Obtain the embedding vectors and implement custom loss function in BERT

To obtain embedding vectors from BERT and implement a custom loss function, you'll first need to load the BERT model and tokenizer from the Hugging Face Transformers library. Then, you can use the tokenizer to convert your input text into input IDs and attention masks. After that, you can pass these inputs to the BERT model to get the hidden states, which represent the embedding vectors. Finally, you can define and use a custom loss function by overriding the compute\_loss method in a custom Trainer class or by using a custom module with a custom loss function.

I. Loading the Model and Tokenizer:

from transformers import AutoTokenizer, AutoModel

# Choose the BERT model you want to use (e.g., "bert-base-uncased")

model\_name = "bert-base-uncased"

# Load the tokenizer

tokenizer = AutoTokenizer.from\_pretrained(model\_name)

# Load the BERT model

model = AutoModel.from\_pretrained(model\_name)

II. Tokenizing and Encoding:

# Example input text

text = "This is an example sentence."

# Tokenize the text

input\_ids = tokenizer.encode(text, return\_tensors="pt")

attention\_mask = torch.ones(input\_ids.shape, dtype=torch.long) # Attention mask for all tokens

III. Obtaining Embedding Vectors:

# Set the model to evaluation mode

model.eval()

# Disable gradient calculations

with torch.no\_grad():

# Pass the input IDs and attention mask to the model

outputs = model(input\_ids, attention\_mask=attention\_mask)

# The first element of the output is the hidden states (embeddings)

embedding\_vectors = outputs[0]

# You can access the [CLS] token embedding for sentence representation

cls\_embedding = embedding\_vectors[:, 0, :]

print(cls\_embedding)

IV. Implementing a Custom Loss Function:

import torch

from torch import nn

from transformers import Trainer, TrainingArguments

# Define a custom loss function (e.g., cosine similarity loss)

class CosineSimilarityLoss(nn.Module):

def \_\_init\_\_(self, embedding\_dim):

super().\_\_init\_\_()

self.embedding\_dim = embedding\_dim

def forward(self, embeddings, labels):

# Calculate cosine similarity

similarity\_matrix = torch.matmul(embeddings, embeddings.transpose(0, 1)) # (batch\_size, batch\_size)

# Example loss: sum of cosine similarities between same classes

loss = torch.sum(torch.diag(similarity\_matrix)) / (embeddings.shape[0] \* self.embedding\_dim)

return loss

# Define a custom model with the loss function

class CustomBertModel(nn.Module):

def \_\_init\_\_(self, bert\_model, num\_labels):

super().\_\_init\_\_()

self.bert = bert\_model

self.classifier = nn.Linear(bert\_model.config.hidden\_size, num\_labels)

def forward(self, input\_ids, attention\_mask, labels=None):

outputs = self.bert(input\_ids, attention\_mask=attention\_mask)

hidden\_states = outputs[0] # [CLS] token embedding

# Pass hidden states to the classifier

logits = self.classifier(hidden\_states[:, 0, :])

if labels is not None:

# Calculate loss (e.g., cross-entropy)

loss = nn.CrossEntropyLoss()(logits, labels)

return {"logits": logits, "loss": loss}

return {"logits": logits}

# Define a custom Trainer with the loss function

class CustomTrainer(Trainer):

def compute\_loss(self, model, inputs, return\_outputs=False):

outputs = model(\*\*inputs)

if "loss" in outputs:

loss = outputs["loss"]

else:

# Define a custom loss here (e.g., CosineSimilarityLoss)

labels = inputs.get("labels")

embeddings = outputs["logits"] # Or outputs["hidden\_states"]

loss = self.custom\_loss(embeddings, labels)

return (loss, outputs) if return\_outputs else loss