```
import os
import torch
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
from collections import Counter
import random
```

```
def load_conllu(file_path):
    # Return list of sentences, each sentence is list of (word, upos)
    sentences = []
    cur = []
    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
            if line == "" or line.startswith("#"):
                if cur:
                    sentences.append(cur)
                    cur = []
                continue
            parts = line.split('\t')
            if len(parts) < 5: continue
            # column 2 is FORM (index 1), column 4 is UPOS (index 3)
            form = parts[1]
            upos = parts[3]
            # skip multiword lines or comments handled above; also ignore if id contains '-'
            if '-' in parts[0] or '.' in parts[0]:
                continue
            cur.append((form, upos)) # last ln
        if cur:
            sentences.append(cur)
    return sentences
```

```
def build_vocabs(train_sentences, min_freq=1):
    word_counter = Counter()
    tag_set = set()
    for sent in train_sentences:
        for w, t in sent:
            word_counter[w] += 1
            tag_set.add(t)
    # special tokens
    word_to_ix = {'<PAD>':0, '<UNK>':1}
    for w, cnt in word_counter.items():
        if cnt >= min_freq and w not in word_to_ix:
            word_to_ix[w] = len(word_to_ix)
    tag_to_ix = {'<PAD>':0}
    for t in sorted(tag_set):
        tag_to_ix[t] = len(tag_to_ix)
    return word_to_ix, tag_to_ix
```

```
# ---------- Dataset ----------
class POSDataset(Dataset):
    def __init__(self, sentences, word_to_ix, tag_to_ix):
        self.sentences = sentences
        self.word_to_ix = word_to_ix
        self.tag_to_ix = tag_to_ix

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

    def __getitem__(self, idx):
        sent = self.sentences[idx]
        words = [w for w,t in sent]
        tags = [t for w,t in sent]
        word_indices = [ self.word_to_ix.get(w, self.word_to_ix['<UNK>']) for w in words ]
        tag_indices = [ self.tag_to_ix[t] for t in tags ]
        return torch.tensor(word_indices, dtype=torch.long), torch.tensor(tag_indices, dtype=torch.long)
```

```
# collate fn for variable-length sequences
def collate_fn(batch):
    word_seqs = [item[0] for item in batch]
    tag_seqs = [item[1] for item in batch]
```

```
    lengths = [len(s) for s in word_seqs]
    words_padded = pad_sequence(word_seqs, batch_first=True, padding_value=0)  # pad word idx 0
    tags_padded = pad_sequence(tag_seqs, batch_first=True, padding_value=0)    # pad tag idx 0 (ignore_index)
    return words_padded, tags_padded, torch.tensor(lengths, dtype=torch.long)
```

```
# ---------- Model ----------
class SimpleRNNForTokenClassification(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, num_tags, padding_idx=0, num_layers=1, bidirectional=False):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=padding_idx)
        self.rnn = nn.RNN(input_size=embed_dim,
                          hidden_size=hidden_dim,
                          num_layers=num_layers,
                          batch_first=True,
                          bidirectional=bidirectional)
        rnn_out_dim = hidden_dim * (2 if bidirectional else 1)
        self.fc = nn.Linear(rnn_out_dim, num_tags)

    def forward(self, x):
        # x: (batch, seq_len)
        emb = self.embedding(x)                      # (batch, seq_len, embed_dim)
        rnn_out, _ = self.rnn(emb)                   # (batch, seq_len, hidden)
        logits = self.fc(rnn_out)                    # (batch, seq_len, num_tags)
        return logits
```

```
# ---------- Training and Evaluation ----------
def compute_accuracy(preds, labels, pad_idx=0):
    # preds, labels: flattened 1D tensors
    mask = labels != pad_idx
    if mask.sum().item() == 0:
        return 0.0
    correct = (preds[mask] == labels[mask]).sum().item()
    total = mask.sum().item()
    return correct / total

def evaluate(model, dataloader, device, tag_pad_idx=0):
    model.eval()
    all_acc = []
    total_loss = 0.0
    criterion = nn.CrossEntropyLoss(ignore_index=tag_pad_idx)
    with torch.no_grad():
        for words, tags, lengths in dataloader:
            words = words.to(device)
            tags = tags.to(device)
            logits = model(words)  # (b, seq, num_tags)
            b, seq, num_tags = logits.size()
            logits_flat = logits.view(-1, num_tags)
            tags_flat = tags.view(-1)
            loss = criterion(logits_flat, tags_flat)
            total_loss += loss.item() * words.size(0)
            preds = torch.argmax(logits, dim=-1).view(-1)
            acc = compute_accuracy(preds, tags_flat, pad_idx=tag_pad_idx)
            all_acc.append((acc, words.size(0)))
    if len(all_acc) == 0:
        return 0.0, 0.0
    # weighted average accuracy
    weighted_acc = sum(a * n for a,n in all_acc) / sum(n for a,n in all_acc)
    avg_loss = total_loss / sum(batch[0].size(0) for batch in dataloader)
    return weighted_acc, avg_loss

def train(model, train_loader, dev_loader, device, tag_pad_idx=0, epochs=5, lr=1e-3):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss(ignore_index=tag_pad_idx)
    model.to(device)
    for epoch in range(1, epochs+1):
        model.train()
        running_loss = 0.0
        total_batches = 0
        for words, tags, lengths in train_loader:
            words = words.to(device)
            tags = tags.to(device)
            logits = model(words)  # (b, seq, num_tags)
            b, seq, num_tags = logits.size()
            logits_flat = logits.view(-1, num_tags)
            tags_flat = tags.view(-1)
```

```
            loss = criterion(logits_flat, tags_flat)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            total_batches += 1
        avg_train_loss = running_loss / max(1, total_batches)
        train_acc, _ = evaluate(model, train_loader, device, tag_pad_idx=tag_pad_idx)
        dev_acc, dev_loss = evaluate(model, dev_loader, device, tag_pad_idx=tag_pad_idx)
        print(f"Epoch {epoch}/{epochs} — train_loss: {avg_train_loss:.4f} | train_acc: {train_acc:.4f} | dev_loss: {dev_loss:.4f
```

```
# ---------- prediction helper ----------
def predict_sentence(model, sentence, word_to_ix, ix_to_tag, device):
    model.eval()
    tokens = sentence.strip().split()
    indices = [ word_to_ix.get(w, word_to_ix['<UNK>']) for w in tokens ]
    x = torch.tensor(indices, dtype=torch.long).unsqueeze(0).to(device)
    with torch.no_grad():
        logits = model(x)  # (1, seq, num_tags)
        preds = torch.argmax(logits, dim=-1).squeeze(0).cpu().tolist()
    tagged = [(tokens[i], ix_to_tag[preds[i]]) for i in range(len(tokens))]
    return tagged
```

```
# ---------- main runnable ----------
def main():
    # paths (adjust if needed)
    train_path = "/content/en_ewt-ud-train.conllu"
    dev_path   = "/content/en_ewt-ud-dev.conllu"
    assert os.path.exists(train_path), f"{train_path} not found"
    assert os.path.exists(dev_path), f"{dev_path} not found"

    # load
    train_sents = load_conllu(train_path)
    dev_sents = load_conllu(dev_path)
    print(f"Loaded {len(train_sents)} train sentences, {len(dev_sents)} dev sentences")

    # build vocabs
    word_to_ix, tag_to_ix = build_vocabs(train_sents, min_freq=1)
    ix_to_tag = {v:k for k,v in tag_to_ix.items()}
    print("Vocab sizes: words =", len(word_to_ix), ", tags =", len(tag_to_ix))

    # datasets and loaders
    train_dataset = POSDataset(train_sents, word_to_ix, tag_to_ix)
    dev_dataset = POSDataset(dev_sents, word_to_ix, tag_to_ix)
    BATCH_SIZE = 64
    train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, collate_fn=collate_fn)
    dev_loader = DataLoader(dev_dataset, batch_size=BATCH_SIZE, shuffle=False, collate_fn=collate_fn)

    # model hyperparams
    vocab_size = len(word_to_ix)
    num_tags = len(tag_to_ix)
    embed_dim = 100
    hidden_dim = 128
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = SimpleRNNForTokenClassification(vocab_size=vocab_size, embed_dim=embed_dim, hidden_dim=hidden_dim, num_tags=num_tag

    # train
    train(model, train_loader, dev_loader, device, tag_pad_idx=tag_to_ix['<PAD>'], epochs=6, lr=5e-4)

    # final evaluation
    dev_acc, _ = evaluate(model, dev_loader, device, tag_pad_idx=tag_to_ix['<PAD>'])
    print(f"Final dev accuracy: {dev_acc:.4f}")

    examples = [
        "I love NLP",
        "Im about to blow",
        "Im a Lion Piza Chicken",
        "You diggn in me"
    ]

    for example in examples:
        tagged = predict_sentence(model, example, word_to_ix, ix_to_tag, device)
        print(f"Input: {example}")
        print("Prediction:", tagged)
        print("-" * 40)
```

```
if __name__ == "__main__":
    # reproducibility
    torch.manual_seed(42)
    random.seed(42)
    main()
```

```
Loaded 12544 train sentences, 2001 dev sentences
Vocab sizes: words = 19675 , tags = 18
Epoch 1/6 — train_loss: 1.6433 | train_acc: 0.6513 | dev_loss: 1.2147 | dev_acc: 0.6279
Epoch 2/6 — train_loss: 0.9851 | train_acc: 0.7318 | dev_loss: 0.9616 | dev_acc: 0.7009
Epoch 3/6 — train_loss: 0.7871 | train_acc: 0.7765 | dev_loss: 0.8441 | dev_acc: 0.7449
Epoch 4/6 — train_loss: 0.6689 | train_acc: 0.8068 | dev_loss: 0.7765 | dev_acc: 0.7684
Epoch 5/6 — train_loss: 0.5836 | train_acc: 0.8298 | dev_loss: 0.7324 | dev_acc: 0.7846
Epoch 6/6 — train_loss: 0.5168 | train_acc: 0.8486 | dev_loss: 0.7073 | dev_acc: 0.7979
Final dev accuracy: 0.7979
Input: I love NLP
Prediction: [('I', 'PRON'), ('love', 'VERB'), ('NLP', 'VERB')]
----------------------------------------
Input: Im about to blow
Prediction: [('Im', 'VERB'), ('about', 'ADP'), ('to', 'ADP'), ('blow', 'VERB')]
----------------------------------------
Input: Im a Lion Piza Chicken
Prediction: [('Im', 'VERB'), ('a', 'DET'), ('Lion', 'NOUN'), ('Piza', 'NOUN'), ('Chicken', 'NOUN')]
----------------------------------------
Input: You diggn in me
Prediction: [('You', 'PRON'), ('diggn', 'VERB'), ('in', 'ADP'), ('me', 'PRON')]
----------------------------------------
```