```
# image_captioning.py
import os
import math
import json
import random
from collections import Counter
from pathlib import Path
from typing import List

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, models
from PIL import Image
from tqdm import tqdm
import nltk

# -------------------------
# Hyperparams (tweakable)
# -------------------------
EMBED_SIZE = 512    # embedding dim for tokens
D_MODEL = 512       # transformer d_model (should match EMBED_SIZE)
NUM_HEADS = 8
NUM_LAYERS = 3
FFN_DIM = 2048
MAX_LEN = 40        # max caption length including <bos>/<eos>
BATCH_SIZE = 32
```

```
LR = 1e-4
NUM_EPOCHS = 20
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
PAD_IDX = 0
BOS_TOKEN = "<bos>"
EOS_TOKEN = "<eos>"
UNK_TOKEN = "<unk>"
MIN_WORD_FREQ = 5


# -------------------------
# Vocabulary
# -------------------------
class Vocabulary:
  def __init__(self, min_freq=MIN_WORD_FREQ):
    self.word2idx = {}
    self.idx2word = {}
    self.counter = Counter()
    self.min_freq = min_freq
    self.specials = [PAD_IDX, BOS_TOKEN, EOS_TOKEN, UNK_TOKEN]

  def build(self, sentences: List[str]):
    for s in sentences:
      tokens = self._tokenize(s)
      self.counter.update(tokens)
    # specials
    self.word2idx = {
      "<pad>": 0,
      BOS_TOKEN: 1,
```

```
        EOS_TOKEN: 2,
        UNK_TOKEN: 3
    }
    idx = len(self.word2idx)
    for word, freq in self.counter.most_common():
        if freq < self.min_freq:
            break
        if word not in self.word2idx:
            self.word2idx[word] = idx
            idx += 1
    self.idx2word = {i: w for w, i in self.word2idx.items()}

def _tokenize(self, s: str):
    # simple tokenizer: lower + nltk word_tokenize (download punkt beforehand)
    return nltk.word_tokenize(s.lower())

def encode(self, s: str, max_len=MAX_LEN):
    tokens = [BOS_TOKEN] + self._tokenize(s) + [EOS_TOKEN]
    ids = []
    for t in tokens[:max_len]:
        if t in self.word2idx:
            ids.append(self.word2idx[t])
        else:
            ids.append(self.word2idx[UNK_TOKEN])
    # pad
    while len(ids) < max_len:
        ids.append(self.word2idx["<pad>"])
    return ids
```

```
    def decode(self, ids: List[int]):
        words = []
        for i in ids:
            w = self.idx2word.get(i, UNK_TOKEN)
            if w == EOS_TOKEN:
                break
            if w not in ("<pad>", BOS_TOKEN):
                words.append(w)
        return " ".join(words)

    def __len__(self):
        return len(self.word2idx)

# -------------------------
# Dataset (COCO-style simple)
# expects a json file list: [{"image": "path/to.jpg", "caption": "a cat ..."}, ...]
# -------------------------
class CaptionDataset(Dataset):
    def __init__(self, data_json, vocab: Vocabulary, transform=None, max_len=MAX_LEN):
        with open(data_json, "r", encoding="utf-8") as f:
            self.data = json.load(f)
        self.vocab = vocab
        self.transform = transform or transforms.Compose([
            transforms.Resize((224,224)),
            transforms.ToTensor(),
            transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
        ])
```

```
        self.max_len = max_len

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

    def __getitem__(self, idx):
        item = self.data[idx]
        img_path = item["image"]
        caption = item["caption"]
        img = Image.open(img_path).convert("RGB")
        img = self.transform(img)
        caption_ids = torch.tensor(self.vocab.encode(caption, max_len=self.max_len),
dtype=torch.long)
        # prepare input (without last token) and target (without first token) for teacher
forcing
        input_ids = caption_ids[:-1]  # remove last (maybe <pad> or <eos>)
        target_ids = caption_ids[1:]  # remove first (<bos>)
        return img, input_ids, target_ids

# -------------------------
# Encoder: ResNet-50
# returns flattened features shape (batch, seq_len, d_model)
# We'll use conv feature map (C x H x W) and project to d_model
# -------------------------
class EncoderCNN(nn.Module):
    def __init__(self, d_model=D_MODEL, pretrained=True):
        super().__init__()
        resnet = models.resnet50(pretrained=pretrained)
        # remove fully connected and avgpool
```

```
        modules = list(resnet.children())[:-2]  # keep until last conv layer -> output shape (B, 2048, H/32, W/32)
        self.backbone = nn.Sequential(*modules)
        self.conv_dim = 2048
        self.proj = nn.Linear(self.conv_dim, d_model)

    def forward(self, images):
        """
        images: (B,3,H,W)
        returns: (B, seq_len, d_model)
        """
        feat = self.backbone(images) # B x 2048 x h x w
        B, C, H, W = feat.shape
        feat = feat.view(B, C, H*W).permute(0, 2, 1)  # B x (H*W) x C
        feat = self.proj(feat)  # B x seq_len x d_model
        return feat

# -------------------------
# Positional encoding
# -------------------------
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1).float()
        div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
```

```
        self.register_buffer('pe', pe.unsqueeze(0))  # 1 x max_len x d_model


    def forward(self, x):
        # x: B x seq_len x d_model
        seq_len = x.size(1)
        x = x + self.pe[:, :seq_len, :].to(x.device)
        return x


# -------------------------
# Transformer-based captioning model
# -------------------------
class CaptionTransformer(nn.Module):
    def __init__(self, vocab_size, d_model=D_MODEL, nhead=NUM_HEADS, num_layers=NUM_LAYERS, dim_feedforward=FFN_DIM, max_len=MAX_LEN, dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.token_embed = nn.Embedding(vocab_size, d_model, padding_idx=PAD_IDX)
        self.pos_enc = PositionalEncoding(d_model, max_len=max_len)
        decoder_layer = nn.TransformerDecoderLayer(d_model=d_model, nhead=nhead, dim_feedforward=dim_feedforward, dropout=dropout)
        self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=num_layers)
        self.output_fc = nn.Linear(d_model, vocab_size)
        self.dropout = nn.Dropout(dropout)


    def forward(self, memory, tgt_ids, memory_key_padding_mask=None, tgt_key_padding_mask=None):
        """
```

```
        memory: (B, mem_len, d_model)  -- from encoder
        tgt_ids: (B, tgt_len)  -- input token ids (with <bos>)
        returns logits: (B, tgt_len, vocab_size)
        """
        # prepare tgt embeddings
        tgt = self.token_embed(tgt_ids) * math.sqrt(self.d_model)  # B x tgt_len x d_model
        tgt = self.pos_enc(tgt)
        # PyTorch Transformer expects (seq_len, batch, d_model)
        tgt = tgt.permute(1,0,2)
        memory = memory.permute(1,0,2)  # mem_len, B, d_model

        # trg_mask for subsequent positions (causal)
        tgt_mask = nn.Transformer.generate_square_subsequent_mask(tgt.size(0)).to(tgt.device)

        out = self.transformer_decoder(tgt, memory, tgt_mask=tgt_mask, memory_key_padding_mask=memory_key_padding_mask, tgt_key_padding_mask=tgt_key_padding_mask)
        out = out.permute(1,0,2)  # B x tgt_len x d_model
        logits = self.output_fc(out)
        return logits

# -------------------------
# Utilities for masks
# -------------------------
def make_src_key_padding_mask(memory, pad=None):
    # For image memory there is no padding (all positions valid) typically; return None
    return None
```

```
def make_tgt_key_padding_mask(tgt_ids, pad_idx=PAD_IDX):
    # tgt_ids: B x tgt_len
    return (tgt_ids == pad_idx)  # bool mask


# -------------------------
# Training loop and helpers
# -------------------------
def train_one_epoch(encoder, decoder, dataloader, optimizer, criterion, epoch):
    encoder.train()
    decoder.train()
    total_loss = 0.0
    for imgs, input_ids, target_ids in tqdm(dataloader, desc=f"Epoch {epoch}"):
        imgs = imgs.to(DEVICE)
        input_ids = input_ids.to(DEVICE)
        target_ids = target_ids.to(DEVICE)
        # encoder
        memory = encoder(imgs)  # B x mem_len x d_model
        tgt_mask = make_tgt_key_padding_mask(input_ids)
        memory_mask = None


        logits = decoder(memory, input_ids, memory_key_padding_mask=memory_mask,
tgt_key_padding_mask=tgt_mask)  # B x tgt_len x V
        # compute loss: flatten
        B, T, V = logits.shape
        logits_flat = logits.view(B*T, V)
        targets_flat = target_ids.view(B*T)
        loss = criterion(logits_flat, targets_flat)
```

```
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(list(encoder.parameters()) +
list(decoder.parameters()), 1.0)
        optimizer.step()
        total_loss += loss.item()
    avg = total_loss / len(dataloader)
    return avg


def evaluate_bleu(encoder, decoder, dataloader, vocab):
    encoder.eval()
    decoder.eval()
    references = []
    hypotheses = []
    with torch.no_grad():
        for imgs, input_ids, target_ids in tqdm(dataloader, desc="Eval"):
            imgs = imgs.to(DEVICE)
            memory = encoder(imgs)
            B = imgs.size(0)
            # Greedy decoding
            generated = greedy_decode(decoder, memory, vocab, max_len=MAX_LEN)
            for i in range(B):
                # target: convert target_ids[i] into text (strip pads)
                tgt_ids = target_ids[i].tolist()
                references.append([vocab.decode(tgt_ids)])  # list of references per sample
                hypotheses.append(generated[i])
    # compute BLEU
    # nltk expects tokenized reference/hypothesis lists
```

```
    bleu_scores = [nltk.translate.bleu_score.sentence_bleu([ref.split()], hyp.split(), weights=(1,0,0,0)) for ref,hyp in zip(references, hypotheses)]
    return sum(bleu_scores)/len(bleu_scores)


# -------------------------
# Greedy decode
# -------------------------
def greedy_decode(decoder, memory, vocab, max_len=MAX_LEN):
    device = memory.device
    B = memory.size(0)
    generated_ids = torch.full((B, 1), vocab.word2idx[BOS_TOKEN], dtype=torch.long, device=device)  # B x 1
    ended = torch.zeros(B, dtype=torch.bool, device=device)
    out_sentences = [None]*B

    for step in range(max_len-1):
        logits = decoder(memory, generated_ids)  # B x seq_len x V
        next_token_logits = logits[:, -1, :]  # B x V
        next_ids = torch.argmax(next_token_logits, dim=-1).unsqueeze(1)  # B x 1
        generated_ids = torch.cat([generated_ids, next_ids], dim=1)
        # check eos
        for i in range(B):
            if not ended[i] and next_ids[i,0].item() == vocab.word2idx[EOS_TOKEN]:
                ended[i] = True
                # decode full sequence excluding BOS and after EOS
                seq = generated_ids[i].tolist()
                out_sentences[i] = vocab.decode(seq[1:])  # exclude BOS
        if ended.all():
            break
```

```
    # for ones not ended, decode
    for i in range(B):
        if out_sentences[i] is None:
            seq = generated_ids[i].tolist()
            out_sentences[i] = vocab.decode(seq[1:])
    return out_sentences

# -------------------------
# Main: training entrypoint
# -------------------------
def main(data_json_train, data_json_val, model_save_dir="models"):
    nltk.download('punkt')

    # load captions to build vocab
    with open(data_json_train, "r", encoding="utf-8") as f:
        train_data = json.load(f)
    captions = [d["caption"] for d in train_data]
    vocab = Vocabulary(min_freq=MIN_WORD_FREQ)
    vocab.build(captions)
    print(f"Vocab size: {len(vocab)}")

    transform = transforms.Compose([
        transforms.Resize((224,224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485,0.456,0.406], std=[0.229,0.224,0.225])
    ])

    train_ds = CaptionDataset(data_json_train, vocab, transform=transform)
```

```
    val_ds = CaptionDataset(data_json_val, vocab, transform=transform)
    train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE, shuffle=True, num_workers=4, pin_memory=True, drop_last=True)
    val_loader = DataLoader(val_ds, batch_size=BATCH_SIZE, shuffle=False, num_workers=4, pin_memory=True)

    encoder = EncoderCNN(d_model=D_MODEL, pretrained=True).to(DEVICE)
    decoder = CaptionTransformer(vocab_size=len(vocab), d_model=D_MODEL, nhead=NUM_HEADS, num_layers=NUM_LAYERS, dim_feedforward=FFN_DIM).to(DEVICE)

    params = list(encoder.proj.parameters()) + list(decoder.parameters()) + list(decoder.token_embed.parameters())
    optimizer = torch.optim.Adam(params, lr=LR)
    criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)

    os.makedirs(model_save_dir, exist_ok=True)

    for epoch in range(1, NUM_EPOCHS+1):
        train_loss = train_one_epoch(encoder, decoder, train_loader, optimizer, criterion, epoch)
        print(f"Epoch {epoch} train loss: {train_loss:.4f}")
        # small eval
        bleu = evaluate_bleu(encoder, decoder, val_loader, vocab)
        print(f"Epoch {epoch} BLEU-1 (greedy): {bleu:.4f}")
        # save
        torch.save({
            "encoder": encoder.state_dict(),
            "decoder": decoder.state_dict(),
            "vocab": vocab.word2idx
```

```
        }, os.path.join(model_save_dir, f"model_epoch{epoch}.pth"))
    print("Training finished")


if __name__ == "__main__":
    # Example usage:
    # Prepare train.json and val.json with list of {"image": "/abs/path/to/image.jpg", "caption":"a caption"} entries
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("--train_json", required=True, help="train json file")
    parser.add_argument("--val_json", required=True, help="val json file")
    parser.add_argument("--save_dir", default="models")
    args = parser.parse_args()
    main(args.train_json, args.val_json, model_save_dir=args.save_dir)
```