

The problem might be with the `summary` function from `torchsummary`. This function expects a certain input shape that might not be compatible with how Hugging Face's Transformers library has implemented BERT.

A workaround for this is to manually specify the input shape to match the expected input by BERT:

```
```python
import torch

from transformers import BertForSequenceClassification

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

Initialize and load model
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
model.to(device)

Create dummy input with shape [batch_size, sequence_length]
and move it to the device
dummy_input = torch.randint(0, 1000, (1, 512)).to(device)

Manually forward pass
with torch.no_grad():
 output = model(dummy_input)

print(output)
```
```

This code snippet skips using `summary` but does a forward pass manually with a dummy input tensor to check if the model works as expected.

Remember to train the model on a downstream task if you're planning to use it for inference, as the warning message suggests.

USER

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
SequenceClassifierOutput(loss=None, logits=tensor([[ -0.2037,  0.1793]]), device='cuda:0'),  
hidden_states=None, attentions=None)
```

ASSISTANT

The output you're seeing indicates that the model has been successfully loaded and executed a forward pass. The logits are the raw, unnormalized scores for each class in the output layer. You can apply a softmax function to these logits to obtain probabilities if needed.

The warning message is indicating that you've loaded a pre-trained BERT model but added a classification layer on top ("classifier.bias", "classifier.weight") which hasn't been trained yet. Because of this, you should further train the model on your specific task to fine-tune these newly initialized weights.

Would you like to proceed with fine-tuning or any other specific operation?

USER

Yes I want to also calculate thop

ASSISTANT

To calculate FLOPs (Floating Point Operations) using the `thop` library, you'll first need to install the package. You can do so by running:

```
```bash
pip install thop
```
```

Then, use the following code snippet to fine-tune your BERT model and calculate its FLOPs. Note that the code assumes you've already loaded your model and tokenizer:

```
```python
from transformers import BertForSequenceClassification, BertTokenizer
from torch.utils.data import DataLoader
from thop import profile
import torch
import torch.optim as optim

Initialize the tokenizer and model
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

model = BertForSequenceClassification.from_pretrained('bert-base-uncased',
num_labels=2).to('cuda')
```

```
Sample training data
```

```
sentences = ["Hello, how are you?", "I am fine."]
```

```
labels = torch.tensor([0, 1])
```

```
Tokenize the sentences
```

```
inputs = tokenizer(sentences, padding=True, truncation=True, return_tensors="pt").to('cuda')
```

```
labels = labels.to('cuda')
```

```
Define the optimizer
```

```
optimizer = optim.Adam(model.parameters(), lr=1e-5)
```

```
Fine-tuning loop
```

```
for epoch in range(3): # 3 epochs, adjust as needed
```

```
 model.train()
```

```
 optimizer.zero_grad()
```

```
 outputs = model(**inputs, labels=labels)
```

```
 loss = outputs.loss
```

```
 loss.backward()
```

```
 optimizer.step()
```

```
 print(f"Epoch {epoch+1}, Loss: {loss.item()}")
```

```
Calculate FLOPs
```

```
input_size = (1, 512) # Batch size 1, sequence length 512
```

```

macs, params = profile(model, inputs=(torch.randn(input_size).to('cuda'),))

print(f"MACs: {macs}, Params: {params}")

...

```

This will output the number of MACs (Multiply-accumulate operations), which are closely related to FLOPs, and the number of parameters.

Remember, fine-tuning like this is a very simplified example. In a real-world scenario, you would use a proper dataset and DataLoader, among other things.

## USER

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Epoch 1, Loss: 0.676777720451355

Epoch 2, Loss: 0.6179714202880859

Epoch 3, Loss: 0.534701943397522

[INFO] Register count\_normalization() for <class 'torch.nn.modules.normalization.LayerNorm'>.

[INFO] Register zero\_ops() for <class 'torch.nn.modules.dropout.Dropout'>.

[INFO] Register count\_linear() for <class 'torch.nn.modules.linear.Linear'>.

-----

RuntimeError                      Traceback (most recent call last)

<ipython-input-23-069885daefc8> in <cell line: 36>()

34 # Calculate FLOPs

35 input\_size = (1, 512) # Batch size 1, sequence length 512

```
---> 36 macs, params = profile(model, inputs=(torch.randn(input_size).to('cuda'),))
```

```
37 print(f"MACs: {macs}, Params: {params}")
```

9 frames

```
/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py in embedding(input, weight,
padding_idx, max_norm, norm_type, scale_grad_by_freq, sparse)
```

```
2208 # remove once script supports set_grad_enabled
```

```
2209 _no_grad_embedding_renorm_(weight, input, max_norm, norm_type)
```

```
-> 2210 return torch.embedding(weight, input, padding_idx, scale_grad_by_freq, sparse)
```

```
2211
```

```
2212
```

RuntimeError: Expected tensor for argument #1 'indices' to have one of the following scalar types: Long, Int; but got torch.cuda.FloatTensor instead (while checking arguments for embedding)

## ASSISTANT

The error "Expected tensor for argument #1 'indices' to have one of the following scalar types: Long, Int; but got torch.cuda.FloatTensor instead" suggests that the input tensor to the embedding layer needs to be of type Long or Int.

The problem is in this line:

```
```python
```

```
macs, params = profile(model, inputs=(torch.randn(input_size).to('cuda'),))
```

```
...
```

`torch.randn()` generates a tensor of floats, but the embedding layer expects Long or Int tensors as input indices. This is not an issue when you are using the model for prediction or fine-tuning since the tokenizer handles it, but it will be an issue when you're directly feeding in random data.

Try replacing that line with something like:

```
```python
```

```
random_input = torch.randint(0, tokenizer.vocab_size, input_size).to('cuda')
```

```
macs, params = profile(model, inputs=(random_input,))
```

```
...
```

This will generate a tensor of random integers between 0 and `tokenizer.vocab_size`, which should be acceptable for the embedding layer.

```
USER
```

```
import torch
```

```
import torch.nn as nn
```

```
import torch.optim as optim
```

```
import numpy as np
```

```
from sklearn.feature_selection import SelectKBest, chi2
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
import time
```

```
Your embeddings and labels (replace with your actual data)

embeddings = np.random.rand(100, 768)

labels = np.random.randint(0, 2, 100)

Feature selection using Chi-Squared

X_new = SelectKBest(chi2, k=200).fit_transform(embeddings, labels)

Convert to PyTorch tensors

X_tensor = torch.tensor(X_new, dtype=torch.float32)

y_tensor = torch.tensor(labels, dtype=torch.long)

Define neural network model

class SimpleNet(nn.Module):

 def __init__(self):

 super(SimpleNet, self).__init__()

 self.fc1 = nn.Linear(200, 128)

 self.fc2 = nn.Linear(128, 2)

 def forward(self, x):

 x = torch.relu(self.fc1(x))

 x = self.fc2(x)

 return x

Initialize model, loss, and optimizer

model = SimpleNet().cuda()

criterion = nn.CrossEntropyLoss()
```



```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
Initialize metrics
```

```
all_metrics = {}
```

```
Training loop
```

```
num_epochs = 10
```

```
for epoch in range(num_epochs):
```

```
 start_time = time.time()
```

```
 model.train()
```

```
 outputs = model(X_tensor.cuda())
```

```
 loss = criterion(outputs, y_tensor.cuda())
```

```
 optimizer.zero_grad()
```

```
 loss.backward()
```

```
 optimizer.step()
```

```
 model.eval()
```

```
 with torch.no_grad():
```

```
 preds = torch.argmax(outputs, dim=1).cpu().numpy()
```

```
Compute metrics
```

```
acc = accuracy_score(y_tensor, preds)
```

```
precision = precision_score(y_tensor, preds, average='weighted')
```

```
recall = recall_score(y_tensor, preds, average='weighted')
```

```
f1 = f1_score(y_tensor, preds, average='weighted')
```

```
all_metrics[epoch] = {
 'Accuracy': acc,
 'Precision': precision,
 'Recall': recall,
 'F1 Score': f1
}
```

```
elapsed_time = time.time() - start_time
```

```
Compute FLOPs
```

```
torch.cuda.synchronize()
```

```
torch.cuda.profiler.start()
```

```
model(X_tensor.cuda())
```

```
torch.cuda.profiler.stop()
```

```
torch.cuda.synchronize()
```

```
print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}, Time: {elapsed_time}, Metrics:
{all_metrics[epoch]}")
```

This is supposed to train on json files in my embeddings dir

## ASSISTANT

Your provided code seems to be a skeleton for training a simple neural network using PyTorch. The code appears to use random embeddings and labels for demonstration, and then applies feature selection using the Chi-Squared test. Following that, it trains a simple neural network. However, I

noticed a couple of things:

1. You mentioned that this is supposed to train on JSON files in your embeddings directory, but the code doesn't include any code for loading JSON files.
2. The FLOPs calculation part is missing; you've mentioned it in a comment but not implemented it.

Here's how you can modify the code to load embeddings and labels from JSON files and also add FLOPs calculation:

1. First, add a function to load data from JSON files:

```
```python
import json

def load_data_from_json(file_path):
    with open(file_path, 'r') as f:
        data = json.load(f)
    return data
```
```

2. Replace the embeddings and labels with loaded data:

```
```python
# Replace these lines

# embeddings = np.random.rand(100, 768)
```

```
# labels = np.random.randint(0, 2, 100)
```

```
# With these
```

```
embeddings = load_data_from_json('path/to/your/embeddings.json')
```

```
labels = load_data_from_json('path/to/your/labels.json')
```

```
...
```

3. For calculating FLOPs, you can use the `torchprofile` package. Install it using `pip install torchprofile`, then add this code:

```
```python
```

```
from torchprofile import profile
```

```
macs, params = profile(model, inputs=(X_tensor[0:1].cuda(),))
```

```
print(f"MACs: {macs}, Params: {params}")
```

```
...
```

Insert this code snippet after the training loop to get the FLOPs.

Combine these modifications with your existing code, and you should be able to train the model using embeddings and labels from your JSON files while also calculating FLOPs.

```
USER
```

```
GPT_MODEL
```

```
str
```

```
18 chars
```

'gpt-3.5-turbo-0613'

all\_embeddings

list

12068352 items

[0.001567214378155768, -0.016733262687921524, -0.0009632634464651346,  
0.0024723762180656195, 0.00245708622969687, 0.013699746690690517,  
-0.0011559161357581615, 0.00047857375466264784, 0.020268285647034645,  
-0.001288173720240593, 0.005195506848394871, -0.0012614164734259248,  
-0.0030488052871078253, -0.007699991576373577, ...]

chroma\_client

SegmentAPI

<chromadb.api.segment.SegmentAPI object at 0x7d7db2de2170>

directories

list

3 items

['./drive/MyDrive/embeddings', './drive/MyDrive/embeddings2', './drive/MyDrive/embeddings3']

directory

str

27 chars

'./drive/MyDrive/embeddings3'

embeddings

list

1536 items

[0.0015720828669145703, -0.016736259683966637, -0.0009687884012237191,  
0.002483523916453123, 0.002459055744111538, 0.01370220072567463,

```
-0.0011500060791149735, 0.000485541153466329, 0.020271915942430496,
-0.0012838166439905763, 0.005208671558648348, -0.0012746411375701427,
-0.003043234348297119, -0.007707487791776657, ...]
```

f

TextIOWrapper

```
<_io.TextIOWrapper name='./drive/MyDrive/embeddings3/embedding_3760.json' mode='r'
encoding='UTF-8'>
```

filename

str

19 chars

'embedding\_3760.json'

filepath

str

47 chars

'./drive/MyDrive/embeddings3/embedding\_3760.json'

Make it train on my variables

## ASSISTANT

Certainly, to train your model on the data you've loaded into variables, you need to modify the code to use those variables instead of random data. Assuming that `all\_embeddings` is a list containing the feature vectors and that each feature vector is of length 1536, and also assuming that you have a corresponding list of labels (not shown in your variables), you can proceed as follows:

Here's how to adapt your existing code:

1. Replace the random embeddings and labels with your variables `all\_embeddings` and your list of labels, which you haven't shown but presumably also have.
2. Ensure that the feature vector length in the `SimpleNet` constructor matches the length of your embeddings.

Here's how the modified code would look:

```
```python
import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

import time

# Replace the random embeddings and labels with your variables

embeddings = np.array(all_embeddings).reshape(-1, 1536) # Reshape according to your actual
shape

# You'd also load your labels here; since I don't see them in your variables, let's assume it's called
'all_labels'

labels = np.array(all_labels)

# Convert to PyTorch tensors
```

```
X_tensor = torch.tensor(embeddings, dtype=torch.float32)
```

```
y_tensor = torch.tensor(labels, dtype=torch.long)
```

Modify SimpleNet to match the shape of your embeddings

```
class SimpleNet(nn.Module):
```

```
    def __init__(self):
```

```
        super(SimpleNet, self).__init__()
```

```
        self.fc1 = nn.Linear(1536, 128) # Feature length is now 1536
```

```
        self.fc2 = nn.Linear(128, 2)
```

```
    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
```

```
        x = self.fc2(x)
```

```
        return x
```

```
# ... rest of your code stays the same
```

```
...
```

Make sure to define `all_labels` to contain your labels, just as `all_embeddings` contains your feature vectors.

This way, your model should train on the embeddings and labels stored in the variables you provided.

```
## USER
```

Combine make all one

ASSISTANT

Certainly, let's combine all the steps into a single code block that trains the model on your variables `all_embeddings` and `all_labels`. Note that I'm assuming `all_labels` exists and has the same length as `all_embeddings`.

```
```python
import torch

import torch.nn as nn

import torch.optim as optim

import numpy as np

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

import time

Your actual embeddings and labels (replace all_embeddings and all_labels with your actual data)
all_embeddings = np.array(all_embeddings).reshape(-1, 1536) # Assuming each feature vector is
of length 1536

all_labels = np.array(all_labels) # Assuming you have a variable named all_labels

Convert to PyTorch tensors

X_tensor = torch.tensor(all_embeddings, dtype=torch.float32)

y_tensor = torch.tensor(all_labels, dtype=torch.long)

Neural network model

class SimpleNet(nn.Module):

 def __init__(self):
```

```
super(SimpleNet, self).__init__()
```

```
self.fc1 = nn.Linear(1536, 128)
```

```
self.fc2 = nn.Linear(128, 2)
```

```
def forward(self, x):
```

```
 x = torch.relu(self.fc1(x))
```

```
 x = self.fc2(x)
```

```
 return x
```

```
Initialize model, loss, and optimizer
```

```
model = SimpleNet().cuda()
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
Initialize metrics
```

```
all_metrics = {}
```

```
Training loop
```

```
num_epochs = 10
```

```
for epoch in range(num_epochs):
```

```
 start_time = time.time()
```

```
 model.train()
```

```
 outputs = model(X_tensor.cuda())
```

```
 loss = criterion(outputs, y_tensor.cuda())
```

```

optimizer.zero_grad()

loss.backward()

optimizer.step()

model.eval()

with torch.no_grad():

 preds = torch.argmax(outputs, dim=1).cpu().numpy()

Compute metrics

acc = accuracy_score(y_tensor.cpu(), preds)

precision = precision_score(y_tensor.cpu(), preds, average='weighted')

recall = recall_score(y_tensor.cpu(), preds, average='weighted')

f1 = f1_score(y_tensor.cpu(), preds, average='weighted')

all_metrics[epoch] = {

 'Accuracy': acc,

 'Precision': precision,

 'Recall': recall,

 'F1 Score': f1

}

elapsed_time = time.time() - start_time

print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}, Time: {elapsed_time}, Metrics:
{all_metrics[epoch]}")
...

```

Remember to define `all\_labels` in the same manner as `all\_embeddings`. This combined code should train the model on the data you've loaded into the `all\_embeddings` and `all\_labels` variables.

## USER

All\_labels is not defined Define it as the variable embeddings

## ASSISTANT

If `all\_labels` is not defined and you want to use `embeddings` as labels, I assume that `embeddings` contains your label information. Below is a modified version of the code that assumes `embeddings` is a list of labels matching the length of `all\_embeddings`.

```
```python
```

```
import torch
```

```
import torch.nn as nn
```

```
import torch.optim as optim
```

```
import numpy as np
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
import time
```

```
# Your actual embeddings (replace all_embeddings with your actual data)
```

```
all_embeddings = np.array(all_embeddings).reshape(-1, 1536) # Assuming each feature vector is  
of length 1536
```

```
all_labels = np.array(embeddings) # Using embeddings as labels
```

```
# Convert to PyTorch tensors
```

```
X_tensor = torch.tensor(all_embeddings, dtype=torch.float32)
```

```
y_tensor = torch.tensor(all_labels, dtype=torch.long)
```

```
# Neural network model
```

```
class SimpleNet(nn.Module):
```

```
    def __init__(self):
```

```
        super(SimpleNet, self).__init__()
```

```
        self.fc1 = nn.Linear(1536, 128)
```

```
        self.fc2 = nn.Linear(128, 2)
```

```
    def forward(self, x):
```

```
        x = torch.relu(self.fc1(x))
```

```
        x = self.fc2(x)
```

```
        return x
```

```
# Initialize model, loss, and optimizer
```

```
model = SimpleNet().cuda()
```

```
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
# Initialize metrics
```

```
all_metrics = {}
```

```
# Training loop
```

```
num_epochs = 10
```

```
for epoch in range(num_epochs):
```

```
start_time = time.time()
```

```
model.train()
```

```
outputs = model(X_tensor.cuda())
```

```
loss = criterion(outputs, y_tensor.cuda())
```

```
optimizer.zero_grad()
```

```
loss.backward()
```

```
optimizer.step()
```

```
model.eval()
```

```
with torch.no_grad():
```

```
    preds = torch.argmax(outputs, dim=1).cpu().numpy()
```

```
# Compute metrics
```

```
acc = accuracy_score(y_tensor.cpu(), preds)
```

```
precision = precision_score(y_tensor.cpu(), preds, average='weighted')
```

```
recall = recall_score(y_tensor.cpu(), preds, average='weighted')
```

```
f1 = f1_score(y_tensor.cpu(), preds, average='weighted')
```

```
all_metrics[epoch] = {
```

```
    'Accuracy': acc,
```

```
    'Precision': precision,
```

```
    'Recall': recall,
```

```
    'F1 Score': f1
```

```
}
```

```

elapsed_time = time.time() - start_time

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item()}, Time: {elapsed_time}, Metrics:
{all_metrics[epoch]}")
'''

```

Make sure the `embeddings` variable actually contains labels for the corresponding `all_embeddings`. If it does not, you'll need to acquire or generate appropriate labels for the training data.

USER

ValueError Traceback (most recent call last)

<ipython-input-9-56aaa5ae0eee> in <cell line: 38>()

```

41 model.train()

42 outputs = model(X_tensor.cuda())
---> 43 loss = criterion(outputs, y_tensor.cuda())

44

45 optimizer.zero_grad()

```

2 frames

/usr/local/lib/python3.10/dist-packages/torch/nn/functional.py in cross_entropy(input, target, weight, size_average, ignore_index, reduce, reduction, label_smoothing)

```

3027     if size_average is not None or reduce is not None:
3028         reduction = _Reduction.legacy_get_string(size_average, reduce)
-> 3029         return torch._C._nn.cross_entropy_loss(input, target, weight,
_Reduction.get_enum(reduction), ignore_index, label_smoothing)
3030
3031

```

ValueError: Expected input batch_size (7857) to match target batch_size (1536)

ASSISTANT

The error suggests that the number of samples in your input (`X_tensor`) and target (`y_tensor`) do not match. You have 7857 samples in the input batch (`X_tensor`) and 1536 in the target batch (`y_tensor`).

Make sure the lengths of `all_embeddings` and `embeddings` (used for `all_labels` in this case) are the same, so that each feature vector from `all_embeddings` has a corresponding label in `all_labels`.

Here is how you can check:

```

```python
print("Length of all_embeddings:", len(all_embeddings))

print("Length of embeddings:", len(embeddings))

if len(all_embeddings) != len(embeddings):

 print("Mismatch: Number of feature vectors does not match number of labels.")

```



...

After confirming that both are of the same length, you should be able to proceed with training the model. If they aren't, you'll need to investigate why and correct it.

## USER

Length of all\_embeddings: 7857

Length of embeddings: 1536

Mismatch: Number of feature vectors does not match number of labels.

## ASSISTANT

The mismatch in lengths between ``all_embeddings`` and ``embeddings`` is causing the issue. You need to make sure that each feature vector in ``all_embeddings`` has a corresponding label in ``embeddings`` (which you're using for ``all_labels``).

Here are some potential solutions:

1. **\*\*Check Data Sources\*\***: Make sure ``all_embeddings`` and ``embeddings`` are generated from the same data source and are supposed to have the same length.
2. **\*\*Truncate or Pad\*\***: You could truncate the longer list to the length of the shorter one. Alternatively, if the missing labels or embeddings can be inferred or set to a default, you could pad the shorter list.
3. **\*\*Debug\*\***: If neither of the above options is applicable, it might be helpful to debug how these two lists are being generated to figure out why they have different lengths.

Here's an example of how to truncate `all\_embeddings` to match the length of `embeddings`:

```
```python
# Truncate all_embeddings to match the length of embeddings
all_embeddings = all_embeddings[:len(embeddings)]

# Now the lengths should match
print("Length of all_embeddings:", len(all_embeddings))
print("Length of embeddings:", len(embeddings))
```
```

Once the lengths match, you can proceed with your existing code for model training.

## USER

No make it update to

The larger value

## ASSISTANT

To make the smaller array match the larger one, you can pad it with zeros or any other default value. This will make the length of `embeddings` the same as `all\_embeddings`. Here's how to do it using NumPy:

```
```python
import numpy as np
```

```
# Assuming all_embeddings is a list of 7857 items and embeddings is a list of 1536 items
```

```
length_all_embeddings = 7857
```

```
length_embeddings = 1536
```

```
# If embeddings is shorter, pad it with zeros to match the length of all_embeddings
```

```
if length_embeddings < length_all_embeddings:
```

```
    pad_length = length_all_embeddings - length_embeddings
```

```
    padding = np.zeros((pad_length, embeddings.shape[1])) # Assuming embeddings is 2D
```

```
    embeddings = np.vstack([embeddings, padding])
```

```
# Now embeddings should have the same length as all_embeddings
```

```
print("Length of all_embeddings:", len(all_embeddings))
```

```
print("Length of embeddings:", len(embeddings))
```

```
# Now you can proceed with your training code
```

```
...
```

Note: Padding the `embeddings` array might not be the best approach in all cases, as the padded values are just placeholders and don't represent actual data. However, this will resolve the dimension mismatch issue.

```
## USER
```

```
eback (most recent call last)
```

```
<ipython-input-11-ea2fd312652f> in <cell line: 8>()
```

```
8 if length_embeddings < length_all_embeddings:
```

```
9     pad_length = length_all_embeddings - length_embeddings
```

```
---> 10 padding = np.zeros((pad_length, embeddings.shape[1])) # Assuming embeddings is 2D
      11 embeddings = np.vstack([embeddings, padding])
      12
```

AttributeError: 'list' object has no attribute 'shape'

ASSISTANT

Apologies for the oversight. If `embeddings` is a list and not a NumPy array, you won't have access to the `.shape` attribute. You can convert the list to a NumPy array first or directly get the number of features in another way. Here's a modified snippet:

```
```python
Convert embeddings to a NumPy array, if it's not already
embeddings_array = np.array(embeddings)

Get the number of features
num_features = embeddings_array.shape[1]

if length_embeddings < length_all_embeddings:
 pad_length = length_all_embeddings - length_embeddings
 padding = np.zeros((pad_length, num_features))
 embeddings_array = np.vstack([embeddings_array, padding])

Convert back to list if needed
embeddings = embeddings_array.tolist()
```

```
Now embeddings should have the same length as all_embeddings

print("Length of all_embeddings:", len(all_embeddings))

print("Length of embeddings:", len(embeddings))

...


```

This should resolve the 'list object has no attribute shape' error.

## USER

or                      Traceback (most recent call last)

<ipython-input-13-7e2fdc6e3ba7> in <cell line: 5>()

3

4 # Get the number of features

----> 5 num\_features = embeddings\_array.shape[1]

6

7 if length\_embeddings < length\_all\_embeddings:

IndexError: tuple index out of range

## ASSISTANT

Ah, my apologies again. The error suggests that `embeddings\_array` is a 1D array, meaning it doesn't have a second dimension for `.shape[1]` to refer to. If your `embeddings` list is supposed to be 1D, it won't have the number of features in the way a 2D array would.

If you want to make `embeddings` and `all\_embeddings` the same length, you can do so as follows:

```
```python
```

```
# Convert both to NumPy arrays

embeddings_array = np.array(embeddings)

all_embeddings_array = np.array(all_embeddings)


# Find the lengths

length_embeddings = len(embeddings_array)

length_all_embeddings = len(all_embeddings_array)


# Padding

if length_embeddings < length_all_embeddings:

    pad_length = length_all_embeddings - length_embeddings

    padding = np.zeros(pad_length)

    embeddings_array = np.concatenate([embeddings_array, padding])


# If you want to convert back to lists:

embeddings = embeddings_array.tolist()

all_embeddings = all_embeddings_array.tolist()


# Now embeddings should have the same length as all_embeddings

print("Length of all_embeddings:", len(all_embeddings))

print("Length of embeddings:", len(embeddings))

...

```

This assumes both `embeddings` and `all_embeddings` are 1D. If this doesn't meet your needs, could you please clarify the dimensions and what you are looking to accomplish?

USER

or Traceback (most recent call last)

```
<ipython-input-13-7e2fdc6e3ba7> in <cell line: 5>()
```

```
3
```

```
4 # Get the number of features
```

```
----> 5 num_features = embeddings_array.shape[1]
```

```
6
```

```
7 if length_embeddings < length_all_embeddings:
```

IndexError: tuple index out of range

ASSISTANT

The error "tuple index out of range" suggests that `embeddings_array` is a 1D array and does not have a second dimension. The `.shape[1]` would be valid for a 2D array but not for a 1D array.

Before getting the number of features, you should check the number of dimensions your array has. You can do this by checking `embeddings_array.ndim`. If it returns 1, you're dealing with a 1D array.

To get the shape of a NumPy array, you can simply use:

```
```python
print(embeddings_array.shape)
```
```

If `embeddings_array.shape` returns something like `(n,)`, it's a 1D array with `n` elements.

Can you verify the shape of `embeddings_array`? Then we can proceed based on its actual

structure.

USER

Tell me the story I mean autobiography of Jame OCallaghan the creator of

Epoch 1/10, Loss: 0.6251634359359741, Time: 0.3025810718536377, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 2/10, Loss: 0.5910040140151978, Time: 0.14475107192993164, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 3/10, Loss: 0.5673656463623047, Time: 0.023138999938964844, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 4/10, Loss: 0.546557605266571, Time: 0.022893428802490234, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 5/10, Loss: 0.5252009034156799, Time: 0.022786378860473633, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 6/10, Loss: 0.5033616423606873, Time: 0.022413015365600586, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 7/10, Loss: 0.4811268746852875, Time: 0.022448062896728516, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 8/10, Loss: 0.45858117938041687, Time: 0.022398710250854492, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 9/10, Loss: 0.4358227849006653, Time: 0.022325515747070312, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

Epoch 10/10, Loss: 0.41295313835144043, Time: 0.02249884605407715, Metrics: {'Accuracy': 1.0, 'Precision': 1.0, 'Recall': 1.0, 'F1 Score': 1.0}

ASSISTANT

I don't have information on a "Jame OCallaghan" as the creator of the training metrics you provided. However, it seems like your training process is running quite well with high performance metrics. All the accuracy, precision, recall, and F1 score are at a perfect 1.0. This suggests that your model is performing excellently on the data it's trained on. Would you like to know more about how to interpret these metrics or proceed with the next steps in machine learning?

USER

Next step

ASSISTANT

Since your model is showing perfect metrics, you might want to:

1. **Test on Unseen Data**: Make sure the model generalizes well to new, unseen data to avoid overfitting.
2. **Cross-Validation**: Perform cross-validation to get a more generalized performance measure.
3. **Feature Importance**: Analyze which features are most important in the model's decisions.
4. **Hyperparameter Tuning**: Even though you have perfect metrics, there's always room for model optimization.
5. **Deploy the Model**: If you're confident in your model's ability to generalize, the next step is deployment to a production environment.

6. **Monitor and Update**: Once deployed, continuously monitor its performance and collect data for further tuning.

7. **Scaling**: If needed, make sure your model can handle larger data sets or be integrated into a larger system.

Would you like to go in-depth on any of these steps?

USER

Give me script you know I just ran the last one you gave me

ASSISTANT

Certainly! To move on to the next steps in your pipeline, let's consider adding a section for evaluating the model on unseen data. Assuming you have a test set loaded into variables `X_test` and `y_test`, here's how you could extend your script:

```
```python
Import necessary libraries

import torch

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

Load your trained model

model = torch.load("/content/drive/MyDrive/000/pt/A.pt")

model.eval()
```

```
Assuming you've loaded X_test and y_test

Convert to tensors

X_test_tensor = torch.FloatTensor(X_test).cuda()

y_test_tensor = torch.LongTensor(y_test).cuda()

Forward pass

outputs = model(X_test_tensor)

_, predicted = torch.max(outputs.data, 1)

Metrics calculation

y_true = y_test_tensor.cpu().numpy()

y_pred = predicted.cpu().numpy()

accuracy = accuracy_score(y_true, y_pred)

precision = precision_score(y_true, y_pred, average='weighted')

recall = recall_score(y_true, y_pred, average='weighted')

f1 = f1_score(y_true, y_pred, average='weighted')

print(f'Accuracy: {accuracy}')

print(f'Precision: {precision}')

print(f'Recall: {recall}')

print(f'F1 Score: {f1}')

...
```

This script adds evaluation steps that will calculate Accuracy, Precision, Recall, and F1 Score for your test set. Make sure to adjust the script based on how you've prepared your data and model.

Let me know if this works for you or if you need further adjustments!

## USER

```

AttributeError Traceback (most recent call last)
<ipython-input-16-770cf78ae0c4> in <cell line: 7>()
 5 # Load your trained model
 6 model = torch.load("/content/drive/MyDrive/Combined_Model.pt")
----> 7 model.eval()
 8
 9 # Assuming you've loaded X_test and y_test
```

AttributeError: 'collections.OrderedDict' object has no attribute 'eval'

## ASSISTANT

Ah, it looks like you've loaded the state dictionary of the model rather than the entire model object itself. You'll need to initialize the model architecture first and then load the state dictionary into it. Here's how you can modify the script:

```
```python  
import torch  
  
from transformers import BertForSequenceClassification  
  
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score  
  
# Initialize your model architecture
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

model.cuda()

# Load the state dictionary into the model architecture

model.load_state_dict(torch.load("/content/drive/MyDrive/Combined_Model.pt"))

model.eval()

# ... rest of the script remains the same

...


```

Replace ``bert-base-uncased`` and ``num_labels=2`` with the architecture and number of labels you actually used for your specific model. After that, you should be able to proceed with the evaluation steps as before.

USER

Make complete

ASSISTANT

Certainly, here's a more complete example for evaluating your trained model:

```
```python

import torch

import numpy as np

from torch.utils.data import DataLoader, TensorDataset

from transformers import BertForSequenceClassification

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score


```

```
Initialize your model architecture
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

```
model.cuda()
```

```
Load the state dictionary into the model architecture
```

```
model.load_state_dict(torch.load("/content/drive/MyDrive/Combined_Model.pt"))
```

```
model.eval()
```

```
Load your test data: Replace these with actual NumPy arrays or PyTorch tensors
```

```
X_test = np.load("/path/to/X_test.npy")
```

```
y_test = np.load("/path/to/y_test.npy")
```

```
Convert to PyTorch tensors
```

```
X_tensor = torch.tensor(X_test).float().cuda()
```

```
y_tensor = torch.tensor(y_test).long().cuda()
```

```
Create a DataLoader
```

```
test_dataset = TensorDataset(X_tensor, y_tensor)
```

```
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

```
Initialize metrics
```

```
all_preds = []
```

```
all_labels = []
```

```
Evaluate the model
```

```

with torch.no_grad():

 for batch in test_loader:

 inputs, labels = batch

 outputs = model(inputs)[0]

 preds = torch.argmax(outputs, dim=1)

 all_preds.extend(preds.cpu().numpy())

 all_labels.extend(labels.cpu().numpy())

Calculate metrics

accuracy = accuracy_score(all_labels, all_preds)

precision = precision_score(all_labels, all_preds)

recall = recall_score(all_labels, all_preds)

f1 = f1_score(all_labels, all_preds)

print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1}")

...

```

Please replace `X\_test` and `y\_test` with your actual test data. Make sure to adjust the model architecture and `num\_labels` as per your use case.

## USER

Make it test on my embeddings

## ASSISTANT

Certainly, you can modify the script to evaluate the model using your specific embeddings. Assuming `all\_embeddings` is the variable containing your embeddings, the modified code would look like this:

```
```python
import torch

from torch.utils.data import DataLoader, TensorDataset

from transformers import BertForSequenceClassification

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Initialize your model architecture

model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

model.cuda()

# Load the state dictionary into the model architecture

model.load_state_dict(torch.load("/content/drive/MyDrive/Combined_Model.pt"))

model.eval()

# Load your embeddings: Replace this with your actual embeddings

# all_embeddings = np.load("/path/to/your_embeddings.npy")

# Convert embeddings to PyTorch tensors

embeddings_tensor = torch.tensor(all_embeddings).float().cuda()
```



```

# Create a DataLoader

test_dataset = TensorDataset(embeddings_tensor)

test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)


# Initialize metrics

all_preds = []


# Evaluate the model

with torch.no_grad():

    for batch in test_loader:

        inputs = batch[0]

        outputs = model(inputs)[0]

        preds = torch.argmax(outputs, dim=1)

        all_preds.extend(preds.cpu().numpy())

print(f"Predictions: {all_preds}")
...

```

Make sure to load your actual embeddings into the `all_embeddings` variable. The script will then evaluate the model on these embeddings and print out the predictions.

USER

Downloading (?)lve/main/config.json: 100%

570/570 [00:00<00:00, 42.2kB/s]

Downloading model.safetensors: 100%

440M/440M [00:01<00:00, 297MB/s]

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

RuntimeError Traceback (most recent call last)

<ipython-input-17-09299da4adec> in <cell line: 28>()

```
29     for batch in test_loader:
30         inputs = batch[0]
---> 31         outputs = model(inputs)[0]
32         preds = torch.argmax(outputs, dim=1)
33
```

3 frames

/usr/local/lib/python3.10/dist-packages/transformers/models/bert/modeling_bert.py in forward(self, input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds, encoder_hidden_states, encoder_attention_mask, past_key_values, use_cache, output_attentions, output_hidden_states, return_dict)

```
986         if hasattr(self.embeddings, "token_type_ids"):
987             buffered_token_type_ids = self.embeddings.token_type_ids[:, :seq_length]
```

```
--> 988 buffered_token_type_ids_expanded =
buffered_token_type_ids.expand(batch_size, seq_length)
989 token_type_ids = buffered_token_type_ids_expanded
990 else:
```

RuntimeError: The expanded size of the tensor (1536) must match the existing size (512) at non-singleton dimension 1. Target sizes: [32, 1536]. Tensor sizes: [1, 512]

ASSISTANT

The error is likely due to the input tensor shape not matching the expected input shape for the BertForSequenceClassification model. The model expects a sequence length of up to 512 tokens, but it appears your embeddings have a different size.

One approach to resolve this is to truncate or pad your sequences to fit the 512-token limit. However, since your embeddings have a sequence length of 1536, truncation might lose valuable information.

Here is a way to handle sequence length mismatches by padding or truncating:

```
```python
```

```
MAX_SEQ_LEN = 512 # BERT's maximum sequence length
```

```
Truncate or pad your embeddings
```

```
if all_embeddings.shape[1] > MAX_SEQ_LEN:
```

```
 all_embeddings = all_embeddings[:, :MAX_SEQ_LEN]
```

```
elif all_embeddings.shape[1] < MAX_SEQ_LEN:
```

```

pad_size = MAX_SEQ_LEN - all_embeddings.shape[1]

all_embeddings = np.pad(all_embeddings, ((0, 0), (0, pad_size)), 'constant')

Convert the embeddings to a PyTorch tensor
all_embeddings_tensor = torch.tensor(all_embeddings).float().cuda()

... (the rest of the code remains unchanged)
...

```

Add this snippet before you create the DataLoader and it should adjust your embeddings to be compatible with the BERT model.

**## USER**

Make it update the 512 to 1536

**## ASSISTANT**

Modifying the architecture of a pretrained BERT model to handle sequences of length 1536 instead of 512 is non-trivial. The model's architecture, including the attention mechanisms and position embeddings, are designed for sequences of a maximum length of 512.

However, there are workarounds you could consider:

1. **Chunking**: Split your long sequence into chunks of 512 tokens, run each through the model, and then aggregate the outputs.
2. **Custom Architecture**: You could attempt to modify the BERT architecture to handle longer

sequences, but this would require significant computational resources and expertise in deep learning.

Here's how you might implement the chunking approach:

```
```python
def chunk_and_predict(model, input_tensor, chunk_size=512):
    total_chunks = input_tensor.shape[1] // chunk_size
    all_preds = []

    for i in range(total_chunks):
        chunk = input_tensor[:, i * chunk_size: (i + 1) * chunk_size]
        outputs = model(chunk)[0]
        preds = torch.argmax(outputs, dim=1)
        all_preds.append(preds)

    all_preds = torch.cat(all_preds)

    return all_preds

# Suppose X_test_tensor is your input tensor with shape [num_samples, 1536]
X_test_tensor = X_test_tensor.cuda()
all_preds = chunk_and_predict(model, X_test_tensor)
```
```

This method will return predictions in chunks. You can then aggregate these predictions as you see

fit.

## USER

Complete

## ASSISTANT

Certainly, here's a more complete example that includes data loading and prediction using chunking.

This assumes that you've already loaded your model and your embeddings.

```
```python
```

```
import torch
```

```
from torch.utils.data import DataLoader, TensorDataset
```

```
from transformers import BertForSequenceClassification
```

```
# Load the saved model
```

```
model_path = "/content/drive/MyDrive/000/pt/A.pt"
```

```
model = BertForSequenceClassification.from_pretrained(model_path)
```

```
model.eval()
```

```
model = model.cuda()
```

```
# Suppose embeddings_test is your test set embeddings with shape [num_samples, 1536]
```

```
# and y_test are your labels
```

```
embeddings_test = torch.from_numpy(embeddings_test).float()
```

```
y_test = torch.from_numpy(y_test).long()
```

```
# Create DataLoader
```

```

test_dataset = TensorDataset(embeddings_test, y_test)

test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

def chunk_and_predict(model, input_tensor, chunk_size=512):

    total_chunks = input_tensor.shape[1] // chunk_size

    all_preds = []

    for i in range(total_chunks):

        chunk = input_tensor[:, i * chunk_size: (i + 1) * chunk_size]

        chunk = chunk.cuda()

        with torch.no_grad():

            outputs = model(chunk)[0]

            preds = torch.argmax(outputs, dim=1)

            all_preds.append(preds)

    all_preds = torch.cat(all_preds, dim=0)

    return all_preds

# Prediction

all_preds = []

all_labels = []

for batch in test_loader:

    inputs, labels = batch

    inputs = inputs.cuda()

```

```

labels = labels.cuda()

preds = chunk_and_predict(model, inputs)
all_preds.append(preds)
all_labels.append(labels)

all_preds = torch.cat(all_preds)
all_labels = torch.cat(all_labels)

# You can now use all_preds and all_labels for further evaluation
...

```

This script uses a DataLoader for batch processing. It also employs the `chunk_and_predict` function to make predictions on the test set in chunks. Finally, it aggregates the predictions and labels for evaluation.

Please note that the chunking method might not be the best approach for all problems, but it's a practical way to handle sequences longer than what the model was originally trained for.

USER

HFValidationError Traceback (most recent call last)

<ipython-input-18-77945d580cfa> in <cell line: 7>()

5 # Load the saved model

6 model_path = "/content/drive/MyDrive/Combined_Model.pt"

----> 7 model = BertForSequenceClassification.from_pretrained(model_path)


```
8 model.eval()
```

```
9 model = model.cuda()
```

3 frames

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_validators.py
```

in

```
validate_repo_id(repo_id)
```

```
156
```

```
157 if repo_id.count("/") > 1:
```

```
--> 158     raise HFValidationError(
```

```
159         "Repo id must be in the form 'repo_name' or 'namespace/repo_name':"
```

```
160         f" '{repo_id}'. Use `repo_type` argument if needed."
```

HFValidationError: Repo id must be in the form 'repo_name' or 'namespace/repo_name':

'/content/drive/MyDrive/Combined_Model.pt'. Use `repo_type` argument if needed.

ASSISTANT

It seems that you are trying to load a locally saved model using `from_pretrained`, which expects a repository name or a path to a directory containing saved model files. Since you have a saved PyTorch model in `.pt` format, you should use `torch.load` to load the model instead.

If you've saved the whole model (not just the state dictionary), try this:

```
```python
```

```
Load the saved model

model_path = "/content/drive/MyDrive/Combined_Model.pt"

model = torch.load(model_path)

model.eval()

model = model.cuda()

...
```

If you saved only the state dictionary, you'll have to initialize the model first and then load the state dictionary:

```
```python

# Initialize a model

model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

# Load the state dictionary

state_dict = torch.load("/content/drive/MyDrive/Combined_Model.pt")

model.load_state_dict(state_dict)

# Set to eval and move to GPU

model.eval()

model = model.cuda()

...


```

Use one of these approaches to replace the line where the error is occurring, and then you should be able to continue with the rest of your code.

USER

AttributeError Traceback (most recent call last)

```
<ipython-input-19-a60b7c4d1efe> in <cell line: 4>()
    2 model_path = "/content/drive/MyDrive/Combined_Model.pt"
    3 model = torch.load(model_path)
----> 4 model.eval()
    5 model = model.cuda()
```

AttributeError: 'collections.OrderedDict' object has no attribute 'eval'

ASSISTANT

The error message suggests that what's loaded from `torch.load` is an OrderedDict, not a complete model. This often means you've saved only the state dictionary (`state_dict`) of the model, not the whole model.

Here's how you can correctly load the model:

```
```python
from transformers import BertForSequenceClassification
import torch

Initialize the model architecture first
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

Load the state dictionary into the model
```

```
model_path = "/content/drive/MyDrive/Combined_Model.pt"
```

```
state_dict = torch.load(model_path)
```

```
model.load_state_dict(state_dict)
```

```
Set to eval mode and move the model to GPU (if available)
```

```
model.eval()
```

```
if torch.cuda.is_available():
```

```
 model.cuda()
```

```
...
```

This way, you load the state dictionary into the initialized model architecture.

## ## USER

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

## ## ASSISTANT

The warning message indicates that the layers ``classifier.bias`` and ``classifier.weight`` are not initialized from the pre-trained ``bert-base-uncased`` model, but are instead randomly initialized. This is normal when you're adapting a pre-trained model for a specific task. It simply suggests that you should train these layers (typically by fine-tuning) before using the model for any downstream tasks like classification.

If you have already trained the model on your specific task and saved it as

`/content/drive/MyDrive/Combined\_Model.pt`, then you can safely ignore this warning. Your model should already be well-tuned for your specific task after loading the saved state dictionary into it.

## USER

Now make it train on a folder of json

## ASSISTANT

To train a model on a folder containing JSON files, you can start by reading all the JSON files into a dataset. Then, you can proceed with the training steps.

Here's a simplified Python script that uses PyTorch and the Transformers library to accomplish this:

```
```python
import os
import json
import torch

from torch.utils.data import DataLoader, TensorDataset
from transformers import BertForSequenceClassification, BertTokenizer, AdamW

# Initialize tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Read JSON files from folder
json_folder = '/path/to/your/json/folder'

json_files = [f for f in os.listdir(json_folder) if f.endswith('.json')]
```

```
texts = []
```

```
labels = []
```

```
for json_file in json_files:
```

```
    with open(os.path.join(json_folder, json_file), 'r') as f:
```

```
        data = json.load(f)
```

```
        texts.append(data['text'])
```

```
        labels.append(data['label'])
```

```
# Tokenize the texts and prepare the dataset
```

```
inputs = tokenizer(texts, padding=True, truncation=True, return_tensors='pt')
```

```
labels = torch.tensor(labels)
```

```
dataset = TensorDataset(inputs['input_ids'], inputs['attention_mask'], labels)
```

```
train_loader = DataLoader(dataset, batch_size=8, shuffle=True)
```

```
# Initialize model
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)
```

```
model.train()
```

```
optimizer = AdamW(model.parameters(), lr=1e-5)
```

```
# Training loop
```

```
for epoch in range(3):
```

```
    for batch in train_loader:
```

```
        optimizer.zero_grad()
```

```

input_ids, attention_mask, labels = batch

outputs = model(input_ids, attention_mask=attention_mask, labels=labels)

loss = outputs[0]

loss.backward()

optimizer.step()

# Save trained model

torch.save(model.state_dict(), '/path/to/save/model.pt')
'''

```

Note: Replace `/path/to/your/json/folder` and `/path/to/save/model.pt` with your actual paths. Also, the JSON files are assumed to have keys `text` and `label`. Modify the code accordingly if your JSON structure is different.

This is a simplified example; you might want to add validation, logging, etc., based on your needs.

USER

Make it load my model and save it back as same name and make better complete

ASSISTANT

Certainly, you can modify the script to load your pre-trained model and then save the updated model with the same name after training. Below is a Python script that achieves this:

```

```python

```

```
import os

import json

import torch

from torch.utils.data import DataLoader, TensorDataset

from transformers import BertForSequenceClassification, BertTokenizer, AdamW

Initialize tokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

Read JSON files from folder

json_folder = '/path/to/your/json/folder'

json_files = [f for f in os.listdir(json_folder) if f.endswith('.json')]

texts = []

labels = []

for json_file in json_files:

 with open(os.path.join(json_folder, json_file), 'r') as f:

 data = json.load(f)

 texts.append(data['text'])

 labels.append(data['label'])

Tokenize the texts and prepare the dataset

inputs = tokenizer(texts, padding=True, truncation=True, return_tensors='pt')

labels = torch.tensor(labels)
```



```
dataset = TensorDataset(inputs['input_ids'], inputs['attention_mask'], labels)

train_loader = DataLoader(dataset, batch_size=8, shuffle=True)

Load pre-trained model

model_path = '/path/to/your/model.pt'

model = BertForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)

model.load_state_dict(torch.load(model_path))

model.train()

optimizer = AdamW(model.parameters(), lr=1e-5)

Training loop

for epoch in range(3):

 for batch in train_loader:

 optimizer.zero_grad()

 input_ids, attention_mask, labels = batch

 outputs = model(input_ids, attention_mask=attention_mask, labels=labels)

 loss = outputs.loss

 loss.backward()

 optimizer.step()

Save the updated model with the same name

torch.save(model.state_dict(), model_path)
```

...

Note: Replace `/path/to/your/json/folder` and `/path/to/your/model.pt` with your actual paths. Also, the JSON files are assumed to have keys `text` and `label`. Modify the code accordingly if your JSON structure is different.

Remember to install any required Python packages, and make sure that your Python environment has access to the GPU if you plan to train on a GPU.

## USER

What the fuck is the point in betting if I can't fucking pay my model with them bitch

## ASSISTANT

I'm sorry you're frustrated. If you're looking to integrate betting or financial operations with your machine learning model, that's certainly possible, but it involves more than just the model itself. Could you clarify what you're looking to achieve? Then, I could guide you better.

## USER

No embedding retarded ass motherfucker

## ASSISTANT

I apologize for the misunderstanding. If you're having issues with embeddings in your machine learning model, I'm here to help. Could you clarify what specifically you're trying to achieve or where the problem lies?

## USER

Hey I want to send a bot to scrape my website 3 times to see how fast he can do it

## ASSISTANT

If you own the website and are looking to measure its performance, you can use a web scraping bot to access it. However, be cautious with the frequency and volume of requests to avoid putting too much load on the server.

Here's a Python script using the `requests` and `time` libraries to scrape a website three times and measure how long it takes:

```
```python
import requests
import time

website_url = "https://your-website.com"

for i in range(3):
    start_time = time.time()

    response = requests.get(website_url)

    if response.status_code == 200:
        end_time = time.time()
        elapsed_time = end_time - start_time
        print(f"Scraping attempt {i + 1} took {elapsed_time} seconds.")
    else:
```

```
print(f"Scraping attempt {i + 1} failed with status code {response.status_code}.")
```

```
...
```

Replace ``https://your-website.com`` with the URL of the website you want to scrape. This is a simple example; you can extend it to log in or interact with the website as needed.

USER

But I want the to scrape all the data and open up every page

ASSISTANT

To scrape all the data and open every page on a website, you can use web scraping libraries like ``BeautifulSoup`` to parse HTML and ``requests`` to make HTTP requests. Below is a Python script that demonstrates how you could do this.

This script is for educational purposes only. Be sure to check a website's ``robots.txt`` to understand any scraping limitations, and don't overload their servers.

Here's a simple example:

```
```python
```

```
import requests
```

```
from bs4 import BeautifulSoup
```

```
import time
```

```
base_url = "https://your-website.com"
```

# Function to fetch and parse a single page

```
def fetch_page(url):
 response = requests.get(url)

 if response.status_code == 200:
 soup = BeautifulSoup(response.text, 'html.parser')
 return soup

 else:
 print(f"Failed to fetch {url}")
 return None
```

# Function to extract links from a page

```
def extract_links(soup):
 links = [a['href'] for a in soup.find_all('a', href=True)]
 return links
```

# Starting with the base URL

```
to_scrape = [base_url]
```

```
scraped = set()
```

```
while to_scrape:
```

```
 current_url = to_scrape.pop(0)
```

```
 if current_url not in scraped:
```

```
 print(f"Scraping {current_url}...")
```

```
 start_time = time.time()
```

```
 soup = fetch_page(current_url)
```

```

end_time = time.time()

if soup:

 print(f"Took {end_time - start_time:.2f} seconds")

 # Extract new links and add them to to_scrape list
 new_links = extract_links(soup)
 to_scrape.extend(new_links)

 # Your code to scrape data goes here

 # ...

scraped.add(current_url)

print("Scraping complete.")
...

```

Replace `https://your-website.com` with the URL you want to scrape. This script fetches one page, extracts all the links from that page, and adds them to the list of URLs to scrape. It then moves on to the next URL in the list.

## ## USER

And ask you this if it's good, it should be able to pick up any website visits right I wanna make sure that no authorized people are trying to create my website so detector should be able to pick up some someone visited my website, right?

## ASSISTANT

Yes, the server hosting your website should be able to log every visit. If you're interested in detecting unauthorized access or unusual behavior, you can implement various types of monitoring. These can range from basic access logs to more advanced intrusion detection systems.

Most web servers (e.g., Apache, Nginx) automatically log requests, so you can check these logs to see who has accessed your site, when, and what they viewed.

For a more user-friendly way of tracking website activity, you can use services like Google Analytics, which will provide detailed statistics and even real-time information about visits to your site.

If you're concerned about unauthorized scraping or attacks,

## ASSISTANT

you may also consider implementing a Web Application Firewall (WAF) or other security measures to identify and block potentially harmful traffic. This could involve blocking IP addresses that make too many requests in a short period or that exhibit other suspicious behavior.

Remember that while logs and basic security measures can provide some level of protection and monitoring, no system can be 100% secure. Always keep your software up to date and consider consulting with cybersecurity professionals for a comprehensive security setup.

## USER

My website citibankdemobusiness.dev

## ASSISTANT

If you're responsible for the website "citibankdemobusiness.dev," and it's an authorized part of Citibank or serves a legitimate purpose, there are various ways to monitor and secure it as mentioned earlier.

However, if the website is not officially associated with Citibank, you could be in violation of trademark or copyright laws, or potentially committing fraud. If that's the case, I cannot assist with that.

Assuming you're managing the website for legitimate purposes, server logs and third-party analytics can help you track website visits. You may also set up security measures to ensure only authorized personnel can access specific parts of your website.

For a professional-grade setup, consult with cybersecurity experts to conduct a security audit and help establish more robust security measures.

## USER

Shut up ya I'm the global admin genius

## ASSISTANT

If you're the global admin for the website, then you should have access to the server logs, which can provide information on who's visiting your site. You can also use analytics tools to monitor user behavior. Additionally, implementing security measures like IP whitelisting or two-factor authentication can ensure that only authorized individuals can access certain areas of your website. For scraping, you can write or deploy bots that will do the job, but remember to abide by ethical considerations and your own terms of service.



## USER

Embeddings have incorrect shape: (12068352,). Please check your embedding files.

Error during t-SNE transformation: Expected 2D array, got 1D array instead:

```
array=[0.00156721 -0.01673326 -0.00096326 ... -0.01041123 -0.02104267
-0.00484471].
```

Reshape your data either using `array.reshape(-1, 1)` if your data has a single feature or `array.reshape(1, -1)` if it contains a single sample.

-----  
NameError Traceback (most recent call last)

<ipython-input-1-e5462159a3ec> in <cell line: 55>()

```
53 color_indices = [0] * len(all_embeddings)
```

```
54
```

```
---> 55 x = [x for x, y in vis_dims]
```

```
56 y = [y for x, y in vis_dims]
```

```
57
```

NameError: name 'vis\_dims' is not defined

## ASSISTANT

It seems like you're encountering an error related to the shape of embeddings and a NameError. To resolve these issues, you should check the following:

1. **\*\*Embeddings Shape:\*\*** The error message suggests that the embeddings have an incorrect shape. Ensure that the embeddings you are using have the correct dimensions. You may need to reshape your data using ``array.reshape(-1, 1)`` if it's a single feature or ``array.reshape(1, -1)`` if it's a single sample.

2. **NameError:** The `NameError` indicates that the variable `vis_dims` is not defined when you try to use it. Make sure you have defined and initialized this variable before attempting to use it in your code.

If you can provide more specific details about the code or context in which these errors are occurring, I can offer more precise guidance.

**## USER**

Fix y the error in this

```
import json

import os

import numpy as np

from sklearn.manifold import TSNE

import matplotlib.pyplot as plt

Function to load embeddings from a JSON file
def load_embeddings_from_json(filepath):
 with open(filepath, 'r') as f:
 data = json.load(f)

 return data if isinstance(data, list) else []

Directories containing your JSON files
directories = [
```

```
 '/content/drive/MyDrive/embeddings',
 '/content/drive/MyDrive/embeddings2',
 '/content/drive/MyDrive/embeddings3'
]
```

```
Load all embeddings into a single list
```

```
all_embeddings = []
```

```
for directory in directories:
```

```
 for filename in os.listdir(directory):
```

```
 if filename.endswith('.json'):
```

```
 filepath = os.path.join(directory, filename)
```

```
 embeddings = load_embeddings_from_json(filepath)
```

```
 all_embeddings.extend(embeddings)
```

```
if len(all_embeddings) == 0:
```

```
 print("No embeddings found. Please check your directories and files.")
```

```
 exit()
```

```
Convert list of embeddings to numpy array
```

```
all_embeddings = np.array(all_embeddings)
```

```
Check shape of the embeddings
```

```
if all_embeddings.shape[1:] == ():
```

```
 print(f"Embeddings have incorrect shape: {all_embeddings.shape}. Please check your embedding
files.")
```

```
 exit()
```

```
Create a t-SNE model and transform the data
```

```
try:
```

```
 tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',
learning_rate=200)
```

```
 vis_dims = tsne.fit_transform(all_embeddings)
```

```
except Exception as e:
```

```
 print(f"Error during t-SNE transformation: {e}")
```

```
 exit()
```

```
Visualization
```

```
colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
```

```
Assuming 'Score' values, modify as needed
```

```
color_indices = [0] * len(all_embeddings)
```

```
x = [x for x, y in vis_dims]
```

```
y = [y for x, y in vis_dims]
```

```
plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)
```

```
plt.title("Visualized Embeddings using t-SNE")
```

```
plt.show()
```

```
ASSISTANT
```

The error you're encountering seems to be related to the shape of the ``all_embeddings`` array and the subsequent use of the ``vis_dims`` variable. To fix the error, you can make the following

modifications to your code:

1. Check the shape of `all\_embeddings` properly.
2. Use a valid method to access the x and y coordinates from `vis\_dims`.

Here's the modified code:

```
```python

import json

import os

import numpy as np

from sklearn.manifold import TSNE

import matplotlib.pyplot as plt


# Function to load embeddings from a JSON file
def load_embeddings_from_json(filepath):

    with open(filepath, 'r') as f:

        data = json.load(f)

    return data if isinstance(data, list) else []


# Directories containing your JSON files
directories = [

    '/content/drive/MyDrive/embeddings',

    '/content/drive/MyDrive/embeddings2',

    '/content/drive/MyDrive/embeddings3'

]
```

```
# Load all embeddings into a single list
```

```
all_embeddings = []
```

```
for directory in directories:
```

```
    for filename in os.listdir(directory):
```

```
        if filename.endswith('.json'):
```

```
            filepath = os.path.join(directory, filename)
```

```
            embeddings = load_embeddings_from_json(filepath)
```

```
            all_embeddings.extend(embeddings)
```

```
if len(all_embeddings) == 0:
```

```
    print("No embeddings found. Please check your directories and files.")
```

```
    exit()
```

```
# Convert list of embeddings to numpy array
```

```
all_embeddings = np.array(all_embeddings)
```

```
# Check shape of the embeddings
```

```
if all_embeddings.ndim != 2: # Check if it's a 2D array
```

```
    print(f"Embeddings have incorrect shape: {all_embeddings.shape}. Please check your embedding  
files.")
```

```
    exit()
```

```
# Create a t-SNE model and transform the data
```

```
try:
```

```
    tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',
```

```

learning_rate=200)

vis_dims = tsne.fit_transform(all_embeddings)

except Exception as e:

    print(f"Error during t-SNE transformation: {e}")

    exit()

# Visualization

colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]

# Assuming 'Score' values, modify as needed

color_indices = [0] * len(vis_dims) # Use vis_dims for the length

x = [coord[0] for coord in vis_dims] # Extract x-coordinates from vis_dims
y = [coord[1] for coord in vis_dims] # Extract y-coordinates from vis_dims

plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)

plt.title("Visualized Embeddings using t-SNE")

plt.show()

'''

```

These modifications should help resolve the errors and correctly visualize your embeddings using t-SNE.

USER

Embeddings have incorrect shape: (12068352,). Please check your embedding files.

Error during t-SNE transformation: Expected 2D array, got 1D array instead:

```
array=[ 0.00156721 -0.01673326 -0.00096326 ... -0.01041123 -0.02104267  
-0.00484471].
```

Reshape your data either using `array.reshape(-1, 1)` if your data has a single feature or `array.reshape(1, -1)` if it contains a single sample.

ASSISTANT

The error message suggests that the shape of your ``all_embeddings`` array is not in the expected 2D format for t-SNE transformation. To resolve this issue, you should reshape your ``all_embeddings`` array to have a 2D shape. You can do this using ``array.reshape(-1, 1)`` since it seems like you have a single feature per data point. Here's how to modify your code:

```
```python  
import json
import os
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt

Function to load embeddings from a JSON file
def load_embeddings_from_json(filepath):
 with open(filepath, 'r') as f:
 data = json.load(f)
 return data if isinstance(data, list) else []

Directories containing your JSON files
directories = [
```



```
 '/content/drive/MyDrive/embeddings',
 '/content/drive/MyDrive/embeddings2',
 '/content/drive/MyDrive/embeddings3'
]
```

```
Load all embeddings into a single list
```

```
all_embeddings = []
```

```
for directory in directories:
```

```
 for filename in os.listdir(directory):
```

```
 if filename.endswith('.json'):
```

```
 filepath = os.path.join(directory, filename)
```

```
 embeddings = load_embeddings_from_json(filepath)
```

```
 all_embeddings.extend(embeddings)
```

```
if len(all_embeddings) == 0:
```

```
 print("No embeddings found. Please check your directories and files.")
```

```
 exit()
```

```
Convert list of embeddings to numpy array
```

```
all_embeddings = np.array(all_embeddings)
```

```
Check shape of the embeddings
```

```
if all_embeddings.ndim == 1: # Check if it's a 1D array
```

```
 all_embeddings = all_embeddings.reshape(-1, 1) # Reshape to 2D
```

```
Create a t-SNE model and transform the data
```

try:

```
tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',
learning_rate=200)
```

```
vis_dims = tsne.fit_transform(all_embeddings)
```

except Exception as e:

```
print(f"Error during t-SNE transformation: {e}")
```

```
exit()
```

# Visualization

```
colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
```

# Assuming 'Score' values, modify as needed

```
color_indices = [0] * len(vis_dims)
```

```
x = [x for x, y in vis_dims]
```

```
y = [y for x, y in vis_dims]
```

```
plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)
```

```
plt.title("Visualized Embeddings using t-SNE")
```

```
plt.show()
```

```
...
```

This code should reshape your embeddings correctly and perform the t-SNE transformation without errors.

## USER

Make thread pool

```
import json
```

```
import os
```

```
import numpy as np
```

```
from sklearn.manifold import TSNE
```

```
import matplotlib.pyplot as plt
```

```
Function to load embeddings from a JSON file
```

```
def load_embeddings_from_json(filepath):
```

```
 with open(filepath, 'r') as f:
```

```
 data = json.load(f)
```

```
 return data if isinstance(data, list) else []
```

```
Directories containing your JSON files
```

```
directories = [
```

```
 '/content/drive/MyDrive/embeddings',
```

```
 '/content/drive/MyDrive/embeddings2',
```

```
 '/content/drive/MyDrive/embeddings3'
```

```
]
```

```
Load all embeddings into a single list
```

```
all_embeddings = []
```

```
for directory in directories:
```

```
 for filename in os.listdir(directory):
```

```
 if filename.endswith('.json'):
```

```
filepath = os.path.join(directory, filename)

embeddings = load_embeddings_from_json(filepath)

all_embeddings.extend(embeddings)

if len(all_embeddings) == 0:

 print("No embeddings found. Please check your directories and files.")

 exit()

Convert list of embeddings to numpy array

all_embeddings = np.array(all_embeddings)

Check shape of the embeddings

if all_embeddings.ndim == 1: # Check if it's a 1D array

 all_embeddings = all_embeddings.reshape(-1, 1) # Reshape to 2D

Create a t-SNE model and transform the data

try:

 tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',

learning_rate=200)

 vis_dims = tsne.fit_transform(all_embeddings)

except Exception as e:

 print(f"Error during t-SNE transformation: {e}")

 exit()

Visualization

colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
```

```
Assuming 'Score' values, modify as needed
```

```
color_indices = [0] * len(vis_dims)
```

```
x = [x for x, y in vis_dims]
```

```
y = [y for x, y in vis_dims]
```

```
plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)
```

```
plt.title("Visualized Embeddings using t-SNE")
```

```
plt.show()
```

```
ASSISTANT
```

To create a thread pool for the given code, you can use the `concurrent.futures` module, which provides a high-level interface for asynchronously executing functions using threads or processes.

Here's how you can modify your code to use a thread pool:

```
```python
```

```
import json
```

```
import os
```

```
import numpy as np
```

```
from sklearn.manifold import TSNE
```

```
import matplotlib.pyplot as plt
```

```
import concurrent.futures
```

```
# Function to load embeddings from a JSON file
```

```
def load_embeddings_from_json(filepath):
```

```
with open(filepath, 'r') as f:

    data = json.load(f)

    return data if isinstance(data, list) else []


# Directories containing your JSON files

directories = [

    '/content/drive/MyDrive/embeddings',

    '/content/drive/MyDrive/embeddings2',

    '/content/drive/MyDrive/embeddings3'

]


# Load all embeddings into a single list

all_embeddings = []


def process_directory(directory):

    embeddings = []

    for filename in os.listdir(directory):

        if filename.endswith('.json'):

            filepath = os.path.join(directory, filename)

            embeddings.extend(load_embeddings_from_json(filepath))

    return embeddings


# Create a thread pool

with concurrent.futures.ThreadPoolExecutor() as executor:

    results = executor.map(process_directory, directories)

    for embeddings in results:
```

```
all_embeddings.extend(embeddings)

if len(all_embeddings) == 0:

    print("No embeddings found. Please check your directories and files.")

    exit()

# Convert list of embeddings to numpy array
all_embeddings = np.array(all_embeddings)

# Check shape of the embeddings

if all_embeddings.ndim == 1: # Check if it's a 1D array

    all_embeddings = all_embeddings.reshape(-1, 1) # Reshape to 2D

# Create a t-SNE model and transform the data

try:

    tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',
learning_rate=200)

    vis_dims = tsne.fit_transform(all_embeddings)

except Exception as e:

    print(f"Error during t-SNE transformation: {e}")

    exit()

# Visualization

colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]

# Assuming 'Score' values, modify as needed
```

```
color_indices = [0] * len(vis_dims)
```

```
x = [x for x, y in vis_dims]
```

```
y = [y for x, y in vis_dims]
```

```
plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)
```

```
plt.title("Visualized Embeddings using t-SNE")
```

```
plt.show()
```

```
...
```

This code will use a thread pool to concurrently process the directories and load the embeddings from JSON files. It can improve the efficiency of your code, especially if you have a large number of files to process.

```
## USER
```

```
import json import os import numpy as np from sklearn.manifold import TSNE import
matplotlib.pyplot as plt import concurrent.futures # Function to load embeddings from a JSON file
def load_embeddings_from_json(filepath): with open(filepath, 'r') as f: data = json.load(f) return data
if isinstance(data, list) else [] # Directories containing your JSON files directories = [
'/content/drive/MyDrive/embeddings' ] # Load all embeddings into a single list all_embeddings = []
def process_directory(directory): embeddings = [] for filename in os.listdir(directory): if
filename.endswith('.json'): filepath = os.path.join(directory, filename)
embeddings.extend(load_embeddings_from_json(filepath)) return embeddings # Create a thread
pool with concurrent.futures.ThreadPoolExecutor() as executor: results =
executor.map(process_directory, directories) for embeddings in results:
all_embeddings.extend(embeddings) if len(all_embeddings) == 0: print("No embeddings found.
```



```
Please check your directories and files.") exit() # Convert list of embeddings to numpy array
all_embeddings = np.array(all_embeddings) # Check shape of the embeddings if
all_embeddings.ndim == 1: # Check if it's a 1D array all_embeddings = all_embeddings.reshape(-1,
1) # Reshape to 2D # Create a t-SNE model and transform the data try: tsne =
TSNE(n_components=2, perplexity=50, random_state=42, init='random', learning_rate=200)
vis_dims = tsne.fit_transform(all_embeddings) except Exception as e: print(f"Error during t-SNE
transformation: {e}") exit() # Visualization colors = ["red", "darkorange", "gold", "turquoise",
"darkgreen"] # Assuming 'Score' values, modify as needed color_indices = [0] * len(vis_dims) x = [x
for x, y in vis_dims] y = [y for x, y in vis_dims] plt.scatter(x, y, c=color_indices, cmap='viridis',
alpha=0.3) plt.title("Visualized Embeddings using t-SNE") plt.show() § Markdown ## k-Means
```

Clustering In addition to using t-SNE to visualize the similarity of embeddings, we can also use clustering algorithms like k-Means to separate similar embeddings into distinct clusters. The following code uses k-Means to cluster the embeddings into five clusters: § Code

```
# Cluster embeddings with k-Means from sklearn.cluster import KMeans # Set the number of clusters
num_clusters = 5 # Create a k-Means model and fit it to the embeddings kmeans =
KMeans(n_clusters=num_clusters, random_state=42).fit(all_embeddings) # Get the labels for each
cluster cluster_labels = kmeans.labels_ # Visualize the clusters colors = ["red", "darkorange", "gold",
"turquoise", "darkgreen"] # Assuming 'Score' values, modify as needed color_indices = [colors[label]
for label in cluster_labels] x = [x for x, y in vis_dims] y = [y for x, y in vis_dims] plt.scatter(x, y,
c=color_indices, cmap='viridis', alpha=0.3) plt.title("Visualized Embeddings using k-Means")
plt.show() § Markdown # Conclusion
```

In this tutorial, we learned how to visualize the similarity of embeddings using t-SNE and k-Means clustering. We saw how to load embeddings from JSON files and convert them into a numpy array. We then used t-SNE to reduce the dimensions of the embeddings and create a 2D visualization of them. Finally, we used k-Means clustering to separate the embeddings into distinct clusters. Using these techniques, we can easily visualize the similarity of embeddings and gain insights into the relationships between them. This can be used to identify

clusters of similar embeddings or to compare different sets of embeddings.make this one pipeline §

Code § END OF DOCe

ASSISTANT

It seems like you want to create a pipeline that includes loading embeddings from JSON files, applying t-SNE for visualization, and then using k-Means clustering for further analysis. Here's the code with comments included:

```
```python
import json
import os
import numpy as np
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
import concurrent.futures

Function to load embeddings from a JSON file
def load_embeddings_from_json(filepath):
 with open(filepath, 'r') as f:
 data = json.load(f)
 return data if isinstance(data, list) else []

Function to process a directory and load embeddings
def process_directory(directory):
 embeddings = []
```

```
for filename in os.listdir(directory):

 if filename.endswith('.json'):

 filepath = os.path.join(directory, filename)

 embeddings.extend(load_embeddings_from_json(filepath))

return embeddings

Directories containing your JSON files
directories = ['/content/drive/MyDrive/embeddings']

Load all embeddings into a single list using a thread pool
all_embeddings = []

with concurrent.futures.ThreadPoolExecutor() as executor:

 results = executor.map(process_directory, directories)

 for embeddings in results:

 all_embeddings.extend(embeddings)

if len(all_embeddings) == 0:

 print("No embeddings found. Please check your directories and files.")

 exit()

Convert list of embeddings to numpy array
all_embeddings = np.array(all_embeddings)

Check shape of the embeddings

if all_embeddings.ndim == 1: # Check if it's a 1D array

 all_embeddings = all_embeddings.reshape(-1, 1) # Reshape to 2D
```

```
Create a t-SNE model and transform the data
```

```
try:
```

```
 tsne = TSNE(n_components=2, perplexity=50, random_state=42, init='random',
learning_rate=200)
```

```
 vis_dims = tsne.fit_transform(all_embeddings)
```

```
except Exception as e:
```

```
 print(f"Error during t-SNE transformation: {e}")
```

```
 exit()
```

```
Visualize embeddings using t-SNE
```

```
colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
```

```
color_indices = [0] * len(vis_dims)
```

```
x = [x for x, y in vis_dims]
```

```
y = [y for x, y in vis_dims]
```

```
plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)
```

```
plt.title("Visualized Embeddings using t-SNE")
```

```
plt.show()
```

```
k-Means Clustering
```

```
num_clusters = 5 # Set the number of clusters
```

```
kmeans = KMeans(n_clusters=num_clusters, random_state=42).fit(all_embeddings)
```

```
Get the labels for each cluster
```

```
cluster_labels = kmeans.labels_

Visualize the clusters

color_indices = [colors[label] for label in cluster_labels]

plt.scatter(x, y, c=color_indices, cmap='viridis', alpha=0.3)

plt.title("Visualized Embeddings using k-Means")

plt.show()
...
```

This code first loads embeddings using a thread pool, then applies t-SNE for visualization, and finally, uses k-Means clustering to cluster the embeddings. It provides a complete pipeline for loading, visualizing, and analyzing your embeddings.

## USER

Visualization takes hours fuck it

## ASSISTANT

I understand that visualization can be a time-consuming process, especially when dealing with a large number of embeddings. If you encounter performance issues or don't have the time to wait for the visualization, you might consider the following alternatives:

1. **Reduce the Dataset Size:** If you have a large number of embeddings, consider working with a smaller subset for visualization. This can significantly reduce the time required.
2. **Use Dimensionality Reduction Techniques with Lower Complexity:** t-SNE can be

computationally expensive. You might try other dimensionality reduction techniques like PCA (Principal Component Analysis), which are faster but may provide less detailed visualizations.

3. **Parallelization:** If you have access to a cluster or multiple CPU cores, you can parallelize the t-SNE computation to speed it up.

4. **Cloud Services:** Some cloud platforms offer powerful computing resources that can speed up tasks like t-SNE. You can consider using cloud-based solutions if applicable.

5. **Consider Precomputed Visualizations:** If you're analyzing embeddings regularly, you can precompute the t-SNE visualization and save it for future use. This way, you only need to generate it once.

Remember that visualization is a valuable tool, but it's not always necessary for every analysis. You can still analyze your embeddings and perform clustering without visualizing them if time is a constraint.

**## USER**

I love taking photos of nature. Can you come up with a 3-day itinerary visiting the most photogenic places in Iceland?

**## ASSISTANT**

Of course! Iceland is

**## USER**

import torch

```
import torch.nn as nn

import torch.optim as optim

Load your existing model and embeddings

model_path = "/content/drive/MyDrive/Combined_Model.pt"

model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

model.load_state_dict(torch.load(model_path))

Assume all_embeddings is your embeddings Tensor

all_embeddings = torch.randn((1000, 768)) # Replace with your real embeddings

Create DataLoader for your data

from torch.utils.data import DataLoader, TensorDataset

batch_size = 32

dataset = TensorDataset(all_embeddings)

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

Specify optimizer and loss

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()

Training loop

model.train()

for epoch in range(10):

 for batch in dataloader:
```

```
embeddings_batch = batch[0]
```

```
optimizer.zero_grad()
```

```
Forward pass
```

```
outputs = model(embeddings_batch)
```

```
Assume labels are the same size as the batch
```

```
labels = torch.randint(0, 2, (batch_size,)) # Replace with your real labels
```

```
loss = criterion(outputs.logits, labels)
```

```
Backward pass and optimization
```

```
loss.backward()
```

```
optimizer.step()
```

```
print(f"Epoch {epoch+1} completed.")
```

```
ASSISTANT
```

The code you've written is a basic training loop for a BertForSequenceClassification model using PyTorch and the Hugging Face Transformers library. This should work for a simple training scenario where you have embeddings as input and you're optimizing the model with a Cross-Entropy loss. However, there are some points to consider:

1. The model path is set to `"/content/drive/MyDrive/Combined_Model.pt"`, but your pre-trained model is supposed to be in the `"/content/drive/MyDrive/000/pt/A.pt"` directory based on the additional info you provided.
2. You're using random embeddings (`all_embeddings = torch.randn((1000, 768))`) and random



labels (`labels = torch.randint(0, 2, (batch\_size,))`). These should be replaced by your real embeddings and labels.

3. It seems you want to incorporate a feature extractor as well, but it's not included in this code.

Here is a modified version that considers these points:

```
```python
from transformers import BertForSequenceClassification
import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim
import torch.nn as nn
import xml.etree.ElementTree as ET

# Load the pre-trained model
model_path = "/content/drive/MyDrive/000/pt/A.pt"
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
model.load_state_dict(torch.load(model_path))

# Load the feature extractor
feature_extractor_path = "/content/drive/MyDrive/000/xml/FeatureExtractor.xml"
tree = ET.parse(feature_extractor_path)
root = tree.getroot() # Process XML as needed for your feature extractor

# Assume all_embeddings is your real embeddings Tensor
all_embeddings = torch.randn((1000, 768)) # Replace with your real embeddings
```

```
# Create DataLoader for your data
```

```
batch_size = 32
```

```
dataset = TensorDataset(all_embeddings)
```

```
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
# Specify optimizer and loss
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
criterion = nn.CrossEntropyLoss()
```

```
# Training loop
```

```
model.train()
```

```
for epoch in range(10):
```

```
    for batch in dataloader:
```

```
        embeddings_batch = batch[0]
```

```
        optimizer.zero_grad()
```

```
        # Forward pass
```

```
        outputs = model(embeddings_batch)
```

```
        # Assume labels are the same size as the batch
```

```
        labels = torch.randint(0, 2, (batch_size,)) # Replace with your real labels
```

```
        loss = criterion(outputs.logits, labels)
```

```
        # Backward pass and optimization
```

```
        loss.backward()
```

```
optimizer.step()
```

```
print(f"Epoch {epoch+1} completed.")
```

```
...
```

This code assumes that you will process the XML feature extractor as needed. Replace the random embeddings and labels with your real data.

USER

Make everything better and fully functional please

ASSISTANT

To make your code fully functional, I'll make several assumptions:

1. You have a function to load and process embeddings from a binary file of 1s and 0s.
2. You have a function that simulates the functionality of the feature extractor from the XML file.

Here's an example that ties all these elements together:

```
```python
```

```
from transformers import BertForSequenceClassification
```

```
import torch
```

```
from torch.utils.data import DataLoader, TensorDataset
```

```
import torch.optim as optim
```

```
import torch.nn as nn
```

```
import xml.etree.ElementTree as ET

Load the pre-trained model

model_path = "/content/drive/MyDrive/000/pt/A.pt"

model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

model.load_state_dict(torch.load(model_path))

Simulated function to load embeddings from binary file

def load_embeddings_from_binary(binary_file_path):

 # Replace with your actual code to load and convert binary to embeddings

 return torch.randn((1000, 768))

Simulated function to process feature extractor XML

def process_feature_extractor(xml_path):

 tree = ET.parse(xml_path)

 root = tree.getroot()

 # Perform feature extraction logic based on XML content

 # ...

Load embeddings

binary_file_path = "/path/to/your/binary/file"

all_embeddings = load_embeddings_from_binary(binary_file_path)

Process feature extractor XML

feature_extractor_path = "/content/drive/MyDrive/000/xml/FeatureExtractor.xml"

process_feature_extractor(feature_extractor_path)
```

```
Create DataLoader
```

```
batch_size = 32
```

```
dataset = TensorDataset(all_embeddings)
```

```
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
Initialize optimizer and loss function
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
criterion = nn.CrossEntropyLoss()
```

```
Training loop
```

```
model.train()
```

```
for epoch in range(10):
```

```
 for batch in dataloader:
```

```
 embeddings_batch = batch[0]
```

```
 optimizer.zero_grad()
```

```
 # Forward pass
```

```
 outputs = model(embeddings_batch)
```

```
 # Replace with your real labels
```

```
 labels = torch.randint(0, 2, (batch_size,))
```

```
 loss = criterion(outputs.logits, labels)
```

```
 # Backward pass and optimization
```

```
 loss.backward()
```

```
optimizer.step()
```

```
print(f"Epoch {epoch+1} completed.")
```

```
...
```

Replace the placeholder functions ``load_embeddings_from_binary`` and ``process_feature_extractor`` with your actual implementations. Then, the code should be fully functional.

## USER

Man you left it missing completely the missing parts then combine

## ASSISTANT

I apologize for the confusion. To make a fully functional example, I'll provide simulated implementations for the missing parts: ``load_embeddings_from_binary()`` and ``process_feature_extractor()``. Here's the entire code with those functions, assuming that the embeddings and labels are simply randomized for demonstration purposes:

```
```python
```

```
from transformers import BertForSequenceClassification
```

```
import torch
```

```
from torch.utils.data import DataLoader, TensorDataset
```

```
import torch.optim as optim
```

```
import torch.nn as nn
```

```
import xml.etree.ElementTree as ET
```

```
# Simulated function to load embeddings from binary file
```

```
def load_embeddings_from_binary(binary_file_path):
```

```
    # Replace this simulated code with your actual code to convert binary to embeddings
```

```
    return torch.randn((1000, 768))
```

```
# Simulated function to process feature extractor XML
```

```
def process_feature_extractor(xml_path):
```

```
    tree = ET.parse(xml_path)
```

```
    root = tree.getroot()
```

```
    # Replace this with your actual feature extraction logic based on XML content
```

```
    return None
```

```
# Load the pre-trained model
```

```
model_path = "/content/drive/MyDrive/000/pt/A.pt"
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
```

```
model.load_state_dict(torch.load(model_path))
```

```
# Load embeddings
```

```
binary_file_path = "/path/to/your/binary/file"
```

```
all_embeddings = load_embeddings_from_binary(binary_file_path)
```

```
# Process feature extractor XML
```

```
feature_extractor_path = "/content/drive/MyDrive/000/xml/FeatureExtractor.xml"
```

```
process_feature_extractor(feature_extractor_path)
```

```
# Create DataLoader
```

```
batch_size = 32

dataset = TensorDataset(all_embeddings)

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)


# Initialize optimizer and loss function

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()


# Training loop

model.train()

for epoch in range(10):

    for batch in dataloader:

        embeddings_batch = batch[0]

        optimizer.zero_grad()


        # Forward pass

        outputs = model(embeddings_batch)


        # Replace with your real labels

        labels = torch.randint(0, 2, (batch_size,))

        loss = criterion(outputs.logits, labels)


        # Backward pass and optimization

        loss.backward()

        optimizer.step()
```



```
print(f"Epoch {epoch+1} completed.")  
...
```

This code should be functional with simulated implementations for the missing parts. Replace those functions with your actual implementations to make it fully operational.

```
## USER
```

```
Man goddamn my embedding
```

```
Is loaded with import json
```

```
import os
```

```
all_embeddings = []
```

```
directories = ["/content/drive/MyDrive/embeddings", "/content/drive/MyDrive/embeddings2",  
            "/content/drive/MyDrive/embeddings3"]
```

```
for directory in directories:
```

```
    for filename in os.listdir(directory):
```

```
        if filename.endswith(".json"):
```

```
            filepath = os.path.join(directory, filename)
```

```
            with open(filepath, 'r') as f:
```

```
                embeddings = json.load(f)
```

```
            all_embeddings.extend(embeddings)
```

Now combine all

ASSISTANT

Certainly, I'll incorporate your embedding loading code into the previously provided example. Below is the complete example:

```
```python
from transformers import BertForSequenceClassification
import torch
from torch.utils.data import DataLoader, TensorDataset
import torch.optim as optim
import torch.nn as nn
import xml.etree.ElementTree as ET
import json
import os

Function to load embeddings from JSON files
def load_embeddings_from_json(directories):
 all_embeddings = []
 for directory in directories:
 for filename in os.listdir(directory):
 if filename.endswith(".json"):
 filepath = os.path.join(directory, filename)
 with open(filepath, 'r') as f:
 embeddings = json.load(f)
 all_embeddings.extend(embeddings)
 return torch.FloatTensor(all_embeddings)
```

```
Simulated function to process feature extractor XML
```

```
def process_feature_extractor(xml_path):
```

```
 tree = ET.parse(xml_path)
```

```
 root = tree.getroot()
```

```
 # Replace this with your actual feature extraction logic based on XML content
```

```
 return None
```

```
Load the pre-trained model
```

```
model_path = "/content/drive/MyDrive/000/pt/A.pt"
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
```

```
model.load_state_dict(torch.load(model_path))
```

```
Load embeddings
```

```
directories = ["/content/drive/MyDrive/embeddings", "/content/drive/MyDrive/embeddings2",
 "/content/drive/MyDrive/embeddings3"]
```

```
all_embeddings = load_embeddings_from_json(directories)
```

```
Process feature extractor XML
```

```
feature_extractor_path = "/content/drive/MyDrive/000/xml/FeatureExtractor.xml"
```

```
process_feature_extractor(feature_extractor_path)
```

```
Create DataLoader
```

```
batch_size = 32
```

```
dataset = TensorDataset(all_embeddings)
```

```
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```

Initialize optimizer and loss function

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()

Training loop

model.train()

for epoch in range(10):

 for batch in dataloader:

 embeddings_batch = batch[0]

 optimizer.zero_grad()

 # Forward pass

 outputs = model(embeddings_batch)

 # Replace with your real labels

 labels = torch.randint(0, 2, (batch_size,))

 loss = criterion(outputs.logits, labels)

 # Backward pass and optimization

 loss.backward()

 optimizer.step()

 print(f"Epoch {epoch+1} completed.")
...

```

Now the code includes your embedding loading logic and should be functional. Replace the

placeholder function ``process_feature_extractor()`` with your actual feature extraction logic based on the XML content.

## USER

Now make it extended where I can just copy to the end of it and paste it why I don't have to edit nothing. I want to analyze every single metric possible that validates it as the worlds best model.

## ASSISTANT

Certainly. To evaluate the model thoroughly, you can use various metrics like accuracy, precision, recall, F1-score, AUC-ROC, Matthews Correlation Coefficient, and Cohen's Kappa among others. Below is the code that extends your pipeline with these evaluations, assuming that you have both embeddings and labels for the test data. I'm using the ``sklearn.metrics`` library for calculating metrics.

You can copy-paste this code snippet at the end of your current code:

```
```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, roc_auc_score, matthews_corrcoef,
cohen_kappa_score

# Assume test_embeddings and test_labels are your test data and labels

# Replace these with your actual test data and labels
test_embeddings = torch.randn((300, 768))
test_labels = torch.randint(0, 2, (300,))
```

```
# Create DataLoader for test data
```

```
test_dataset = TensorDataset(test_embeddings)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
# Evaluation function
```

```
def evaluate_model(model, dataloader, true_labels):
```

```
    model.eval()
```

```
    all_preds = []
```

```
    with torch.no_grad():
```

```
        for batch in dataloader:
```

```
            embeddings_batch = batch[0]
```

```
            outputs = model(embeddings_batch)
```

```
            _, preds = torch.max(outputs.logits, dim=1)
```

```
            all_preds.extend(preds.cpu().numpy())
```

```
accuracy = accuracy_score(true_labels, all_preds)
```

```
precision = precision_score(true_labels, all_preds, average='weighted')
```

```
recall = recall_score(true_labels, all_preds, average='weighted')
```

```
f1 = f1_score(true_labels, all_preds, average='weighted')
```

```
conf_matrix = confusion_matrix(true_labels, all_preds)
```

```
auc_roc = roc_auc_score(true_labels, all_preds, multi_class="ovr", average="weighted")
```

```
matthews = matthews_corrcoef(true_labels, all_preds)
```

```
cohen_kappa = cohen_kappa_score(true_labels, all
```

```
## ASSISTANT
```

```
_preds)
```

```
print(f"Accuracy: {accuracy}")
```

```
print(f"Precision: {precision}")
```

```
print(f"Recall: {recall}")
```

```
print(f"F1 Score: {f1}")
```

```
print(f"Confusion Matrix: {conf_matrix}")
```

```
print(f"AUC-ROC: {auc_roc}")
```

```
print(f"Matthews Correlation Coefficient: {matthews}")
```

```
print(f"Cohen's Kappa: {cohen_kappa}")
```

```
# Evaluate the model
```

```
evaluate_model(model, test_dataloader, test_labels.numpy())
```

```
...
```

This extended code will print out various metrics to assess your model's performance. Make sure to replace `test_embeddings` and `test_labels` with your actual test data and labels.

```
## USER
```

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
-----
```

```
IndexError
```

```
Traceback (most recent call last)
```

```
<ipython-input-12-e94d166fa2c5> in <cell line: 53>()
```

57

58 # Forward pass

---> 59 outputs = model(embeddings_batch)

60

61 # Replace with your real labels

4 frames

/usr/local/lib/python3.10/dist-packages/transformers/modeling_utils.py

in

warn_if_padding_and_no_attention_mask(self, input_ids, attention_mask)

3819

3820 # Check only the first and last input IDs to reduce overhead.

-> 3821 if self.config.pad_token_id in input_ids[:, [-1, 0]]:

3822 warn_string = (

3823 "We strongly recommend passing in an `attention_mask` since your input_ids may
be padded. See "

IndexError: too many indices for tensor of dimension 1

ASSISTANT

The warning about uninitialized weights is expected if you're loading the model architecture but haven't trained it with your data yet. The real issue is the `IndexError`, which occurs when handling dimensions.

In the Transformers library, `BertForSequenceClassification` expects the input shape to be `(batch_size, sequence_length)`, and it looks like your input does not match this shape. Specifically, your embeddings should have a sequence length dimension.

Here's how to correct the error:

1. Reshape `embeddings_batch` to include a sequence length dimension.

Below is the modified portion of the code:

```
```python
... (Previous code)

Training loop
model.train()

for epoch in range(10):
 for batch in dataloader:
 embeddings_batch = batch[0]

 # Reshape to add sequence_length dimension
 embeddings_batch = embeddings_batch.unsqueeze(1)

 optimizer.zero_grad()

 # Forward pass
 try:
 outputs = model(input_ids=None, inputs_embeds=embeddings_batch)
 except Exception as e:
```

```
print(f"An error occurred: {e}")
```

```
continue
```

```
Assume labels are the same size as the batch
```

```
labels = torch.randint(0, 2, (batch_size,))
```

```
loss = criterion(outputs.logits, labels)
```

```
Backward pass and optimization
```

```
loss.backward()
```

```
optimizer.step()
```

```
print(f"Epoch {epoch+1} completed.")
```

```
... (Evaluation code)
```

```
...
```

I've added a ``try-except`` block around the forward pass to catch any exceptions and print an error message. This is useful for debugging. Also, I used ``inputs_embeds`` instead of ``input_ids`` because you're providing embeddings directly.

Make sure that this modification aligns with the dimensionality expected by your model and how you've prepared your embeddings.

```
USER
```

Now combine with

```
from transformers import BertForSequenceClassification

import torch

from torch.utils.data import DataLoader, TensorDataset

import torch.optim as optim

import torch.nn as nn

import xml.etree.ElementTree as ET

import json

import os
```

```
Function to load embeddings from JSON files
```

```
def load_embeddings_from_json(directories):

 all_embeddings = []

 for directory in directories:

 for filename in os.listdir(directory):

 if filename.endswith(".json"):

 filepath = os.path.join(directory, filename)

 with open(filepath, 'r') as f:

 embeddings = json.load(f)

 all_embeddings.extend(embeddings)

 return torch.FloatTensor(all_embeddings)
```

```
Simulated function to process feature extractor XML
```

```
def process_feature_extractor(xml_path):

 tree = ET.parse(xml_path)

 root = tree.getroot()

 # Replace this with your actual feature extraction logic based on XML content
```

```
return None
```

```
Load the pre-trained model
```

```
model_path = "/content/drive/MyDrive/Combined_Model.pt"
```

```
model = BertForSequenceClassification.from_pretrained('bert-base-uncased')
```

```
model.load_state_dict(torch.load(model_path))
```

```
Load embeddings
```

```
directories = ["/content/drive/MyDrive/embeddings", "/content/drive/MyDrive/embeddings2",
"/content/drive/MyDrive/embeddings3"]
```

```
all_embeddings = load_embeddings_from_json(directories)
```

```
Process feature extractor XML
```

```
feature_extractor_path = "/content/drive/MyDrive/xml/FeatureExtractor.xml"
```

```
process_feature_extractor(feature_extractor_path)
```

```
Create DataLoader
```

```
batch_size = 32
```

```
dataset = TensorDataset(all_embeddings)
```

```
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
Initialize optimizer and loss function
```

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
criterion = nn.CrossEntropyLoss()
```

```
Training loop
```

```
model.train()
```

```
for epoch in range(10):
```

```
 for batch in dataloader:
```

```
 embeddings_batch = batch[0]
```

```
 optimizer.zero_grad()
```

```
 # Forward pass
```

```
 outputs = model(embeddings_batch)
```

```
 # Replace with your real labels
```

```
 labels = torch.randint(0, 2, (batch_size,))
```

```
 loss = criterion(outputs.logits, labels)
```

```
 # Backward pass and optimization
```

```
 loss.backward()
```

```
 optimizer.step()
```

```
 print(f"Epoch {epoch+1} completed.")
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
from sklearn.metrics import confusion_matrix, roc_auc_score, matthews_corrcoef,
cohen_kappa_score
```

```
Assume test_embeddings and test_labels are your test data and labels
```

```
Replace these with your actual test data and labels
```

```
test_embeddings = torch.randn((300, 768))
```

```
test_labels = torch.randint(0, 2, (300,))
```

```
Create DataLoader for test data
```

```
test_dataset = TensorDataset(test_embeddings)
```

```
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
Evaluation function
```

```
def evaluate_model(model, dataloader, true_labels):
```

```
 model.eval()
```

```
 all_preds = []
```

```
 with torch.no_grad():
```

```
 for batch in dataloader:
```

```
 embeddings_batch = batch[0]
```

```
 outputs = model(embeddings_batch)
```

```
 _, preds = torch.max(outputs.logits, dim=1)
```

```
 all_preds.extend(preds.cpu().numpy())
```

```
accuracy = accuracy_score(true_labels, all_preds)
```

```
precision = precision_score(true_labels, all_preds, average='weighted')
```

```
recall = recall_score(true_labels, all_preds, average='weighted')
```

```
f1 = f1_score(true_labels, all_preds, average='weighted')
```

```
conf_matrix = confusion_matrix(true_labels, all_preds)
```

```
auc_roc = roc_auc_score(true_labels, all_preds, multi_class="ovr", average="weighted")
```

```
matthews = matthews_corrcoef(true_labels, all_preds)
```

```
cohen_kappa = cohen_kappa_score(true_labels, all_preds)
```

```
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1 Score: {f1}")
print(f"Confusion Matrix: {conf_matrix}")
print(f"AUC-ROC: {auc_roc}")
print(f"Matthews Correlation Coefficient: {matthews}")
print(f"Cohen's Kappa: {cohen_kappa}")
```

```
#Evaluate the model
```

```
evaluate_model(model, test_dataloader, test_labels.numpy())
```

```
ASSISTANT
```

Certainly, the combined code will incorporate model loading, feature extraction, training, and evaluation, all in one go. The code will train the BERT model on your embeddings and evaluate its performance using various metrics. Here it is:

```
```python  
from transformers import BertForSequenceClassification  
import torch  
from torch.utils.data import DataLoader, TensorDataset  
import torch.optim as optim
```

```

import torch.nn as nn

import xml.etree.ElementTree as ET

import json

import os

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

from sklearn.metrics import confusion_matrix, roc_auc_score, matthews_corrcoef,
cohen_kappa_score


# Function to load embeddings from JSON files

def load_embeddings_from_json(directories):

    all_embeddings = []

    for directory in directories:

        for filename in os.listdir(directory):

            if filename.endswith(".json"):

                filepath = os.path.join(directory, filename)

                with open(filepath, 'r') as f:

                    embeddings = json.load(f)

                    all_embeddings.extend(embeddings)

    return torch.FloatTensor(all_embeddings)


# Load embeddings

directories = ["/content/drive/MyDrive/embeddings", "/content/drive/MyDrive/embeddings2",
"/content/drive/MyDrive/embeddings3"]

all_embeddings = load_embeddings_from_json(directories)


# Load the pre-trained model

```



```
model_path = "/content/drive/MyDrive/Combined_Model.pt"

model = BertForSequenceClassification.from_pretrained('bert-base-uncased')

model.load_state_dict(torch.load(model_path))


# Create DataLoader

batch_size = 32

dataset = TensorDataset(all_embeddings)

dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)


# Initialize optimizer and loss function

optimizer = optim.Adam(model.parameters(), lr=0.001)

criterion = nn.CrossEntropyLoss()


# Training loop

model.train()

for epoch in range(10):

    for batch in dataloader:

        embeddings_batch = batch[0]

        optimizer.zero_grad()


        # Forward pass

        outputs = model(input_ids=None, inputs_embeds=embeddings_batch)


        # Replace with your real labels

        labels = torch.randint(0, 2, (batch_size,))

        loss = criterion(outputs.logits, labels)
```

```
# Backward pass and optimization

loss.backward()

optimizer.step()


print(f"Epoch {epoch+1} completed.")


# Assume test_embeddings and test_labels are your test data and labels

test_embeddings = torch.randn((300, 768))

test_labels = torch.randint(0, 2, (300,))


# Create DataLoader for test data

test_dataset = TensorDataset(test_embeddings)

test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)


# Evaluation function

def evaluate_model(model, dataloader, true_labels):

    model.eval()

    all_preds = []

    with torch.no_grad():

        for batch in dataloader:

            embeddings_batch = batch[0]

            outputs = model(input_ids=None, inputs_embeds=embeddings_batch)

            _, preds = torch.max(outputs.logits, dim=1)

            all_preds.extend(preds.cpu().numpy())
```

```

accuracy = accuracy_score(true_labels, all_preds)

precision = precision_score(true_labels, all_preds)

recall = recall_score(true_labels, all_preds)

f1 = f1_score(true_labels, all_preds)

conf_matrix = confusion_matrix(true_labels, all_preds)

auc_roc = roc_auc_score(true_labels, all_preds)

matthews = matthews_corrcoef(true_labels, all_preds)

cohen_kappa = cohen_kappa_score(true_labels, all_preds)


print(f"Accuracy: {accuracy}")

print(f"Precision: {precision}")

print(f"Recall: {recall}")

print(f"F1 Score: {f1}")

print(f"Confusion Matrix: {conf_matrix}")

print(f"AUC-ROC: {auc_roc}")

print(f"Matthews Correlation Coefficient: {matthews}")

print(f"Cohen's Kappa: {cohen_kappa}")


# Evaluate the model

evaluate_model(model, test_dataloader, test_labels.numpy())

...

```

Note: I removed the function `process_feature_extractor` because it wasn't used. You can add it back if you intend to use it later. Also, replace the `test_embeddings` and `test_labels` with your actual test data when you're ready to evaluate the model.

USER

The error is that you are trying to unpack 2 values from a list that only contains 0 values. This is likely because you are using the `torch.unbind()` function incorrectly. The `torch.unbind()` function takes a list of tensors as its first argument and returns a list of tensors, one for each element of the input list. In your case, you are trying to unpack a list of 0 tensors, which is why you are getting the error.

To fix this error, you can either change the input list to contain at least 2 tensors, or you can use the `torch.split()` function instead of the `torch.unbind()` function. The `torch.split()` function takes a tensor as its first argument and a number of splits as its second argument. It then returns a list of tensors, each of which is a split of the input tensor.

Rate this answer

thumb_up_off_alt

thumb_down_off_alt

Responses may display inaccurate or offensive information that doesn't represent Google's views.

[Learn more](#)

ASSISTANT

I apologize for the oversight. The error could be occurring because the batch size and the number of elements in the tensor may not be matching up, or possibly due to an incomplete batch. Without explicit use of ``torch.unbind()`` in the provided code, it's likely the issue might be in how the `DataLoader` is handling the batch of embeddings.

One way to resolve this issue would be to drop the last incomplete batch by setting ``drop_last=True`` in the `DataLoader`:

```
```python
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True, drop_last=True)
```
```

This ensures that the `DataLoader` only returns complete batches. It could help eliminate the error you're encountering. Give it a try and see if it resolves your issue.

USER

Some weights of `BertForSequenceClassification` were not initialized from the model checkpoint at `bert-base-uncased` and are newly initialized: `['classifier.bias', 'classifier.weight']`

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-14-eacb4547cbe3> in <cell line: 44>()
    48
    49     # Forward pass
--> 50     outputs = model(input_ids=embeddings, inputs_embeds=embeddings_batch)
```

51

52 # Replace with your real labels

3 frames

```
/usr/local/lib/python3.10/dist-packages/transformers/models/bert/modeling_bert.py in forward(self,
input_ids, attention_mask, token_type_ids, position_ids, head_mask, inputs_embeds,
encoder_hidden_states, encoder_attention_mask, past_key_values, use_cache, output_attentions,
output_hidden_states, return_dict)
```

965

966 if input_ids is not None and inputs_embeds is not None:

```
--> 967         raise ValueError("You cannot specify both input_ids and inputs_embeds at the same
time")
```

968 elif input_ids is not None:

969 self.warn_if_padding_and_no_attention_mask(input_ids, attention_mask)

ValueError: You cannot specify both input_ids and inputs_embeds at the same time

ASSISTANT

The error message indicates that you're providing both `input_ids` and `inputs_embeds` to the model, which is not allowed. You should provide either `input_ids` or `inputs_embeds`, but not both.

If you're using pre-computed embeddings, you should only use the `inputs_embeds` parameter. If you're using token IDs, you should only use the `input_ids` parameter.

Here's the part of your code that is causing the issue:

```
```python
outputs = model(input_ids=embeddings, inputs_embeds=embeddings_batch)
```
```

To resolve this issue, you can choose one of the following options:

1. If you're using embeddings, then:

```
```python
outputs = model(inputs_embeds=embeddings_batch)
```
```

2. If you're using input IDs, then:

```
```python
outputs = model(input_ids=embeddings)
```
```

Pick the one that aligns with your use case.

USER

Some weights of BertForSequenceClassification were not initialized from the model checkpoint at bert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']

You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

ValueError Traceback (most recent call last)

<ipython-input-15-b10f85d368c8> in <cell line: 44>()

```
48
49     # Forward pass
--> 50     outputs = model(inputs_embeds=embeddings_batch)
51
52     # Replace with your real labels
```

3 frames

```
/usr/local/lib/python3.10/dist-packages/transformers/models/bert/modeling_bert.py in forward(self,
input_ids,    attention_mask,    token_type_ids,    position_ids,    head_mask,    inputs_embeds,
encoder_hidden_states, encoder_attention_mask, past_key_values, use_cache, output_attentions,
output_hidden_states, return_dict)
```

```
974         raise ValueError("You have to specify either input_ids or inputs_embeds")
975
--> 976         batch_size, seq_length = input_shape
977         device = input_ids.device if input_ids is not None else inputs_embeds.device
978
```

ValueError: not enough values to unpack (expected 2, got 0)

ASSISTANT

The error suggests that the `input_shape` does not have enough dimensions to unpack into `batch_size` and `seq_length`. This is likely because `embeddings_batch` is not correctly shaped.

Here are some debugging steps:

1. **Check the shape of `embeddings_batch`:** Use `print(embeddings_batch.shape)` before the line where the error occurs to make sure it has the right shape. It should be a 2D tensor with dimensions `[batch_size, embedding_size]`.
2. **Check DataLoader Output:** Since you are using `DataLoader` to create mini-batches, verify that the `DataLoader` is actually providing the data in the shape you expect.

If you've verified that the shape is correct and still encounter the issue, you might need to reshape the tensor to explicitly include the sequence length:

```
```python
Reshape to include sequence length of 1
embeddings_batch = embeddings_batch.view(embeddings_batch.size(0), 1, -1)
```
```

After that, try running the forward pass again:

```
```python
outputs = model(inputs_embeds=embeddings_batch)
```

...

Make sure to adapt the reshaping according to the actual shape and dimensions that your data should have.

## USER

Here is an example of how to add a classification head to the BertForSequenceClassification model:

content\_copy

```
from transformers import BertForSequenceClassification, BertConfig
```

```
Create the configuration for the model.
```

```
config = BertConfig(
 num_labels=2, # The number of classes to classify.
)
```

```
Create the model.
```

```
model = BertForSequenceClassification(config)
```

```
Add a classification head to the model.
```

```
model.add_classification_head(num_labels=2)
```

Use code with caution

Once you have added a classification head to the model, you can then use the logits attribute to get

the logits for the model.

Here is an example of how to get the logits for the model:

```
content_copy
```

```
Get the logits for the model.
```

```
logits = model(input_ids=None, inputs_embeds=embeddings_batch)
```

```
ASSISTANT
```

Your example shows how to add a classification head to a `BertForSequenceClassification` model using the `BertConfig` and then getting logits from the model.

However, note that the `BertForSequenceClassification` model already comes with a classification head when you specify the number of labels in its configuration, so you typically wouldn't need to manually add another classification head using a method like `add_classification_head()`. This function doesn't actually exist in the Hugging Face Transformers library for the `BertForSequenceClassification` model, as the model is designed to already include a classification head.

Here's an example for clarification:

```
```python
```

```
from transformers import BertForSequenceClassification, BertConfig
```

```
# Create the configuration for the model.
```

```
config = BertConfig(
```

```

num_labels=2 # The number of output labels
)

# Create the model with the given configuration.
model = BertForSequenceClassification(config)

# The model already has a classification head, so you can directly get the logits.
# Assuming `embeddings_batch` is your input tensor
outputs = model(inputs_embeds=embeddings_batch)

# Get the logits from the model output.
logits = outputs.logits
...

```

In this example, `logits` will contain the output logits from the classification head, which you can then use for further calculations like loss computation or inference.

USER

Combine everything

ASSISTANT

Certainly, combining the various elements of code snippets and explanations you've given, you can create a complete pipeline as follows:

First, install the required packages if you haven't:

```
```bash
```

```
pip install torch transformers
```

```
...
```

Here's the Python code that puts everything together:

```
```python
```

```
from transformers import BertForSequenceClassification, BertConfig, BertTokenizer
```

```
import torch
```

```
# Define the feature extractor. Normally, you would load it from an XML file,
```

```
# but for the sake of this example, let's assume it's a function.
```

```
def feature_extractor(text):
```

```
    # Replace with your actual feature extraction logic.
```

```
    return text
```

```
# Create the Bert Configuration
```

```
config = BertConfig(
```

```
    num_labels=2 # The number of output labels, adjust as needed
```

```
)
```

```
# Initialize the model with the Bert configuration
```

```
model = BertForSequenceClassification(config)
```

```
# Load pre-trained weights (replace the path with your actual path)
```

```
model.load_state_dict(torch.load("/content/drive/MyDrive/000/pt/A.pt"))
```

```
# Initialize the Bert Tokenizer
```

```
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
# Let's assume we have a batch of raw text data that needs to be processed
```

```
raw_text_batch = ["Hello, world!", "Another example."] # Replace with your actual text data
```

```
# Feature extraction
```

```
processed_text_batch = [feature_extractor(text) for text in raw_text_batch]
```

```
# Tokenization
```

```
input_ids_batch = [tokenizer(text, padding=True, truncation=True, return_tensors="pt")['input_ids']  
for text in processed_text_batch]
```

```
# Convert list of tensors to a single tensor
```

```
input_ids_tensor = torch.cat(input_ids_batch, dim=0)
```

```
# Forward pass to get the logits
```

```
with torch.no_grad():
```

```
    outputs = model(input_ids=input_ids_tensor)
```

```
    logits = outputs.logits
```

```
# Here, 'logits' will contain the output logits from the model, which you can use for further  
calculations
```

```
print(logits)
```

```
...
```

This script:

1. Defines a mock feature extractor function. Replace it with your actual logic.
2. Creates a Bert model configuration.
3. Loads the model from a file.
4. Initializes the Bert tokenizer.
5. Processes a batch of raw text data.
6. Performs a forward pass to get logits.

Note that this is a simplified example. You'd typically add more elements like model evaluation, loss computation, data loaders, etc., depending on your specific needs.

```
## USER
```

```
import pandas as pd
```

```
from sklearn.manifold import TSNE
```

```
import matplotlib.pyplot as plt
```

```
import matplotlib
```

```
df = pd.read_csv('output/embedded_1k_reviews.csv') now combine that with my logic that loads all  
my embedding in the three different directories
```

```
matrix = df.ada_embedding.apply(eval).to_list()
```

```
# Create a t-SNE model and transform the data
```

```
tsne = TSNE(n_components=2, perplexity=15, random_state=42, init='random', learning_rate=200)
```

```
vis_dims = tsne.fit_transform(matrix)
```

```
colors = ["red", "darkorange", "gold", "turquoise", "darkgreen"]
```

```
x = [x for x,y in vis_dims]
```

```
y = [y for x,y in vis_dims]
```

```
color_indices = df.Score.values - 1
```

```
colormap = matplotlib.colors.ListedColormap(colors)
```

```
plt.scatter(x, y, c=color_indices, cmap=colormap, alpha=0.3)
```

```
plt.title("Amazon ratings visualized in language using t-SNE")
```

ASSISTANT

Certainly, you can combine the logic for loading embeddings from different directories and the t-SNE visualization code. Let's assume you have three directories: `dir1`, `dir2`, and `dir3`, containing CSV files `embedded_1k_reviews_1.csv`, `embedded_1k_reviews_2.csv`, and `embedded_1k_reviews_3.csv` respectively.

Here's how you can put it all together:

```
```python
```

```
import pandas as pd
```

```
from sklearn.manifold import TSNE
```

```
import matplotlib.pyplot as plt
```

```
import matplotlib
```

```
Function to load embeddings from a CSV file
```

```
def load_embeddings_from_csv(filename):
```

```
 df = pd.read_csv(filename)
```

```
 return df.ada_embedding.apply(eval).to_list()
```



```
Paths to your CSV files in the different directories
```

```
filenames = [
```

```
 'dir1/embedded_1k_reviews_1.csv',
```

```
 'dir2/embedded_1k_reviews_2.csv',
```

```
 'dir3/embedded_1k_reviews_3.csv'
```

```
]
```

```
Load all embeddings into a single list
```

```
all_embeddings = []
```

```
for filename in filenames:
```

```
 embeddings = load_embeddings_from_csv(filename)
```

```
 all_embeddings.extend(embeddings)
```

```
Create a t-SNE model and transform the data
```

```
tsne = TSNE(n_components=2, perplexity=15, random_state=42, init='random', learning_rate=200)
```

```
vis_dims = tsne.fit_transform(all_embeddings)
```