Vision-Text dual encoder model training examples

> Note: This example is experimental and might not give the best possible results

The following example showcases how to train a CLIP like vision-text dual encoder model using a pre-trained vision and text encoder using the JAX/Flax backend.

Such a model can be used for natural language image search and potentially zero-shot image classification. The model is inspired by the CLIP approach, introduced by Alec Radford et al. The idea is to train a vision encoder and a text encoder jointly to project the representation of images and their captions into the same embedding space, such that the caption embeddings are located near the embeddings of the images they describe.

JAX/Flax allows you to trace pure functions and compile them into efficient, fused accelerator code on both GPU and TPU. Models written in JAX/Flax are **immutable** and updated in a purely functional way which enables simple and efficient model parallelism.

In this example we will use the vision model from CLIP as the image encoder and `roberta-base` as the text encoder. Note that one can also use the ViT model as image encoder and any other BERT or ROBERTa model as text encoder. To train the model on languages other than English one should choose a text encoder trained on the desired language and a image-text dataset in that language. One such dataset is WIT.

Let's start by creating a model repository to save the trained model and logs. Here we call the model `"clip-roberta-base"`, but you can change the model name as you like.

You can do this either directly on huggingface.co (assuming that you are logged in) or via the command line:

```
huggingface-cli repo create clip-roberta-base
```

Next we clone the model repository to add the tokenizer and model files.

```
git clone https://huggingface.co/<your-username>/clip-roberta-base
```

To ensure that all tensorboard traces will be uploaded correctly, we need to track them. You can run the following command inside your model repo to do so.

```
cd clip-roberta-base
git lfs track "*tfevents*"
```

Great, we have set up our model repository. During training, we will automatically push the training logs and model weights to the repo.

Next, let's add a symbolic link to the `run_hybrid_clip.py`.

```
export MODEL_DIR="./clip-roberta-base
ln -s ~/transformers/examples/research_projects/jax-projects/hybrid_clip/run_hybrid_clip.py
```

How to use the `FlaxHybridCLIP` model:

The `FlaxHybridCLIP` class let's you load any text and vision encoder model to create a dual encoder. Here is an example of how to load the model using pre-trained text and vision models.

```
from modeling_hybrid_clip import FlaxHybridCLIP

model = FlaxHybridCLIP.from_text_vision_pretrained("bert-base-uncased", "openai/clip-vit-bas

# save the model
model.save_pretrained("bert-clip")

# load the saved model
model = FlaxHybridCLIP.from_pretrained("bert-clip")
```

If the checkpoints are in PyTorch then one could pass `text_from_pt=True` and `vision_from_pt=True`. This will load the model PyTorch checkpoints convert them to flax and load the model.

```
model = FlaxHybridCLIP.from_text_vision_pretrained("bert-base-uncased", "openai/clip-vit-bas
```

This loads both the text and vision encoders using pre-trained weights, the projection layers are randomly initialized except for CLIP's vision model. If you use CLIP to initialize the vision model then the vision projection weights are also loaded using the pre-trained weights.

Prepare the dataset

We will use the MS-COCO dataset to train our dual encoder model. MS-COCO contains over 82,000 images, each of which has at least 5 different caption annotations. The dataset is usually used for image captioning tasks, but we can repurpose the image-caption pairs to train our dual encoder model for image search.

Download and extract the data.

It consists of two compressed folders: one with images, and the other—with associated image captions. Note that the compressed images folder is 13GB in size.

```
wget http://images.cocodataset.org/annotations/annotations_trainval2014.zip
wget http://images.cocodataset.org/zips/train2014.zip

unzip annotations_trainval2014.zip
unzip train2014.zip
```

```
mkdir coco_dataset
mv train2014 coco_dataset/
mv annotations coco_dataset/
```

Prepare dataset files and split the dataset.

```
import json
import collections

images_dir = "coco_dataset/train2014"
annotation_file = "coco_dataset/annotations/captions_train2014.json"
with open(annotation_file, "r") as f:
    annotations = json.load(f)["annotations"]

image_path_to_caption = collections.defaultdict(list)
for element in annotations:
    caption = f"{element['caption'].lower().rstrip('.')}"
    image_path = images_dir + "/COCO_train2014_" + "%012d.jpg" % (element["image_id"])
    image_path_to_caption[image_path].append(caption)

lines = []
for image_path, captions in image_path_to_caption.items():
    lines.append(json.dumps({"image_path": image_path, "captions": captions}))

train_lines = lines[:-8000]
valid_line = lines[-8000:]
with open("coco_dataset/train_dataset.json", "w") as f:
    f.write("\n".join(train_lines))

with open("coco_dataset/valid_dataset.json", "w") as f:
    f.write("\n".join(valid_line))
```

> Note: The data loading and processing part of this script can still be improved for maximum performance. In particular one should decode the images beforehand and use those instead decoding them each time. If the dataset is small or if you have huge disk space the you could also pre-process all the dataset beforehand and then use it.

Train the model

Next we can run the example script to train the model:

```
python run_hybrid_clip.py \
    --output_dir ${MODEL_DIR} \
    --text_model_name_or_path="roberta-base" \
    --vision_model_name_or_path="openai/clip-vit-base-patch32" \
```

```
--tokenizer_name="roberta-base" \
--train_file="coco_dataset/train_dataset.json" \
--validation_file="coco_dataset/validation_dataset.json" \
--do_train --do_eval \
--num_train_epochs="40" --max_seq_length 96 \
--per_device_train_batch_size="64" \
--per_device_eval_batch_size="64" \
--learning_rate="5e-5" --warmup_steps="0" --weight_decay 0.1 \
--overwrite_output_dir \
--preprocessing_num_workers 32 \
--push_to_hub
```

This should finish in ~1h50 mins with min validation loss 2.43. Training statistics can be accessed on tfhub.de