```
!pip install kaggle

from google.colab import files



# List available datasets.
!kaggle datasets list

# Download the dataset
!kaggle datasets download -d topkek69/captcha




!unzip captcha -d extracted_data


!pip install -q transformers


!pip install -q evaluate jiwer
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 84.0/84.0 kB 2.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 480.6/480.6 kB 20.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 116.3/116.3 kB 8.8 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 179.3/179.3 kB 13.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 134.8/134.8 kB 7.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.1/3.1 MB 59.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 194.1/194.1 kB 10.6 MB/s eta 0:00:00
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the sou
gcsfs 2024.10.0 requires fsspec==2024.10.0, but you have fsspec 2024.9.0 which is incompatible.
```

```
import torch
from torch.utils.data import Dataset
from PIL import Image
import os
import random
import matplotlib.pyplot as plt
from evaluate import load
from transformers import AdamW
from tqdm.notebook import tqdm
from transformers import TrOCRProcessor, VisionEncoderDecoderModel, VisionEncoderDecoderConfig
import warnings

warnings.filterwarnings("ignore")
os.environ["TOKENIZERS_PARALLELISM"] = 'false'


# Define the paths
dataset_dir = '/content/extracted_data'
train_dir = os.path.join(dataset_dir, 'train')
valid_dir = os.path.join(dataset_dir, 'valid')
test_dir = os.path.join(dataset_dir, 'test')


class CaptchaDataset(Dataset):
    def __init__(self, root_dir, processor, max_target_length=10):
        self.root_dir = root_dir
        self.data = []
        self.processor = processor
        self.max_target_length = max_target_length

        for file_name in os.listdir(self.root_dir):
            self.data.append((os.path.join(root_dir, file_name), file_name.removesuffix('.png')))

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        # get file name + text
        file_path, text = self.data[idx]

        # prepare image (i.e. resize + normalize)
        image = Image.open(file_path).convert("RGB")
        pixel_values = self.processor(image, return_tensors="pt").pixel_values
```

```
        # add labels (input_ids) by encoding the text
        labels = self.processor.tokenizer(text,
                                          padding="max_length",
                                          max_length=self.max_target_length).input_ids
        # important: make sure that PAD tokens are ignored by the loss function
        labels = [label if label != self.processor.tokenizer.pad_token_id else -100 for label in labels]

        encoding = {"pixel_values": pixel_values.squeeze(), "labels": torch.tensor(labels)}
        return encoding


processor = TrOCRProcessor.from_pretrained("microsoft/trocr-small-handwritten")
train_dataset = CaptchaDataset(root_dir=train_dir, processor=processor)
valid_dataset = CaptchaDataset(root_dir=valid_dir, processor=processor)
test_dataset = CaptchaDataset(root_dir=test_dir, processor=processor)
```

```
preprocessor_config.json: 100%  272/272 [00:00<00:00, 15.3kB/s]
tokenizer_config.json: 100%  327/327 [00:00<00:00, 5.43kB/s]
sentencepiece.bpe.model: 100%  1.36M/1.36M [00:00<00:00, 9.69MB/s]
special_tokens_map.json: 100%  238/238 [00:00<00:00, 6.36kB/s]
```

```
print("Number of training examples:", len(train_dataset))
print("Number of validation examples:", len(valid_dataset))
print("Number of testing examples:", len(test_dataset))
```

```
Number of training examples: 6839
Number of validation examples: 684
Number of testing examples: 683
```

```
encoding = train_dataset[0]
for k,v in encoding.items():
  print(k, v.shape)
```

```
pixel_values torch.Size([3, 384, 384])
labels torch.Size([10])
```

```
image = Image.open(os.path.join(train_dir, os.listdir(train_dir)[0])).convert("RGB")
image
```

```
os.listdir(train_dir)[0].removesuffix('.png')
```

```
'mbdjyfr'
```

```
labels = encoding['labels']
labels[labels == -100] = processor.tokenizer.pad_token_id
label_str = processor.decode(labels, skip_special_tokens=True)
print(label_str)
```

```
mbdjyfr
```

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(train_dataset, batch_size=16, shuffle=True)
valid_dataloader = DataLoader(valid_dataset, batch_size=16)


from transformers import VisionEncoderDecoderModel

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

model = VisionEncoderDecoderModel.from_pretrained("microsoft/trocr-small-handwritten")
model.to(device)
```

config.json: 100% 4.21k/4.21k [00:00<00:00, 57.4kB/s]

pytorch_model.bin: 100% 246M/246M [00:01<00:00, 196MB/s]

```
Some weights of VisionEncoderDecoderModel were not initialized from the model checkpoint at microsoft/trocr-small-handwritten and
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

generation_config.json: 100% 190/190 [00:00<00:00, 4.52kB/s]

```
VisionEncoderDecoderModel(
  (encoder): DeiTModel(
    (embeddings): DeiTEmbeddings(
      (patch_embeddings): DeiTPatchEmbeddings(
        (projection): Conv2d(3, 384, kernel_size=(16, 16), stride=(16, 16))
      )
      (dropout): Dropout(p=0.0, inplace=False)
    )
    (encoder): DeiTEncoder(
      (layer): ModuleList(
        (0-11): 12 x DeiTLayer(
          (attention): DeiTAttention(
            (attention): DeiTSelfAttention(
              (query): Linear(in_features=384, out_features=384, bias=True)
              (key): Linear(in_features=384, out_features=384, bias=True)
              (value): Linear(in_features=384, out_features=384, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
            (output): DeiTSelfOutput(
              (dense): Linear(in_features=384, out_features=384, bias=True)
              (dropout): Dropout(p=0.0, inplace=False)
            )
          )
          (intermediate): DeiTIntermediate(
            (dense): Linear(in_features=384, out_features=1536, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): DeiTOutput(
            (dense): Linear(in_features=1536, out_features=384, bias=True)
            (dropout): Dropout(p=0.0, inplace=False)
          )
          (layernorm_before): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
          (layernorm_after): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
        )
      )
    )
    (layernorm): LayerNorm((384,), eps=1e-12, elementwise_affine=True)
    (pooler): DeiTPooler(
      (dense): Linear(in_features=384, out_features=384, bias=True)
      (activation): Tanh()
    )
  )
  (decoder): TrOCRForCausalLM(
    (model): TrOCRDecoderWrapper(
      (decoder): TrOCRDecoder(
        (embed_tokens): TrOCRScaledWordEmbedding(64044, 256, padding_idx=1)
        (embed_positions): TrOCRLearnedPositionalEmbedding(514, 256)
        (layernorm_embedding): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
        (layers): ModuleList(
          (0-5): 6 x TrOCRDecoderLayer(
            (self_attn): TrOCRAttention(
              (k_proj): Linear(in_features=256, out_features=256, bias=True)
              (v_proj): Linear(in_features=256, out_features=256, bias=True)
              (q_proj): Linear(in_features=256, out_features=256, bias=True)
              (out_proj): Linear(in_features=256, out_features=256, bias=True)
            )
            (activation_fn): ReLU()
            (self_attn_layer_norm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
            (encoder_attn): TrOCRAttention(
              (k_proj): Linear(in_features=384, out_features=256, bias=True)
              (v_proj): Linear(in_features=384, out_features=256, bias=True)
              (q_proj): Linear(in_features=256, out_features=256, bias=True)
              (out_proj): Linear(in_features=256, out_features=256, bias=True)
            )
            (encoder_attn_layer_norm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
            (fc1): Linear(in_features=256, out_features=1024, bias=True)
            (fc2): Linear(in_features=1024, out_features=256, bias=True)
            (final_layer_norm): LayerNorm((256,), eps=1e-05, elementwise_affine=True)
          )
        )
      )
    )
    (output_projection): Linear(in_features=256, out_features=64044, bias=False)
  )
)
```

```
# set special tokens used for creating the decoder_input_ids from the labels
model.config.decoder_start_token_id = processor.tokenizer.cls_token_id
model.config.pad_token_id = processor.tokenizer.pad_token_id
```

```
# make sure vocab size is set correctly
model.config.vocab_size = model.config.decoder.vocab_size

# set beam search parameters
model.config.eos_token_id = processor.tokenizer.sep_token_id
model.config.max_length = 10
model.config.length_penalty = 0
model.config.early_stopping = True
model.config.num_beams = 2


cer_metric = load("cer")

def compute_cer(pred_ids, label_ids):
    pred_str = processor.batch_decode(pred_ids, skip_special_tokens=True)
    label_ids[label_ids == -100] = processor.tokenizer.pad_token_id
    label_str = processor.batch_decode(label_ids, skip_special_tokens=True)

    cer = cer_metric.compute(predictions=pred_str, references=label_str)

    return cer
```

```
Downloading builder script: 100%  5.60k/5.60k [00:00<00:00, 99.6kB/s]
```

```
from transformers import AdamW

optimizer = AdamW(model.parameters(), lr=5e-5)

for epoch in range(5):  # loop over the dataset multiple times
   # train
   model.train()
   train_loss = 0.0
   for batch in tqdm(train_dataloader):
      # get the inputs
      for k,v in batch.items():
        batch[k] = v.to(device)

      # forward + backward + optimize
      outputs = model(**batch)
      loss = outputs.loss
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

      train_loss += loss.item()

   print(f"Loss after epoch {epoch}:", train_loss/len(train_dataloader))

   # evaluate
   model.eval()
   valid_cer = 0.0
   with torch.no_grad():
     for batch in tqdm(valid_dataloader):
       # run batch generation
       outputs = model.generate(batch["pixel_values"].to(device))
       # compute metrics
       cer = compute_cer(pred_ids=outputs, label_ids=batch["labels"])
       valid_cer += cer

   print("Validation CER:", valid_cer / len(valid_dataloader))

model.save_pretrained("OCR-model")
```

```
100%                                          428/428 [05:23<00:00,  1.58it/s]
Loss after epoch 0: 3.840682171215521
100%                                          43/43 [00:25<00:00,  1.86it/s]
Validation CER: 0.9502093918427432
100%                                          428/428 [05:18<00:00,  1.59it/s]
Loss after epoch 1: 3.468869754087145
100%                                          43/43 [00:24<00:00,  1.96it/s]
Validation CER: 0.9436469519247762
100%                                          428/428 [05:18<00:00,  1.55it/s]
Loss after epoch 2: 3.403738567205233
100%                                          43/43 [00:24<00:00,  1.77it/s]
Validation CER: 0.9285004632023405
100%                                          428/428 [05:17<00:00,  1.59it/s]
Loss after epoch 3: 1.6910868330954392
100%                                          43/43 [00:24<00:00,  1.92it/s]
Validation CER: 0.13396992050187131
100%                                          428/428 [05:18<00:00,  1.60it/s]
Loss after epoch 4: 0.44982808877096
100%                                          43/43 [00:24<00:00,  1.96it/s]
Some non-default generation parameters are set in the model config. These should go into a GenerationConfig file (https://huggingfa
Non-default generation parameters: {'max_length': 10, 'early_stopping': True, 'num_beams': 2, 'length_penalty': 0}
Validation CER: 0.0829151283532357
```

```
from google.colab import drive
drive.mount('/content/drive')

# ... your existing code ...

model.save_pretrained("/content/drive/MyDrive/OCR-model") #Save to your drive
```

```
Some non-default generation parameters are set in the model config. These should go into a GenerationConfig file (https://huggingfa
Non-default generation parameters: {'max_length': 10, 'early_stopping': True, 'num_beams': 2, 'length_penalty': 0}
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# loading model and config from pretrained folder
encoder_decoder_config = VisionEncoderDecoderConfig.from_pretrained("OCR-model")
OCR_model = VisionEncoderDecoderModel.from_pretrained("OCR-model", config=encoder_decoder_config).to(device)
```

TEST

```
def random_infer():
    # Get a random image path from the test folder
    image_files = [f for f in os.listdir('/content/extracted_data/test') if f.endswith('.png')]
    random_image_file = random.choice(image_files)
    print(random_image_file)

    image_path = os.path.join('/content/extracted_data/test', random_image_file)
    image = Image.open(image_path)
    plt.imshow(image)

    # Process the image
    pixel_values = processor(image, return_tensors="pt").pixel_values.to(device)
    #print(f"Pixel values shape: {pixel_values.shape}")

    # Generate text from the image
    generated_ids = OCR_model.generate(pixel_values)
    generated_text = processor.batch_decode(generated_ids, skip_special_tokens=True)[0]
    #print(f"Generated text: {generated_text}")

    return generated_text


random_infer()
```