```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from torch.nn.utils.rnn import pad_sequence
import numpy as np
import os
from tqdm import tqdm

DATA_DIR = "/kaggle/input/ud-english-ewt"
TRAIN_FILE = os.path.join(DATA_DIR, "en_ewt-ud-train.conllu")
DEV_FILE = os.path.join(DATA_DIR, "en_ewt-ud-dev.conllu")
TEST_FILE = os.path.join(DATA_DIR, "en_ewt-ud-test.conllu")

EMBED_DIM = 20000
HIDDEN_DIM = 1024
BATCH_SIZE = 16
EPOCHS = 5
LR = 0.001

PAD_TOKEN = "<PAD>"
UNK_TOKEN = "<UNK>"
PAD_IDX = 0
UNK_IDX = 1

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

def load_conllu(file_path):
    if not os.path.exists(file_path):
        raise FileNotFoundError(f"Không tìm thấy file: {file_path}")

    sentences = []
    current_sent = []

    with open(file_path, 'r', encoding='utf-8') as f:
        for line in f:
            line = line.strip()
            if line.startswith("#"):
                continue

            if not line:
                if current_sent:
                    sentences.append(current_sent)
                    current_sent = []
                continue

            parts = line.split('\t')
            if len(parts) >= 4:
                word, tag = parts[1], parts[3]
                current_sent.append((word, tag))
```

```
    if current_sent:
        sentences.append(current_sent)

    return sentences

def build_vocab(sentences):
    word_set = set()
    tag_set = set()

    for sent in sentences:
        for word, tag in sent:
            word_set.add(word)
            tag_set.add(tag)

    word_to_ix = {PAD_TOKEN: PAD_IDX, UNK_TOKEN: UNK_IDX}
    for i, w in enumerate(sorted(list(word_set)), start=2):
        word_to_ix[w] = i

    tag_to_ix = {PAD_TOKEN: PAD_IDX}
    for i, t in enumerate(sorted(list(tag_set)), start=1):
        tag_to_ix[t] = i

    return word_to_ix, tag_to_ix

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]
        word_idxs = [self.word_to_ix.get(w, UNK_IDX) for w, _ in sent]
        tag_idxs = [self.tag_to_ix.get(t, 0) for _, t in sent]

        return torch.tensor(word_idxs, dtype=torch.long),
torch.tensor(tag_idxs, dtype=torch.long)

def collate_fn(batch):
    word_seqs, tag_seqs = zip(*batch)
    padded_words = pad_sequence(word_seqs, batch_first=True,
padding_value=PAD_IDX)
    padded_tags = pad_sequence(tag_seqs, batch_first=True,
padding_value=PAD_IDX)
    return padded_words, padded_tags
```

```
class SimpleRNNTagger(nn.Module):
    def __init__(self, vocab_size, tag_size, embedding_dim,
hidden_dim):
        super(SimpleRNNTagger, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim,
padding_idx=PAD_IDX)
        self.rnn = nn.RNN(embedding_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, tag_size)

    def forward(self, text):
        # text: [batch, seq_len]
        embedded = self.embedding(text)          # [batch, seq_len,
emb_dim]
        output, _ = self.rnn(embedded)           # [batch, seq_len,
hidden_dim]
        predictions = self.fc(output)            # [batch, seq_len,
tag_size]
        return predictions

def calculate_accuracy(preds, targets, ignore_idx):
    preds = preds.cpu().numpy().flatten()
    targets = targets.cpu().numpy().flatten()

    mask = (targets != ignore_idx)
    correct = np.sum(preds[mask] == targets[mask])
    total = np.sum(mask)

    return correct / total if total > 0 else 0.0

def evaluate(model, dataloader, criterion):
    model.eval()
    total_loss = 0
    total_acc = 0

    with torch.no_grad():
        for words, tags in dataloader:
            words, tags = words.to(DEVICE), tags.to(DEVICE)

            outputs = model(words)
            loss = criterion(outputs.view(-1, outputs.shape[-1]),
tags.view(-1))
            total_loss += loss.item()

            predictions = torch.argmax(outputs, dim=-1)
            total_acc += calculate_accuracy(predictions, tags,
PAD_IDX)

    return total_loss / len(dataloader), total_acc / len(dataloader)

def predict_sentence(model, sentence, word_to_ix, tag_to_ix):
```

```
    model.eval()
    ix_to_tag = {v: k for k, v in tag_to_ix.items()}
    tokens = sentence.strip().split()
    indices = [word_to_ix.get(w, UNK_IDX) for w in tokens]

    tensor_in = torch.tensor([indices], dtype=torch.long).to(DEVICE)

    with torch.no_grad():
        output = model(tensor_in)
        pred_indices = torch.argmax(output, dim=-
1).squeeze(0).cpu().numpy()

    results = [(w, ix_to_tag.get(idx, "<UNK>")) for w, idx in
zip(tokens, pred_indices)]
    print(f"Sentence: {sentence}")
    print(f"Predicted: {results}")

train_data = load_conllu(TRAIN_FILE)
dev_data = load_conllu(DEV_FILE)
print(f"Đã load: {len(train_data)} câu train, {len(dev_data)} câu
dev.")
```

```
Đã load: 12544 câu train, 2001 câu dev.
```

```
word_to_ix, tag_to_ix = build_vocab(train_data)
print(f"Vocab size: {len(word_to_ix)}, Tag size: {len(tag_to_ix)}")
```

```
Vocab size: 20203, Tag size: 19
```

```
train_ds = POSDataset(train_data, word_to_ix, tag_to_ix)
dev_ds = POSDataset(dev_data, word_to_ix, tag_to_ix)

train_loader = DataLoader(train_ds, batch_size=BATCH_SIZE,
shuffle=True, collate_fn=collate_fn)
dev_loader = DataLoader(dev_ds, batch_size=BATCH_SIZE, shuffle=False,
collate_fn=collate_fn)

model = SimpleRNNTagger(len(word_to_ix), len(tag_to_ix), EMBED_DIM,
HIDDEN_DIM).to(DEVICE)
criterion = nn.CrossEntropyLoss(ignore_index=PAD_IDX)
optimizer = optim.Adam(model.parameters(), lr=LR)

best_dev_acc = 0.0

for epoch in range(EPOCHS):
    model.train()
    train_loss = 0
    train_acc_batch = 0

    progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{EPOCHS}", unit="batch", leave=False)
```

```
    for words, tags in progress_bar:
        words, tags = words.to(DEVICE), tags.to(DEVICE)

        optimizer.zero_grad()
        outputs = model(words)
        loss = criterion(outputs.view(-1, outputs.shape[-1]),
tags.view(-1))
        loss.backward()
        optimizer.step()

        train_loss += loss.item()
        preds = torch.argmax(outputs, dim=-1)
        train_acc_batch += calculate_accuracy(preds, tags, PAD_IDX)

        progress_bar.set_postfix({'loss': f'{loss.item():.4f}'})

    avg_train_loss = train_loss / len(train_loader)
    avg_train_acc = train_acc_batch / len(train_loader)

    avg_dev_loss, avg_dev_acc = evaluate(model, dev_loader, criterion)

    print(f"Epoch {epoch+1}/{EPOCHS} | "
          f"Train Loss: {avg_train_loss:.4f} | Train Acc:
{avg_train_acc*100:.2f}% | "
          f"Dev Acc: {avg_dev_acc*100:.2f}%")

    if avg_dev_acc > best_dev_acc:
        best_dev_acc = avg_dev_acc
        torch.save(model.state_dict(), "best_model.pth")
        print("--> Saved Best Model!")

print(f"\nFinal Best Dev Acc: {best_dev_acc*100:.2f}%")

print("\n--- Demo Prediction ---")
model.load_state_dict(torch.load("best_model.pth",
map_location=DEVICE))
predict_sentence(model, "The quick brown fox jumps over the lazy dog",
word_to_ix, tag_to_ix)
```

```
Epoch 1/5 | Train Loss: 0.4328 | Train Acc: 86.15% | Dev Acc: 85.90%
--> Saved Best Model!
```

```
Epoch 2/5 | Train Loss: 0.1990 | Train Acc: 92.51% | Dev Acc: 85.66%
```

```
Epoch 3/5 | Train Loss: 0.1697 | Train Acc: 93.33% | Dev Acc: 85.70%


Epoch 4/5 | Train Loss: 0.1622 | Train Acc: 93.45% | Dev Acc: 86.23%
--> Saved Best Model!


Epoch 5/5 | Train Loss: 0.1582 | Train Acc: 93.64% | Dev Acc: 85.19%

Final Best Dev Acc: 86.23%

--- Demo Prediction ---
Sentence: The quick brown fox jumps over the lazy dog
Predicted: [('The', 'DET'), ('quick', 'ADJ'), ('brown', 'ADJ'),
('fox', 'NOUN'), ('jumps', 'VERB'), ('over', 'ADP'), ('the', 'DET'),
('lazy', 'ADJ'), ('dog', 'NOUN')]
```

```
model.load_state_dict(torch.load("best_model.pth"))

test_sent = "I love NLP"
predict_sentence(model, test_sent, word_to_ix, tag_to_ix)
```

```
Sentence: I love NLP
Predicted: [('I', 'PRON'), ('love', 'VERB'), ('NLP', 'ADV')]
```