task1

March 23, 2025

1 Task 1: Sentence Transformer Implementation

[1]:
```
import numpy as np
import torch
import torch.nn as nn
from transformers import AutoModel, AutoTokenizer
```

[2]:
```
class SentenceTransformer(nn.Module):
    def __init__(self,
                 model_name: str = "distilbert-base-uncased",
                 embedding_dim: int = 768,
                 pooling_strategy: str = "mean",
                 max_length: int = 128) -> None:
        super(SentenceTransformer, self).__init__()
        self.tokenizer = AutoTokenizer.from_pretrained(model_name)
        self.transformer = AutoModel.from_pretrained(model_name)
        self.pooling_strategy = pooling_strategy
        self.max_length = max_length

        self.use_projection = None
        if embedding_dim != self.transformer.config.hidden_size:
            self.use_projection = True
        else:
            self.use_projection = False

        if self.use_projection is True:
            self.projection = nn.Linear(self.transformer.config.hidden_size,␣
↪embedding_dim)

    def forward(self, sentences: list[str]) -> torch.Tensor:
        inputs = self.tokenizer(sentences, return_tensors="pt", padding=True,␣
↪truncation=True, max_length=self.max_length)
        device = next(self.transformer.parameters()).device
        inputs = {k: v.to(device) for k, v in inputs.items()}

        with torch.no_grad():
            outputs = self.transformer(**inputs)
```

```
        hidden_states = outputs.last_hidden_state

        if self.pooling_strategy == "cls":
            pooled = hidden_states[:, 0]
        elif self.pooling_strategy == "max":
            pooled = torch.max(hidden_states, dim=1)[0]
        else:
            attention_mask = inputs["attention_mask"].unsqueeze(-1)
            pooled = torch.sum(hidden_states * attention_mask, dim=1) / torch.
↪sum(attention_mask, dim=1)

        if self.use_projection is True:
            pooled = self.projection(pooled)

        return pooled

    def encode(self, sentences: list[str], batch_size: int = 32, normalize:␣
↪bool = False) -> np.ndarray:
        self.eval()
        all_embeddings = []
        with torch.no_grad():
            for i in range(0, len(sentences), batch_size):
                batch = sentences[i : i + batch_size]
                embeddings = self.forward(batch)
                if normalize is True:
                    embeddings = nn.functional.normalize(embeddings)
                all_embeddings.append(embeddings.cpu().numpy())
        all_embeddings = np.vstack(all_embeddings)
        return all_embeddings
```

[3]:
```
def main() -> None:
    test_model = SentenceTransformer(model_name="distilbert-base-uncased",␣
↪embedding_dim=384, pooling_strategy="mean", max_length=128)
    test_sentences = [
            "This is a simple sentence to encode.",
            "The quick brown fox jumps over the lazy dog.",
            "Sentence transformers are useful for many NLP tasks.",
            "This sentence is similar to the first one but with different words.
↪",
            "Machine learning models can process natural language effectively."
    ]
    test_embeddings = test_model.encode(sentences=test_sentences,␣
↪normalize=True)
    print(f"Generated {test_embeddings.shape[0]} embeddings with dimension␣
↪{test_embeddings.shape[1]}")
    print(f"Sample of the first embedding vector:\n{test_embeddings[0][:10]}")
```

```
    print("Computing similarities between sentences:")
    for i in range(len(test_sentences)):
        for j in range(i + 1, len(test_sentences)):
            similarity = np.dot(test_embeddings[i], test_embeddings[j])
            print(f"Similarity between sentence {i + 1} and {j + 1}:␣
 ↪{similarity:.4f}")

if __name__ == "__main__":
    main()
```

```
Generated 5 embeddings with dimension 384
Sample of the first embedding vector:
[ 0.01348774  0.02294292 -0.07688586 -0.05261944 -0.04356397  0.05017351
 -0.05570818 -0.0127305   0.04803646 -0.02740604]
Computing similarities between sentences:
Similarity between sentence 1 and 2: 0.5727
Similarity between sentence 1 and 3: 0.8527
Similarity between sentence 1 and 4: 0.8012
Similarity between sentence 1 and 5: 0.7978
Similarity between sentence 2 and 3: 0.6158
Similarity between sentence 2 and 4: 0.5676
Similarity between sentence 2 and 5: 0.5876
Similarity between sentence 3 and 4: 0.6867
Similarity between sentence 3 and 5: 0.8547
Similarity between sentence 4 and 5: 0.6446
```

1.1 Description for Task 1

1.1.1 Choosing a suitable large language model

I started Task 1 by selecting the suitable architectural model for the transformer. I decided to choose DistilBERT, a distilled version of BERT. BERT itself has been a popular large language model (LLM) for natural language processing (NLP) experiments for many years. Although DistilBERT is smaller, we can use it to achieve a 60% faster training speed while maintaining a 95% performance, as suggested by HuggingFace. By using DistilBERT, we can reduce the computational burden by 40% while retaining 97% of its language understanding capabilities and being 60% faster.

1.1.2 Other design choices

For pooling strategies, I implemented three different pooling strategies in the code. They play a significant role no how the meaning of a word inside a sentence is parsed and interpreted. The **mean pooling** strategy works by calculating the mean among all token embeddings, which is the default strategy used by the sentence transformer model. For instance, if we want to get a 2×2 sample pool size, then we can calculate the mean for each of the 4 token embeddings and create a downsized pool. The "CLS" in **CLS token** stands for "classification". It works by going through tokenization at sentence-level classification, which is an essential feature in BERT. The last strategy is **max pooling**, which is similar to the mean pooling aforementioned. But instead of calculating the mean values, it takes the maximum values.

Moving on, let's discuss the dimension reduction layer of my model. By default, DistilBERT has a dimensionality of 768 for the encoder layers and the pooler layer. However, I've reduced this number to half (384) in my code by using an optional projection layer. By cutting the embedding dimension in half, I've also reduced the computational burden and storage requirements. Furthermore, it helps to prevent overfitting in downstream tasks, and makes calculations on similarity values faster.

Additionally, I've also added an option for normalization on the embeddings, which uses the L2 normalization to calculate the norm in a Euclidean space. The L2 norm is calculated as:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{k=1}^{n} |x_k|^2} = \sqrt{x_1^2 + \cdots + x_n^2}$$

The norm option ensures that all embeddings have unit length and simplified the calculations of cosine similarity. It also improves peroformance in retrieval tasks. Cosine similarity is calculated as a dot product as:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

Finally, I implemented batch processing in the `encode` function by passing the parameter `batch_size`. Introducing batch processing to lists and other large size data helps to prevent memory issues associated with large size inputs and improves the customizability of the code for future developers.

1.1.3 References:

DistilBERT

[ ]: