do a simple split by fixed random seed
    r = 1024

    # load original data from FishSegmentDataSet
    train_set, val_set, test_set = random_split(
        dataset, [train_size, val_size, test_size],
generator=torch.Generator().manual_seed(r)
    )
    data_dict['train'] = [dataset[i] for i in train_set.indices]  # get the data from
original dataset
    data_dict['validation'] = [dataset[i] for i in val_set.indices]
    data_dict['test'] = [dataset[i] for i in test_set.indices]

    # Convert lists to Dataset objects
    for split in data_dict:
        data_dict[split] = HFDataset.from_list(data_dict[split], features=features)
    return DatasetDict(data_dict)
```

2) Using Generator for memory-friendly

But using the code above, it would cost a huge amount of memory which will cause lead program to broke, and the problem is following costly `for` loop :

```
data_dict['train'] = [dataset[i] for i in train_set.indices]  # get the data from original dataset
data_dict['validation'] = [dataset[i] for i in val_set.indices]
data_dict['test'] = [dataset[i] for i in test_set.indices]
```

So we can use a construction function by generator, and using `from_generator` is more memory-friendly :

```
def construct_hf_dataset_from_subset(subset, features, split: NamedSplit):
    def generator():
        for idx in subset.indices:
            yield subset[idx]
    return HFDataset.from_generator(generator, split=split, features=features)
```

Then the new

```
def construct_hf_dataset_from_subset(subset, features, split: NamedSplit):
    def generator():
        for idx in range(len(subset)):
            yield subset[idx]
    return HFDataset.from_generator(generator, split=split, features=features)

def convert_to_hf_dataset(dataset: FishSegmentDataSet) -> DatasetDict:
    """
    Convert our custom FishSegmentDataSet to HuggingFace DatasetDict
    """
    # Define features
    features = Features({
        'image_id': Value('int32'),
        'image': Image(),  # HF Image
        'mask': Array2D(dtype='int64', shape=(590, 445)),  # HF Array2D
        'class_id': Value('int32'),
        'class_name': Value('string'),
    })   # used for converting list of dicts to Dataset

    # Create dataset dict
    data_dict = {
        'train': [],
        'validation': [],
        'test': []
    }

    # Split dataset into train, validation, test (80%, 10%, 10%)
    total_size = len(dataset)
    train_size = int(0.8 * total_size)
    val_size = int(0.1 * total_size)
    test_size = total_size - train_size - val_size

    # since we don't want sorted data, we just do a simple split by fixed random seed
    r = 1024
    # load original data from FishSegmentDataSet
    train_set, val_set, test_set = random_split(
        dataset, [train_size, val_size, test_size],
generator=torch.Generator().manual_seed(r)
    )
    # Using generator to avoid memory leaks

    data_dict["train"] = construct_hf_dataset_from_subset(train_set, features, Split.TRAIN)
    data_dict["validation"] = construct_hf_dataset_from_subset(val_set, features,
Split.VALIDATION)
    data_dict["test"] = construct_hf_dataset_from_subset(test_set, features, Split.TEST)
    return DatasetDict(data_dict)
```

3. Load the dataset as Hugging Face Dataset

We can load it easily and save to disk :

```
def build_datasets():
    dataset = load_dataset(train_size=1.0)[0]  # load the full dataset
    print("converting to hugging face dataset...")
    # convert to DatasetDict
    hf_dataset = convert_to_hf_dataset(dataset)
    hf_dataset.save_to_disk("./fish_dataset_hf")  # save to local disk for backup
```

Then upload to hugging face :

Hint

`hf-mirror.com` may not support uploading large-scale dataset, so I used a server to upload instead.

```
def push_to_hub():
    """
    only a dataset dict or dataset can be pushed to hub  # dict may encounter error, so
load firstly as a dataset.
    """
    dataset = hf_load_dataset("./fish_dataset_hf")  # load from local disk
    dataset.push_to_hub("FriedParrot/a-large-scale-fish-dataset", private=True)  # push the
dataset to hub
```

Then just wait for the dataset to upload 😄 .

---

1. https://huggingface.co/docs/transformers/main/en/notebooks ↩
2. https://huggingface.co/docs/transformers/index ↩
3. https://huggingface.co/docs/transformers/model_doc/auto ↩
4. huggingface.co/docs/transformers/v4.56.2/en/main_classes/trainer#transformers.Trainer ↩
5. https://huggingface.co/docs/transformers/v4.56.2/en/main_classes/output#transformers.utils.ModelOutput ↩
6. huggingface.co/docs/transformers/v4.56.2/en/main_classes/trainer#transformers.TrainingArguments ↩
7. https://huggingface.co/docs/transformers/main/en/main_classes/backbones ↩
8. 1. Better Python Code Auto Completion ↩
9. https://www.kaggle.com/datasets/crowww/a-large-scale-fish-dataset/data ↩
10. 2. Dataset Loading ↩ ↩ ↩
11. https://huggingface.co/docs/transformers/main/en/main_classes/data_collator ↩
12. https://huggingface.co/docs/transformers/main/en/main_classes/data_collator ↩

13. https://github.com/huggingface/notebooks/blob/main/examples/multiple_choice.ipynb ↩
14. https://docs.pytorch.org/docs/stable/generated/torch.nn.MultiheadAttention.html ↩
15. Attention Layer ↩