Processo Seletivo Engenheiro de LLM

versão 19 de janeiro de 2025

Nome: Livia Souza

E-mail: lsa5@cin.ufpe.br

Instalação e importação de pacotes

```
!pip install datasets -q

import torch
import random
from torch.utils.data import Dataset, DataLoader, random_split

from collections import Counter
import torch.nn as nn
import torch.optim as optim
from datasets import load_dataset

from sklearn.model_selection import train_test_split
import re
from tqdm import tqdm
from transformers import AutoTokenizer

import time
from transformers import AutoTokenizer
```

I - Vocabulário e Tokenização

Exemplo do dataset

```
train_dataset = load_dataset("stanfordnlp/imdb", split="train")

len(train_dataset)

25000

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
```

Retirei a função *encode_sentence* e usei o AutoTokenizer da Huggingface

Escolhi o modelo BERT-base-uncased para realizar a tokenização do texto. Esse modelo divide palavras desconhecidas em subpalavras mais comuns, o que reduz problemas com palavras raras e melhora a generalização do modelo. Poderia ter sido usado um tokenizer básico, em virtude da simplicidade do dataset e do modelo. Mas já que a tarefa foi melhoraro tokenizador, escolhi esse modelo.

Além disso reduzi o tamanho do vocabulário para reduzir a dimensionalidade de entrada e assim o tempo de processamento.

```
# limit the vocabulary size to 20000 most frequent tokens
vocab_size = 5000

counter = Counter()
for sample in tqdm(train_dataset, desc="Building vocabulary"):
    tokens = tokenizer.tokenize(sample["text"])
    counter.update(tokens)

# Criando vocabulário
most_frequent_words = [word for word, _ in
counter.most_common(vocab_size)]
vocab = {word: i for i, word in enumerate(most_frequent_words, 1)}
vocab_size = len(vocab)
```

```
Building vocabulary:   0%|          | 0/25000 [00:00<?, ?it/s]Token
indices sequence length is longer than the specified maximum sequence
length for this model (718 > 512). Running this sequence through the
model will result in indexing errors
Building vocabulary: 100%|██████████| 25000/25000 [01:11<00:00,
351.90it/s]
```

II - Dataset

A implementação garante que o processo de tokenização, vetorização e a construção de conjuntos de dados (train, val, test) seja alinhado com as premissas fornecidas. Então aqui, se deixa o conjunto de dados pronto pra ser treinado.

```
class IMDBDataset(Dataset):
    def __init__(self, data, vocab):
        self.vocab = vocab
        self.processed_data = []

        for sample in data:
            text = sample["text"]
            label = 1 if sample["label"] == 1 else 0

            # Tokenizar o texto
            tokens = tokenizer.tokenize(text)

            # Vetorização one-hot
            encoded = torch.zeros(len(vocab) + 1)
            for token in tokens:
                word_idx = vocab.get(token, 0)  # 0 para OOV
                encoded[word_idx] = 1

            self.processed_data.append((encoded, torch.tensor(label)))
```

```
    def __len__(self):
        return len(self.processed_data)

    def __getitem__(self, idx):
        return self.processed_data[idx]

# Divisão treino/validação
train_size = int(0.8 * len(train_dataset))
val_size = len(train_dataset) - train_size
train_subset, val_subset =
torch.utils.data.random_split(train_dataset, [train_size, val_size])

# Criar datasets
train_data = IMDBDataset(train_subset, vocab)
val_data = IMDBDataset(val_subset, vocab)
test_data = IMDBDataset(load_dataset("stanfordnlp/imdb",
split="test"), vocab)
```

III - Data Loader

Aqui otimizei a paralelização para 2 *workers*, que tráz um bom equilíbrio entre o paralelismo e o overhead. Além disso, adicionei o parâmetro *pin_memory* para otimizar a transferência de dados da CPU para a GPU, reduzindo o overhead de comunicação.

```
batch_size = 128
train_loader = DataLoader(train_data, batch_size=batch_size,
shuffle=True, num_workers=2, pin_memory=True)
val_loader = DataLoader(val_data, batch_size=batch_size,
shuffle=False, num_workers=2, pin_memory=True)
test_loader = DataLoader(test_data, batch_size=batch_size,
shuffle=False, num_workers=2, pin_memory=True)
```

IV - Modelo

```
class OneHotMLP(nn.Module):
    def __init__(self, vocab_size):
        super(OneHotMLP, self).__init__()

        self.fc1 = nn.Linear(vocab_size+1, 200)
        self.fc2 = nn.Linear(200, 1)

        self.relu = nn.ReLU()

    def forward(self, x):
        o = self.fc1(x.float())
        o = self.relu(o)
        return self.fc2(o)
```

```
# Model instantiation
model = OneHotMLP(vocab_size)
```

V - Laço de Treinamento - Otimização da função de Perda pelo Gradiente descendente

```
# Verifica se há uma GPU disponível e define o dispositivo para GPU se
possível,
# caso contrário, usa a CPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
if device.type == 'cuda':
    print('GPU:',
torch.cuda.get_device_name(torch.cuda.current_device()))
else:
    print('using CPU')
```

```
using CPU
```

O código original só realizava o treinamento dentro do laço de épocas, sem realizar validação. Então, incluí a fase de validação logo após o término do treinamento de cada época, garantindo que o modelo seja avaliado adequadamente. A validação é importante para monitorar o desempenho do modelo em dados não vistos durante o treinamento e ajudar a evitar o overfitting.

```
import time

model = model.to(device)
# Define loss and optimizer
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)

num_epochs = 5

for epoch in range(num_epochs):
    start_time = time.time()  # Inicia contagem do tempo

    model.train()
    total_train_loss = 0
    num_train_batches = 0

    for inputs, targets in train_loader:
        inputs, targets = inputs.to(device), targets.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs.squeeze(), targets.float())
        loss.backward()
        optimizer.step()

        total_train_loss += loss.item()
```

```
        num_train_batches += 1

    # Validação
    model.eval()
    total_val_loss = 0
    num_val_batches = 0

    with torch.no_grad():
        for inputs, targets in val_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            loss = criterion(outputs.squeeze(), targets.float())
            total_val_loss += loss.item()
            num_val_batches += 1

    avg_train_loss = total_train_loss / num_train_batches
    avg_val_loss = total_val_loss / num_val_batches
    epoch_time = time.time() - start_time  # Calcula tempo da época


    print(f'Epoch [{epoch+1}/{num_epochs}], '
          f'Train Loss: {avg_train_loss:.4f}, '
          f'Val Loss: {avg_val_loss:.4f}, '
          f'Time: {epoch_time:.2f}s')
```

```
Epoch [1/5], Train Loss: 0.6892, Val Loss: 0.6833, Time: 5.71s
Epoch [2/5], Train Loss: 0.6749, Val Loss: 0.6645, Time: 4.04s
Epoch [3/5], Train Loss: 0.6498, Val Loss: 0.6326, Time: 3.96s
Epoch [4/5], Train Loss: 0.6110, Val Loss: 0.5875, Time: 4.73s
Epoch [5/5], Train Loss: 0.5613, Val Loss: 0.5362, Time: 4.07s
```

VI - Avaliação

A acurácia foi acima de 65%, obedecendo a premissa dada. Mas ressalvo que foi melhor ainda no modelo que usei um tokenizador manual básico.

```
## evaluation
model.eval()

with torch.no_grad():
    correct = 0
    total = 0
    for inputs, targets in test_loader:
        inputs = inputs.to(device)
        targets = targets.to(device)
        logits = model(inputs)
        predicted = torch.round(torch.sigmoid(logits.squeeze()))
        total += targets.size(0)
        correct += (predicted == targets).sum().item()
```

```
    print(f'Test Accuracy: {100 * correct / total}%')
```

```
Test Accuracy: 80.448%
```