Text Processing Using Machine Learning

USING PRE-TRAINED MODELS

Dr. Fan Zhenzhen

zhenzhen@nus.edu.sg

SETUP

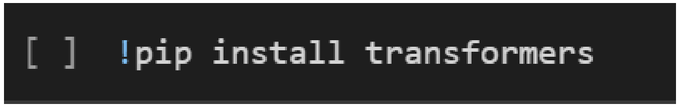
Why Transformers?

"*Transformers* (formerly known as *pytorch-transformers* and *pytorch-pretrained-bert*) provides state-of-the-art general-purpose architectures (BERT, GPT-2, RoBERTa, XLM, DistilBert, XLNet, T5, CTRL...) for Natural Language Understanding (NLU) and Natural Language Generation (NLG) with over thousands of pretrained models in 100+ languages and deep interoperability between PyTorch & TensorFlow 2.0."

- By HuggingFace

Installing transformers

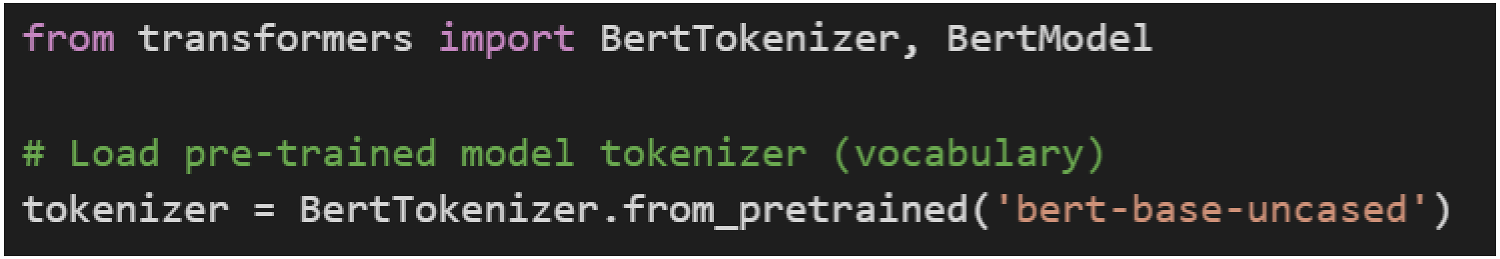
- On Google Colab (we follow this approach)
  - Pros: Simple



```
[ ]  !pip install transformers
```

  - Cons: Have to install every time you reconnect.
- On your local machine (**with GPU**):
  - Should install in a python virtual environment
  - Install TensorFlow 2.0 and PyTorch first
  - Follow the detailed instruction here: https://huggingface.co/transformers/installation.html

Import BERT



```
from transformers import BertTokenizer, BertModel

# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

- BERT-Base and BERT-Large
  - BERT Base: 12 encoder layers, 768 hidden units, 12 attention heads, 110M total parameters
  - BERT Large: 24 layers, 1024 hidden units,16 attention heads, 340M total parameters
- Uncased and Cased
  - Uncased: the text has been lowercased before WordPiece Tokenization; accent markers are removed.
  - Cased: case and accent markers are preserved.

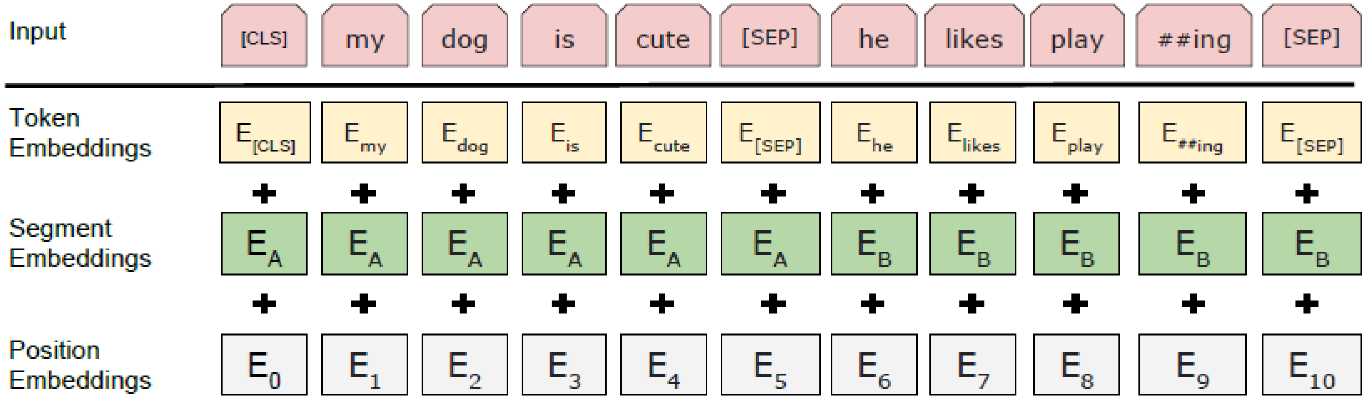
Using BERT

- Feature-based approach
  - Use BERT to extract features (word and sentence embeddings) of text input
  - Use the extracted vectors as input to downstream models
  - Or as contextualized representation to support downstream applications like search expansion, question answering, etc.
- Fine-tuning approach
  - Fine-tune BERT for a specific task with additional examples

BERT BASICS

Recap…

My dog is cute. He likes playing.



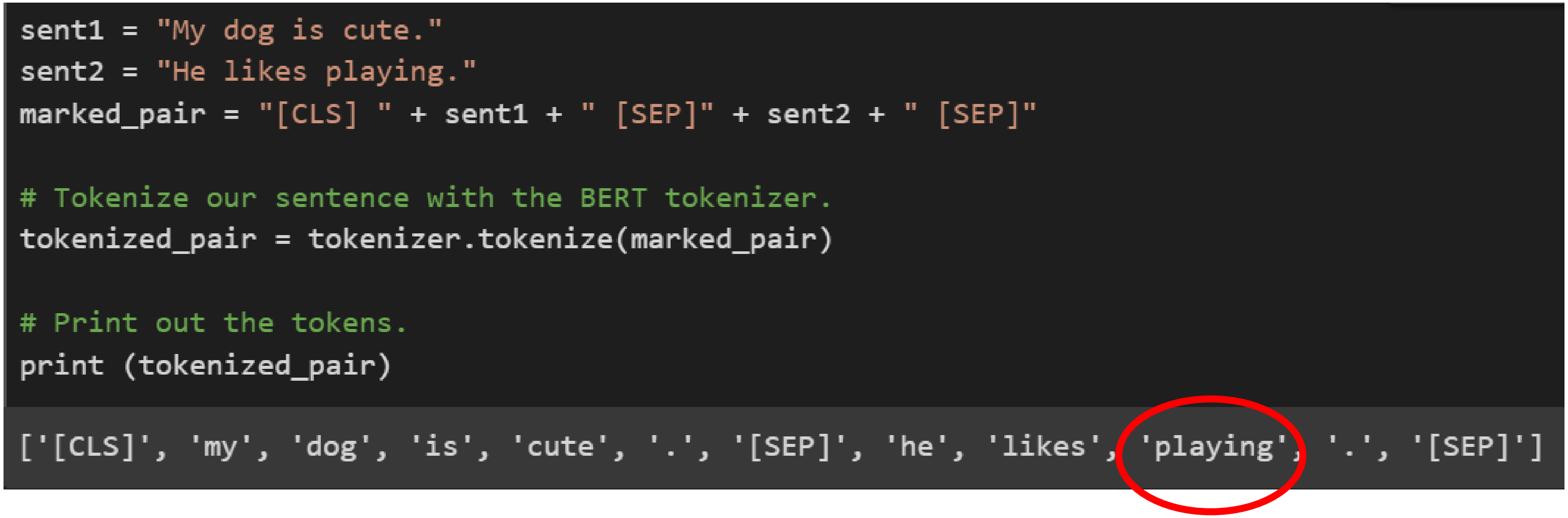
Input	[CLS]	my	dog	is	cute	[SEP]	he	likes	play	##ing	[SEP]
Token Embeddings	$E_{[CLS]}$	E_{my}	E_{dog}	E_{is}	E_{cute}	$E_{[SEP]}$	E_{he}	E_{likes}	E_{play}	$E_{\#\#ing}$	$E_{[SEP]}$
	+	+	+	+	+	+	+	+	+	+	+
Segment Embeddings	E_A	E_A	E_A	E_A	E_A	E_A	E_B	E_B	E_B	E_B	E_B
	+	+	+	+	+	+	+	+	+	+	+
Position Embeddings	E_0	E_1	E_2	E_3	E_4	E_5	E_6	E_7	E_8	E_9	E_{10}

Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

Devlin, Jacob, et al. "*Bert: Pre-training of deep bidirectional transformers for language understanding*."

BERT Input Requirements

- Text
  - One sentence or two sentences
  - Starting with a special token, [CLS]
  - [SEP] marking the end of a sentence
  - Single sentence: “[CLS] My dog is cute. [SEP]”
  - Sentence pair: “[CLS] My dog is cute. [SEP] He likes playing. [SEP]”
- Tokenize text



```
sent1 = "My dog is cute."
sent2 = "He likes playing."
marked_pair = "[CLS] " + sent1 + " [SEP]" + sent2 + " [SEP]"

# Tokenize our sentence with the BERT tokenizer.
tokenized_pair = tokenizer.tokenize(marked_pair)

# Print out the tokens.
print (tokenized_pair)
```

```
['[CLS]', 'my', 'dog', 'is', 'cute', '.', '[SEP]', 'he', 'likes', 'playing', '.', '[SEP]']
```

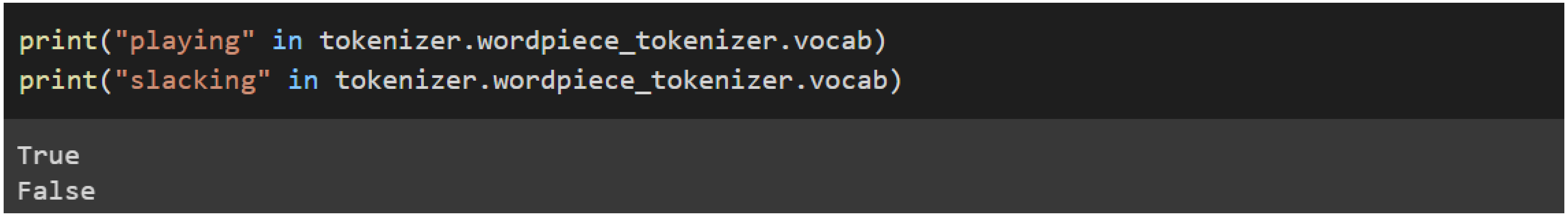
BERT Tokenizer

- The tokenizer processes the text in 3 steps

1. **Text normalization**: Convert all whitespace characters to spaces, and (for the Uncased model) lowercase the input and strip out accent markers. E.g., John Johanson's, → john johanson's,.
2. **Punctuation splitting**: Split all punctuation characters on both sides (i.e., add whitespace around all punctuation characters). E.g., john johanson's, → john johanson ' s ,
3. **WordPiece tokenization**: Apply whitespace tokenization to the output of the above procedure, and apply WordPiece tokenization to each token separately.

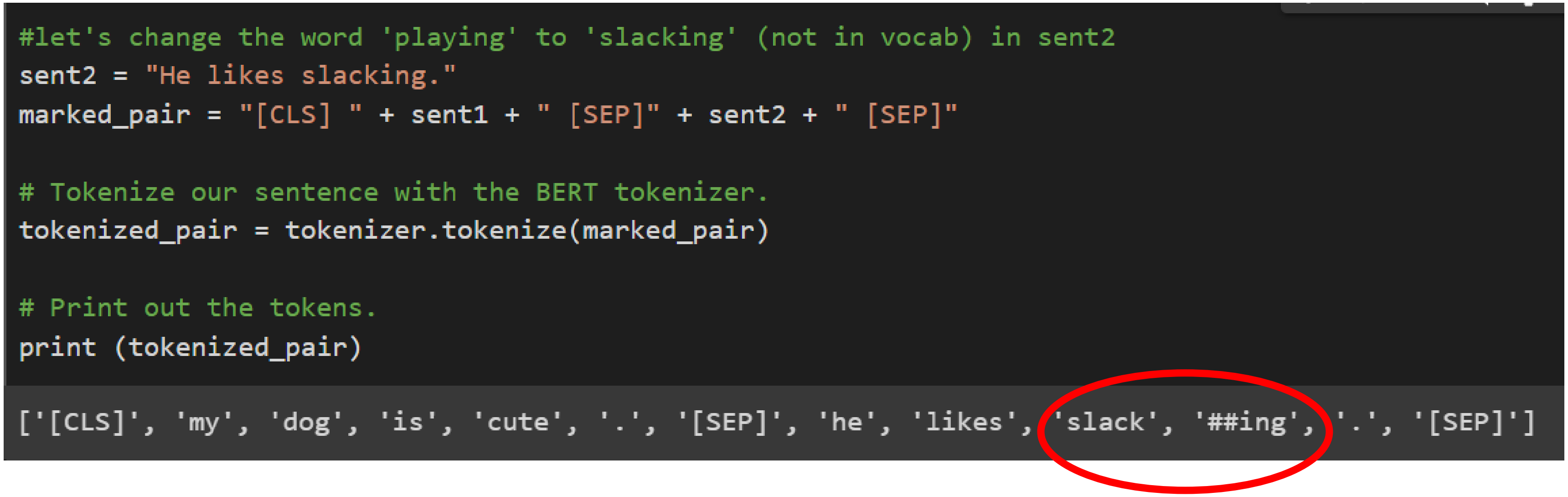
E.g., john johanson ' s , → john johan ##son ' s ,

Handling of out-of-vocabulary words



```
print("playing" in tokenizer.wordpiece_tokenizer.vocab)
print("slacking" in tokenizer.wordpiece_tokenizer.vocab)
```

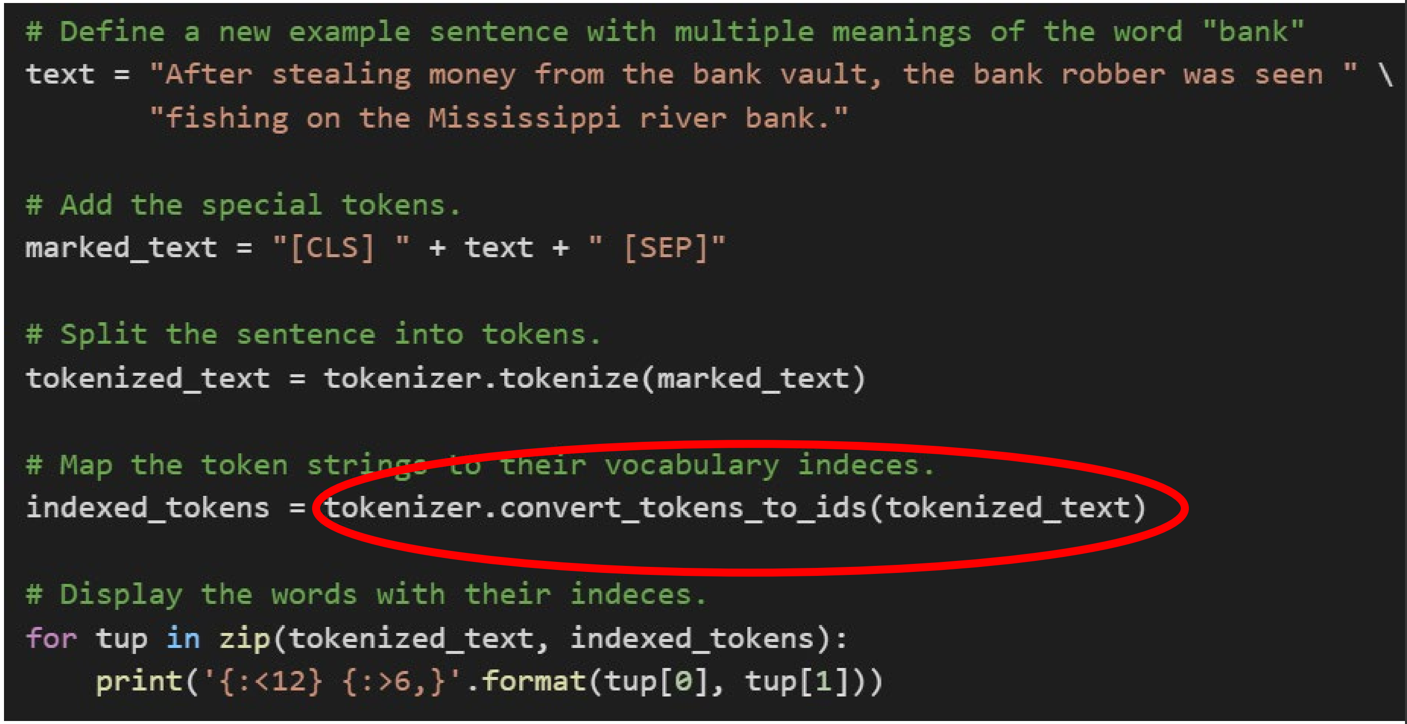
```
True
False
```



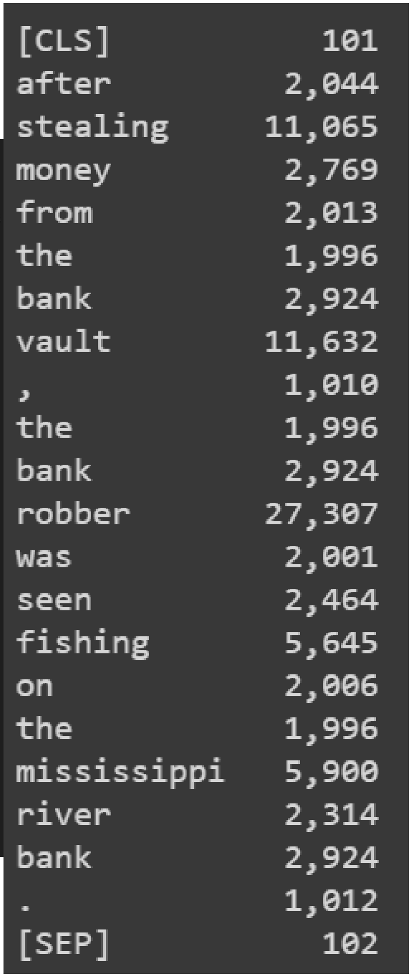
```
#let's change the word 'playing' to 'slacking' (not in vocab) in sent2
sent2 = "He likes slacking."
marked_pair = "[CLS] " + sent1 + " [SEP]" + sent2 + " [SEP]"

# Tokenize our sentence with the BERT tokenizer.
tokenized_pair = tokenizer.tokenize(marked_pair)

# Print out the tokens.
print (tokenized_pair)
```

```
['[CLS]', 'my', 'dog', 'is', 'cute', '.', '[SEP]', 'he', 'likes', 'slack', '##ing', '.', '[SEP]']
```

Map tokens to vocab indices



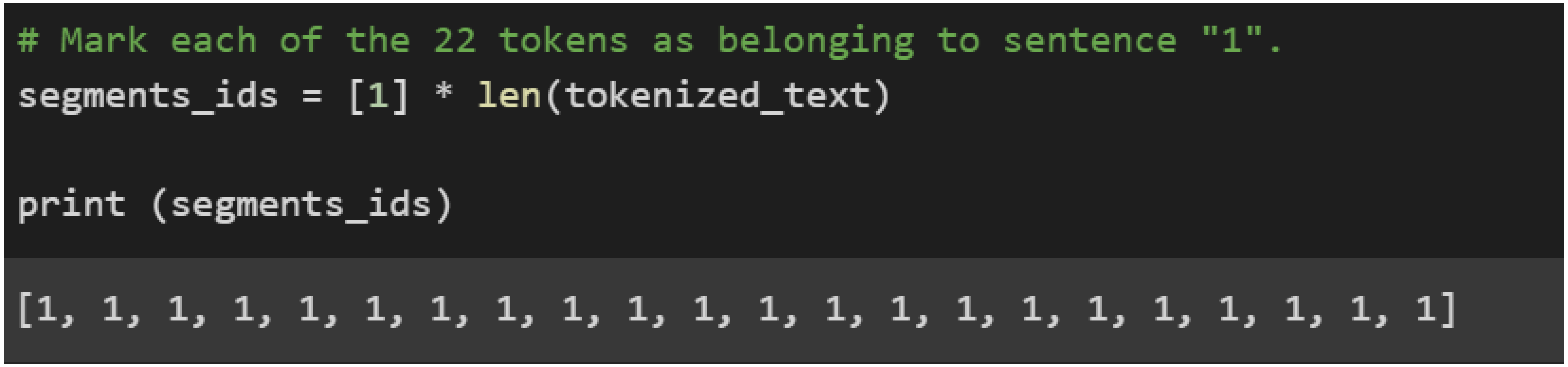
```
# Define a new example sentence with multiple meanings of the word "bank"
text = "After stealing money from the bank vault, the bank robber was seen " \
       "fishing on the Mississippi river bank."

# Add the special tokens.
marked_text = "[CLS] " + text + " [SEP]"

# Split the sentence into tokens.
tokenized_text = tokenizer.tokenize(marked_text)

# Map the token strings to their vocabulary indeces.
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)

# Display the words with their indeces.
for tup in zip(tokenized_text, indexed_tokens):
    print('{:<12} {:>6,}'.format(tup[0], tup[1]))
```



```
[CLS]           101
after         2,044
stealing     11,065
money         2,769
from          2,013
the           1,996
bank          2,924
vault        11,632
,             1,010
the           1,996
bank          2,924
robber       27,307
was           2,001
seen          2,464
fishing       5,645
on            2,006
the           1,996
mississippi   5,900
river         2,314
bank          2,924
.             1,012
[SEP]           102
```

Assign segment ID

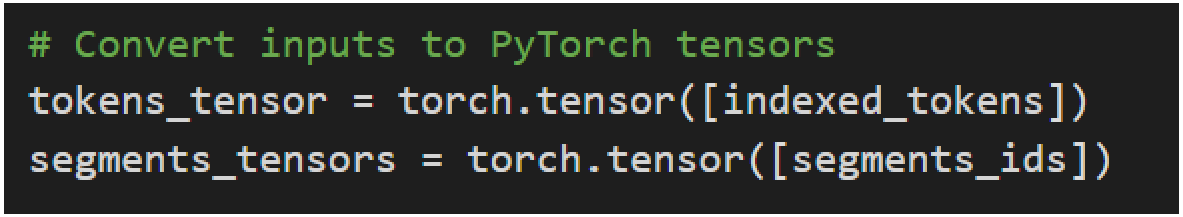
- To distinguish the two sentences in the pair
  - For each token, assign a value to indicate which sentence it belongs to: 0 (for sentence 0), 1 (for sentence 1)
- For single sentence:
  - Assign 1



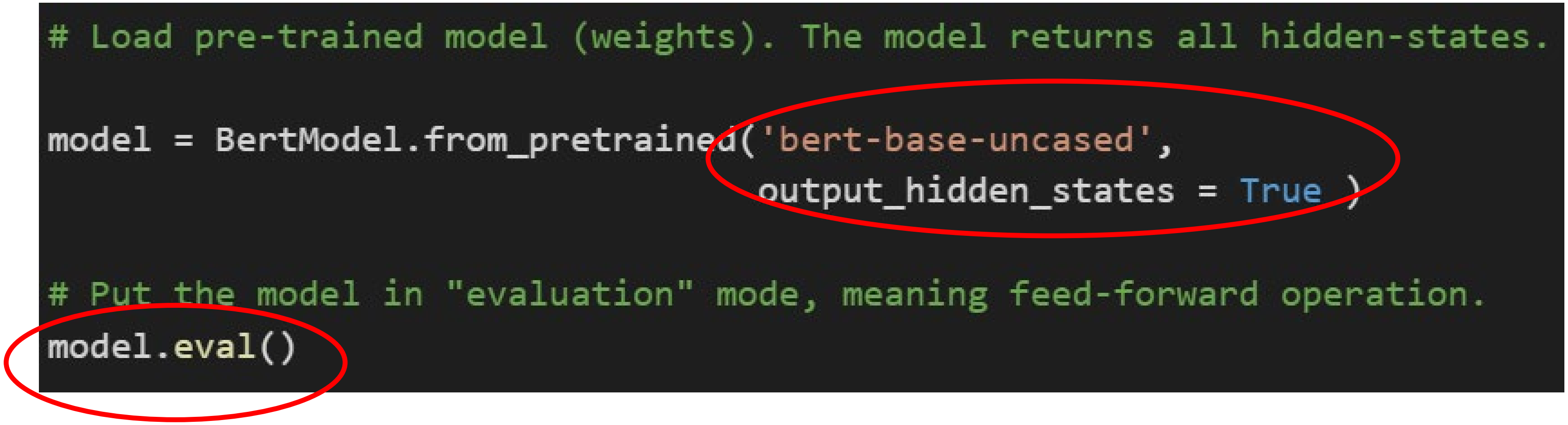
```
# Mark each of the 22 tokens as belonging to sentence "1".
segments_ids = [1] * len(tokenized_text)

print (segments_ids)
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

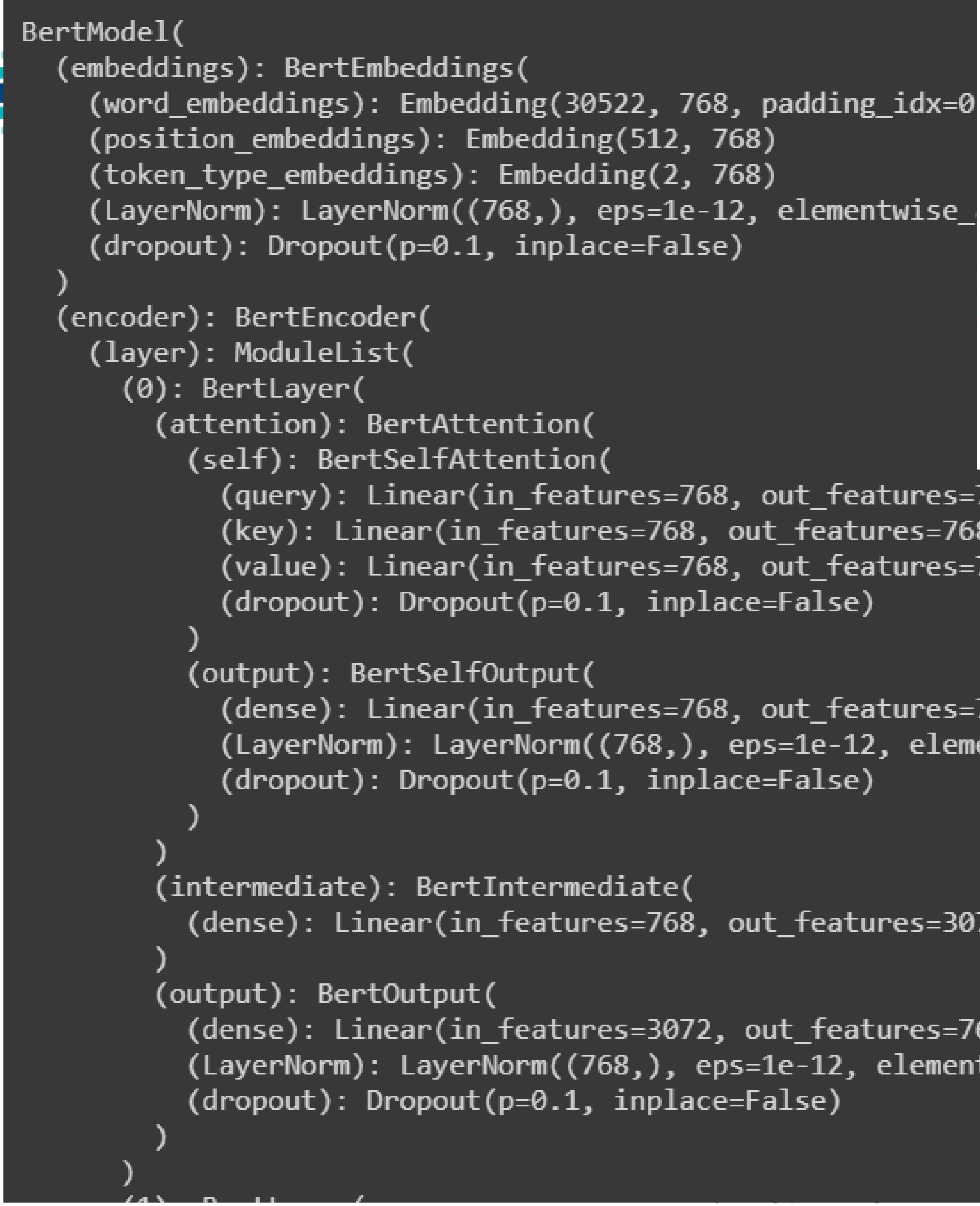
Get the inputs and the model ready



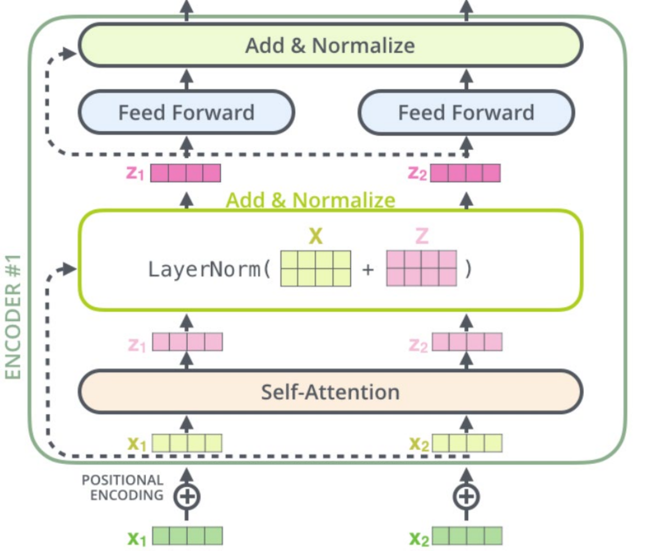
```
# Convert inputs to PyTorch tensors
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
```



```
# Load pre-trained model (weights). The model returns all hidden-states.

model = BertModel.from_pretrained('bert-base-uncased',
                                  output_hidden_states = True )


# Put the model in "evaluation" mode, meaning feed-forward operation.
model.eval()
```

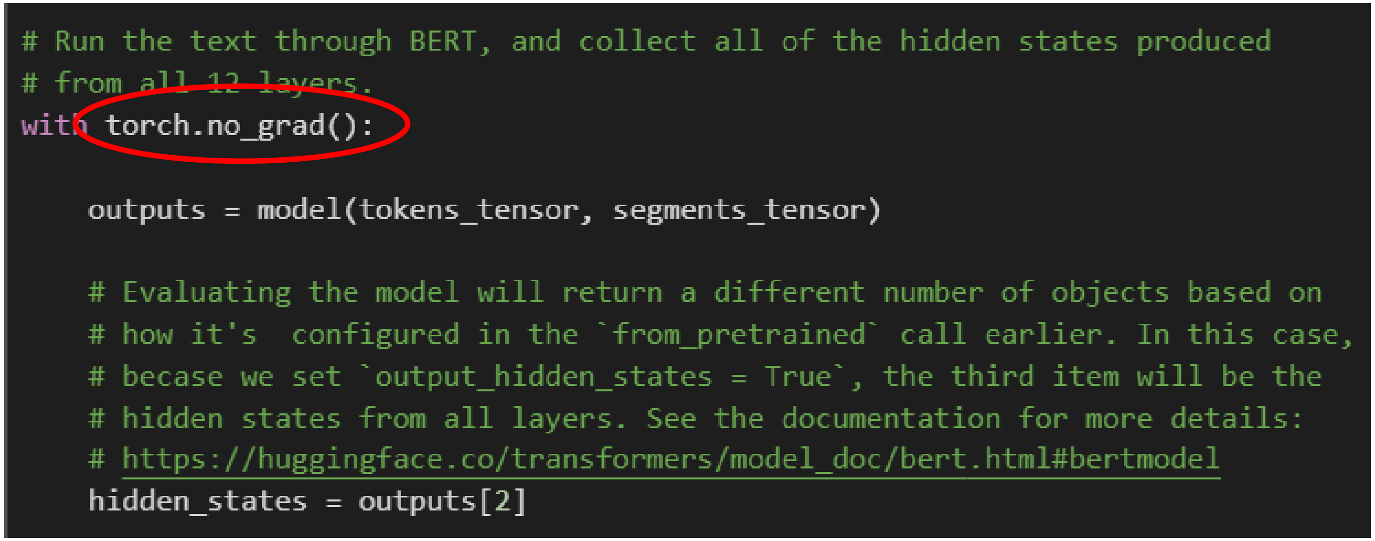


```
BertModel(
  (embeddings): BertEmbeddings(
    (word_embeddings): Embedding(30522, 768, padding_idx=0
    (position_embeddings): Embedding(512, 768)
    (token_type_embeddings): Embedding(2, 768)
    (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_
    (dropout): Dropout(p=0.1, inplace=False)
  )
  (encoder): BertEncoder(
    (layer): ModuleList(
      (0): BertLayer(
        (attention): BertAttention(
          (self): BertSelfAttention(
            (query): Linear(in_features=768, out_features=768, bias=True)
            (key): Linear(in_features=768, out_features=768, bias=True)
            (value): Linear(in_features=768, out_features=768, bias=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
          (output): BertSelfOutput(
            (dense): Linear(in_features=768, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
        (intermediate): BertIntermediate(
          (dense): Linear(in_features=768, out_features=3072, bias=True)
        )
        (output): BertOutput(
          (dense): Linear(in_features=3072, out_features=768, bias=True)
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (1): BertLayer(
```

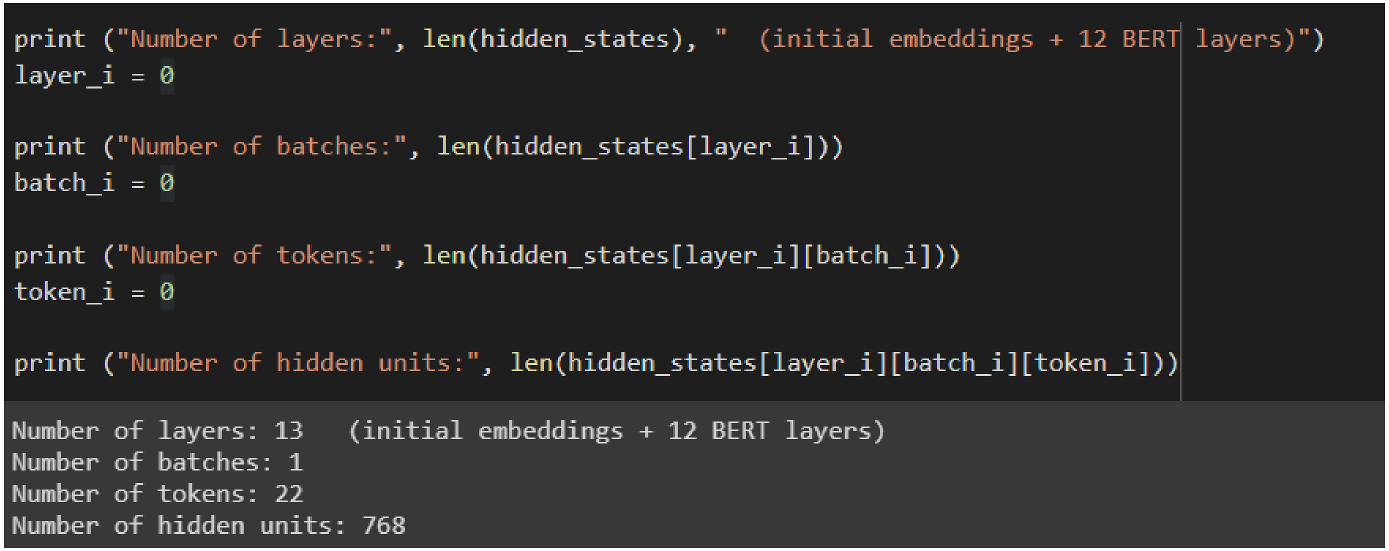


Collect the hidden states

- No gradient calculation is needed.



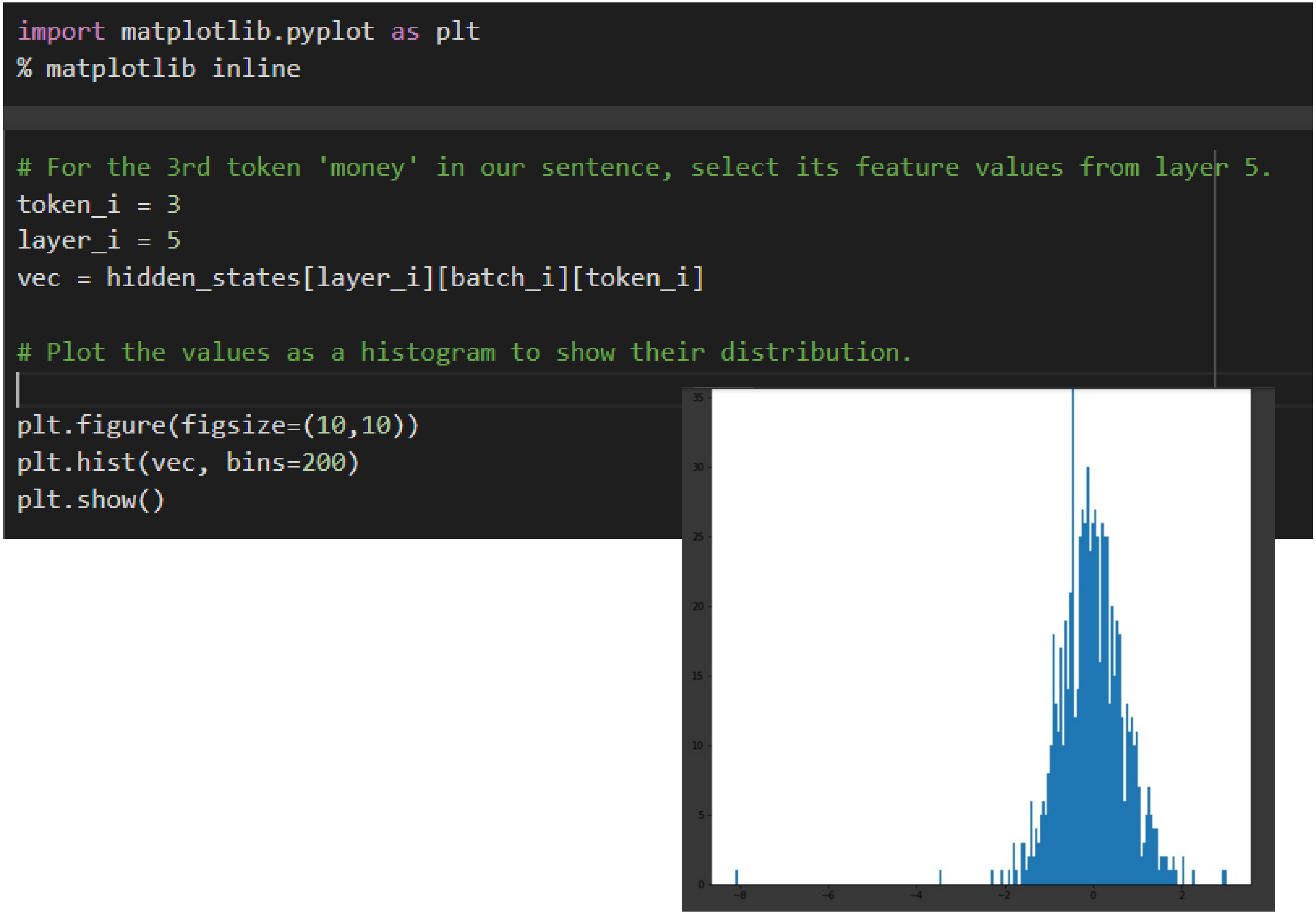
```
# Run the text through BERT, and collect all of the hidden states produced
# from all 12 layers.
with torch.no_grad():

    outputs = model(tokens_tensor, segments_tensor)

    # Evaluating the model will return a different number of objects based on
    # how it's  configured in the `from_pretrained` call earlier. In this case,
    # becase we set `output_hidden_states = True`, the third item will be the
    # hidden states from all layers. See the documentation for more details:
    # https://huggingface.co/transformers/model_doc/bert.html#bertmodel
    hidden_states = outputs[2]
```



```
print ("Number of layers:", len(hidden_states), "  (initial embeddings + 12 BERT layers)")
layer_i = 0

print ("Number of batches:", len(hidden_states[layer_i]))
batch_i = 0

print ("Number of tokens:", len(hidden_states[layer_i][batch_i]))
token_i = 0

print ("Number of hidden units:", len(hidden_states[layer_i][batch_i][token_i]))
```

```
Number of layers: 13   (initial embeddings + 12 BERT layers)
Number of batches: 1
Number of tokens: 22
Number of hidden units: 768
```

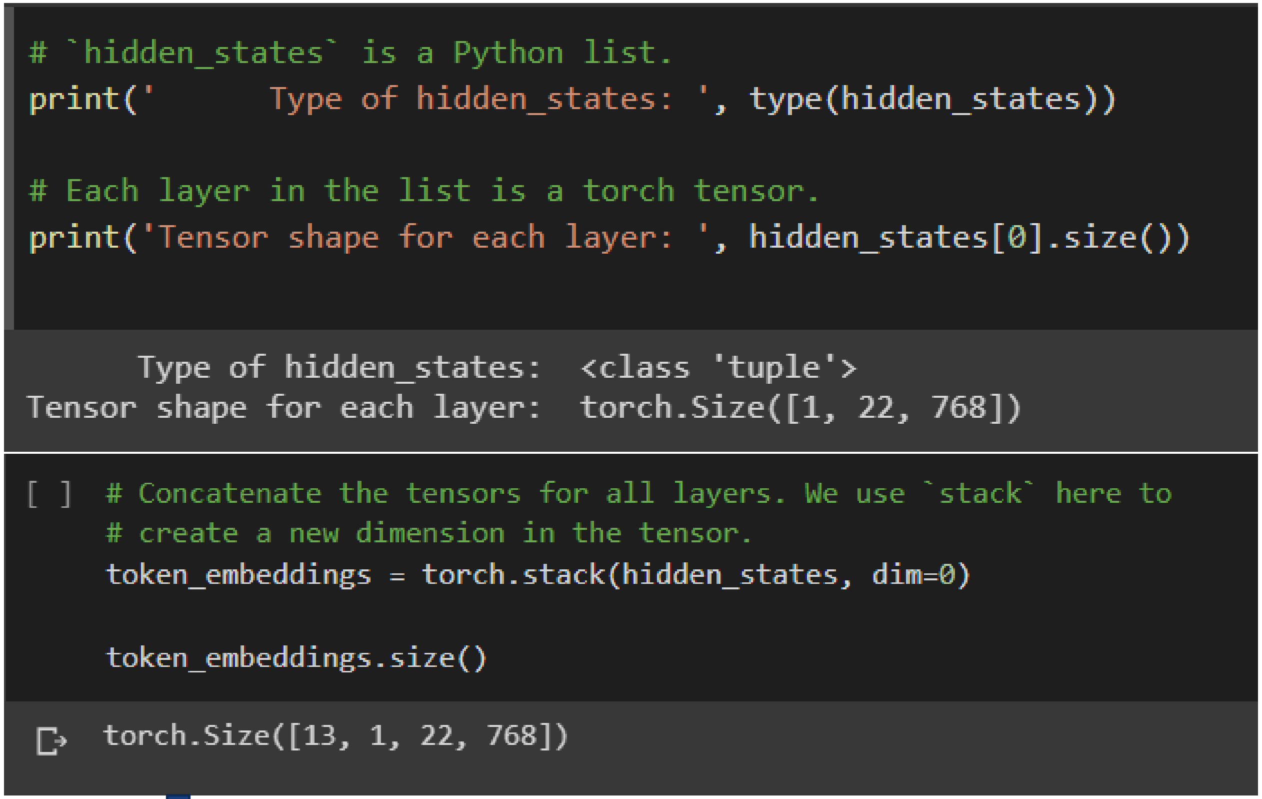
The vector for ‘money’, in layer 5



```
import matplotlib.pyplot as plt
% matplotlib inline
```

```
# For the 3rd token 'money' in our sentence, select its feature values from layer 5.
token_i = 3
layer_i = 5
vec = hidden_states[layer_i][batch_i][token_i]

# Plot the values as a histogram to show their distribution.

plt.figure(figsize=(10,10))
plt.hist(vec, bins=200)
plt.show()
```

Reshape



```
# `hidden_states` is a Python list.
print('      Type of hidden_states: ', type(hidden_states))

# Each layer in the list is a torch tensor.
print('Tensor shape for each layer: ', hidden_states[0].size())
```

```
      Type of hidden_states:  <class 'tuple'>
Tensor shape for each layer:  torch.Size([1, 22, 768])
```

```
[ ]  # Concatenate the tensors for all layers. We use `stack` here to
     # create a new dimension in the tensor.
     token_embeddings = torch.stack(hidden_states, dim=0)

     token_embeddings.size()
```

```
torch.Size([13, 1, 22, 768])
```

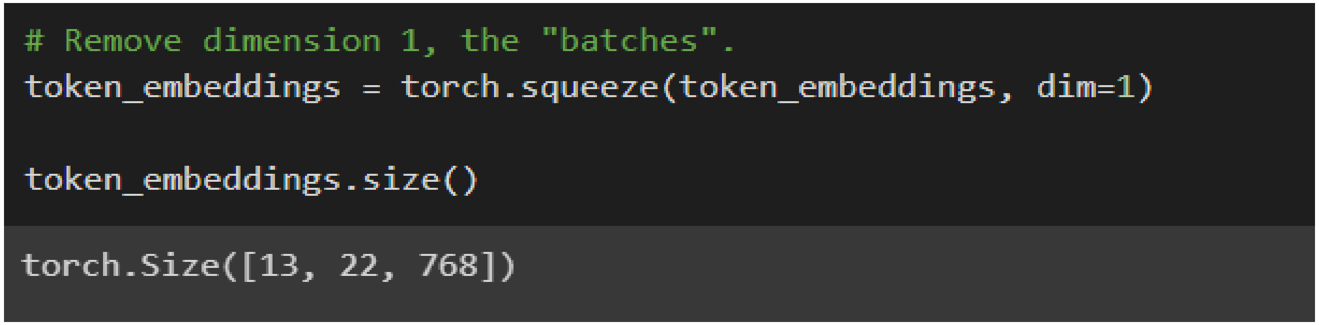
Current dimensions:

[# layers, # batches, # tokens, # features]

Desired dimensions:

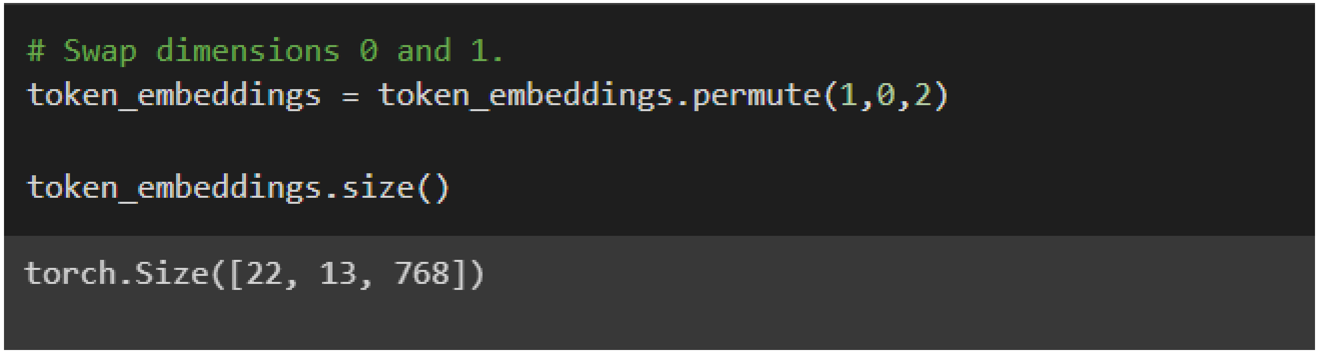
[# tokens, # layers, # features]

Reshape



```
# Remove dimension 1, the "batches".
token_embeddings = torch.squeeze(token_embeddings, dim=1)

token_embeddings.size()
```

```
torch.Size([13, 22, 768])
```



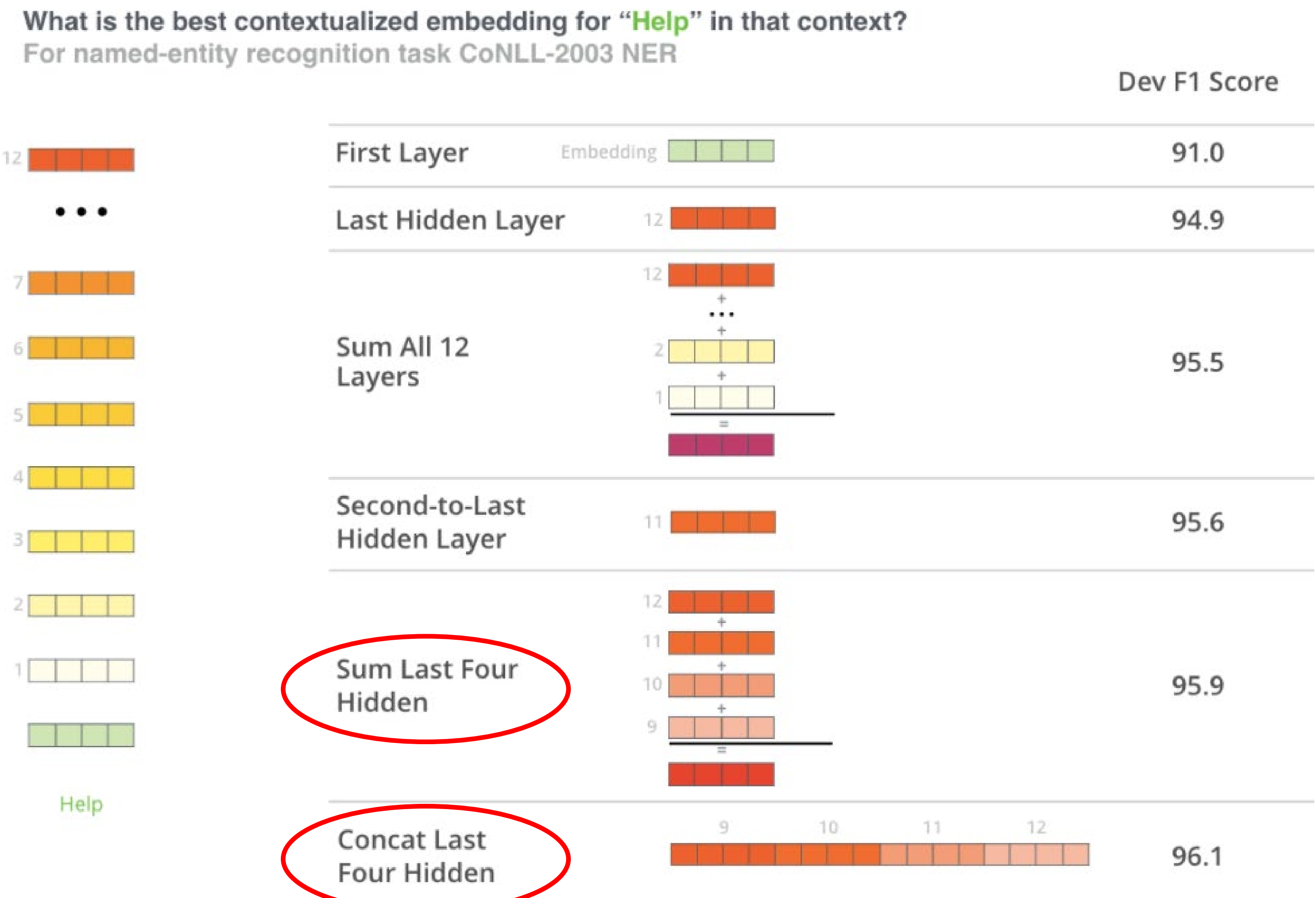
```
# Swap dimensions 0 and 1.
token_embeddings = token_embeddings.permute(1,0,2)

token_embeddings.size()
```

```
torch.Size([22, 13, 768])
```

Desired dimensions:

[# tokens, # layers, # features]

Recall: BERT for Feature Extraction



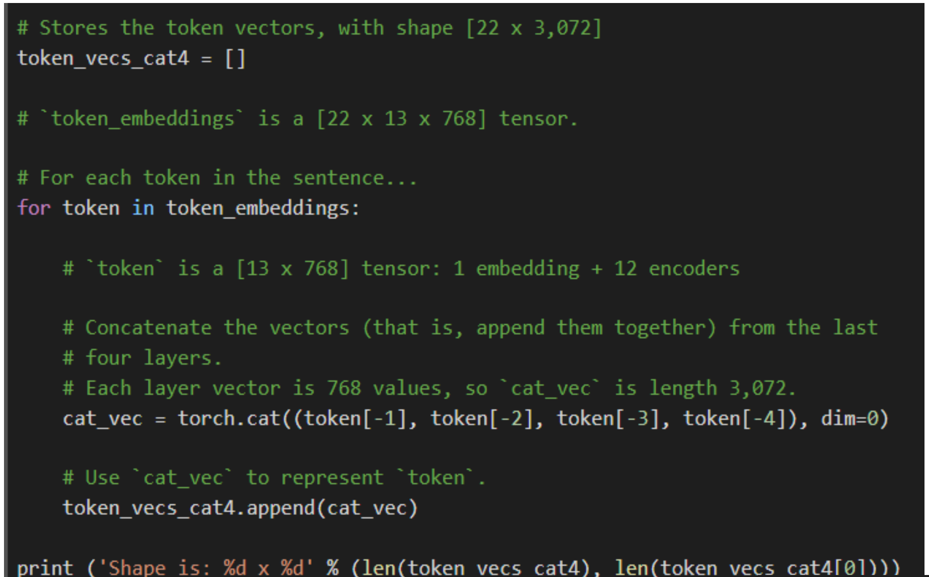
What is the best contextualized embedding for "Help" in that context?

For named-entity recognition task CoNLL-2003 NER

	Dev F1 Score
First Layer	91.0
Last Hidden Layer	94.9
Sum All 12 Layers	95.5
Second-to-Last Hidden Layer	95.6
Sum Last Four Hidden	95.9
Concat Last Four Hidden	96.1

Jay Alammar, *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)*

Try 2 ways

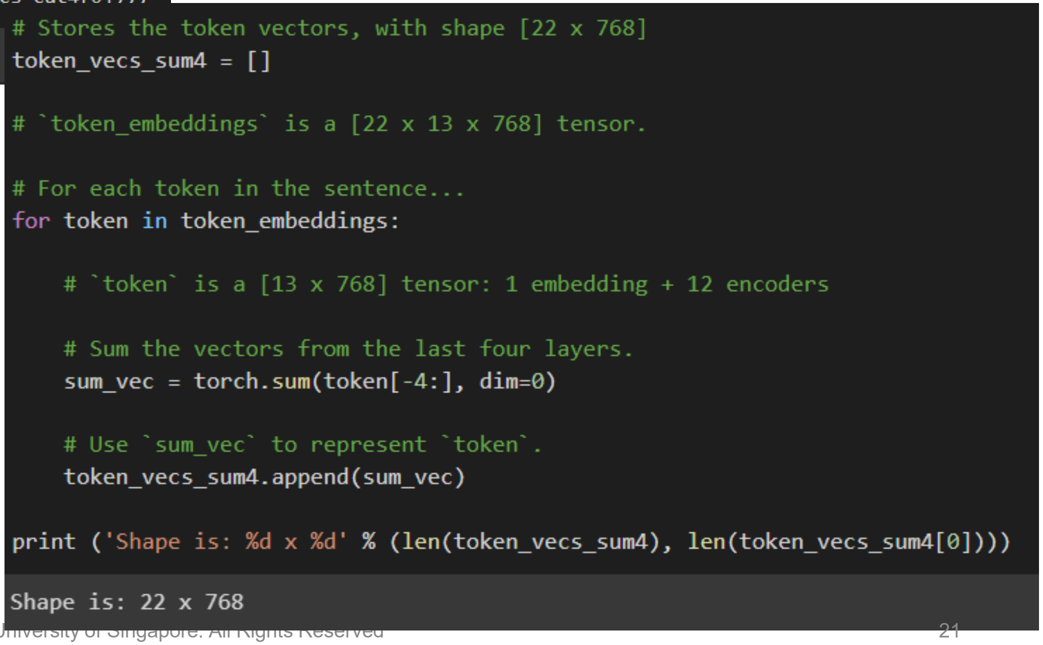


```
# Stores the token vectors, with shape [22 x 3,072]
token_vecs_cat4 = []

# `token_embeddings` is a [22 x 13 x 768] tensor.

# For each token in the sentence...
for token in token_embeddings:

    # `token` is a [13 x 768] tensor: 1 embedding + 12 encoders

    # Concatenate the vectors (that is, append them together) from the last
    # four layers.
    # Each layer vector is 768 values, so `cat_vec` is length 3,072.
    cat_vec = torch.cat((token[-1], token[-2], token[-3], token[-4]), dim=0)

    # Use `cat_vec` to represent `token`.
    token_vecs_cat4.append(cat_vec)

print ('Shape is: %d x %d' % (len(token_vecs_cat4), len(token_vecs_cat4[0])))
```

```
Shape is: 22 x 3072
```

Concatenate the last 4 hidden layers

Sum the last 4 hidden layers

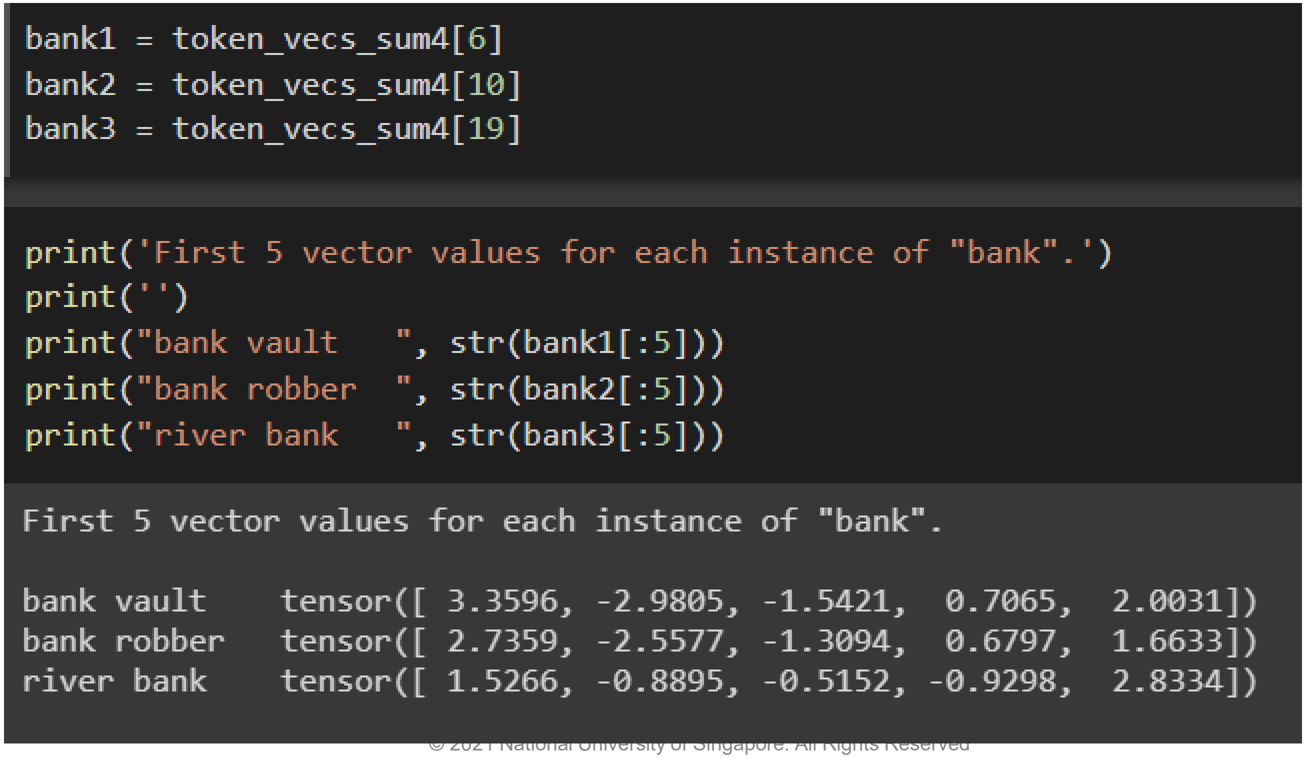


```
# Stores the token vectors, with shape [22 x 768]
token_vecs_sum4 = []

# `token_embeddings` is a [22 x 13 x 768] tensor.

# For each token in the sentence...
for token in token_embeddings:

    # `token` is a [13 x 768] tensor: 1 embedding + 12 encoders

    # Sum the vectors from the last four layers.
    sum_vec = torch.sum(token[-4:], dim=0)

    # Use `sum_vec` to represent `token`.
    token_vecs_sum4.append(sum_vec)

print ('Shape is: %d x %d' % (len(token_vecs_sum4), len(token_vecs_sum4[0])))
```

```
Shape is: 22 x 768
```

Contextual embeddings

- Recall our example sentence:
  - “[CLS] After stealing money from the **bank vault**, the **bank robber** was seen fishing on the Mississippi **river bank.**[SEP]”
  - 3 tokens of “bank” with index 6, 10, 19

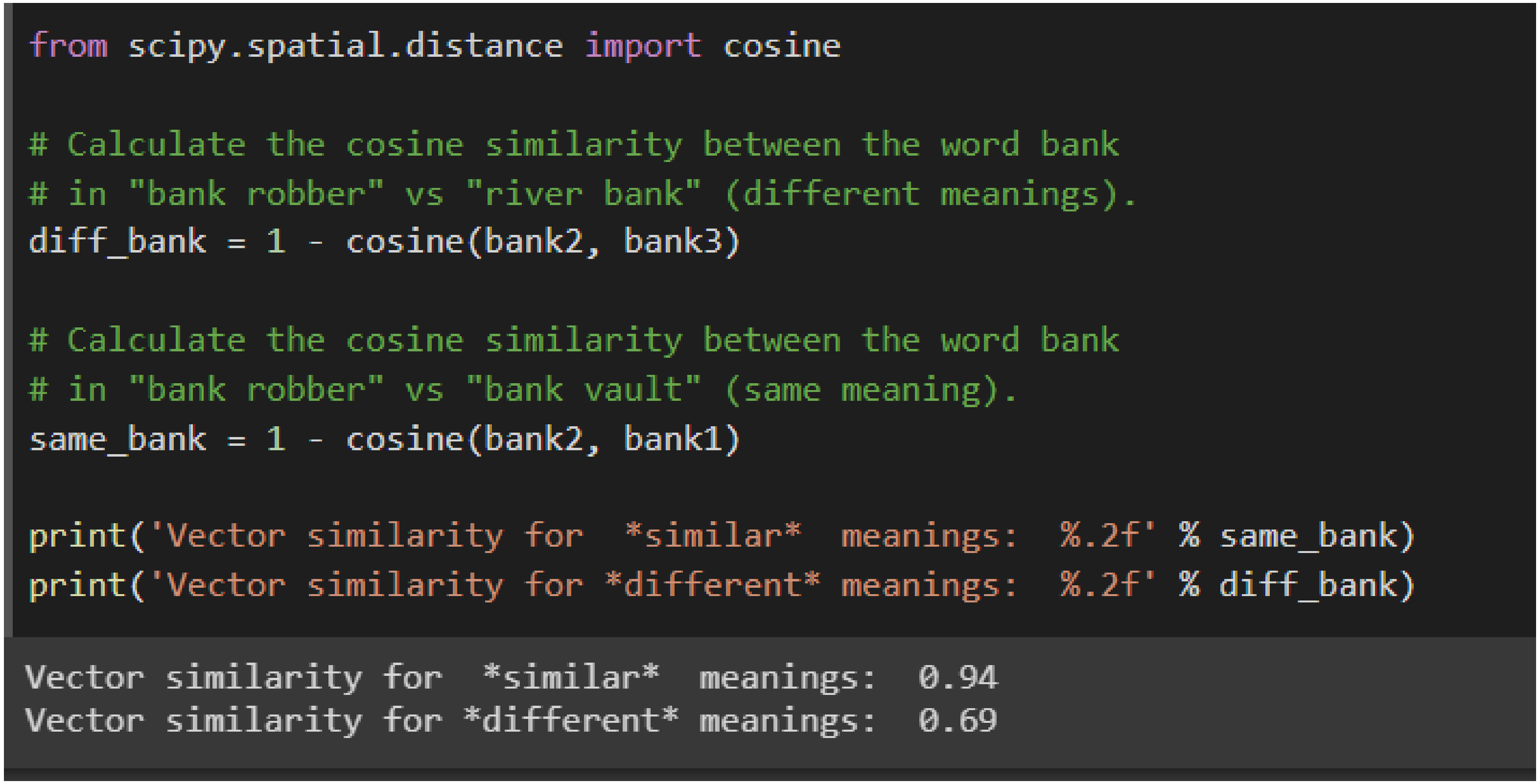


```
bank1 = token_vecs_sum4[6]
bank2 = token_vecs_sum4[10]
bank3 = token_vecs_sum4[19]
```

```
print('First 5 vector values for each instance of "bank".')
print('')
print("bank vault   ", str(bank1[:5]))
print("bank robber  ", str(bank2[:5]))
print("river bank   ", str(bank3[:5]))
```

```
First 5 vector values for each instance of "bank".

bank vault    tensor([ 3.3596, -2.9805, -1.5421,  0.7065,  2.0031])
bank robber   tensor([ 2.7359, -2.5577, -1.3094,  0.6797,  1.6633])
river bank    tensor([ 1.5266, -0.8895, -0.5152, -0.9298,  2.8334])
```

Check bank’s semantic similarity

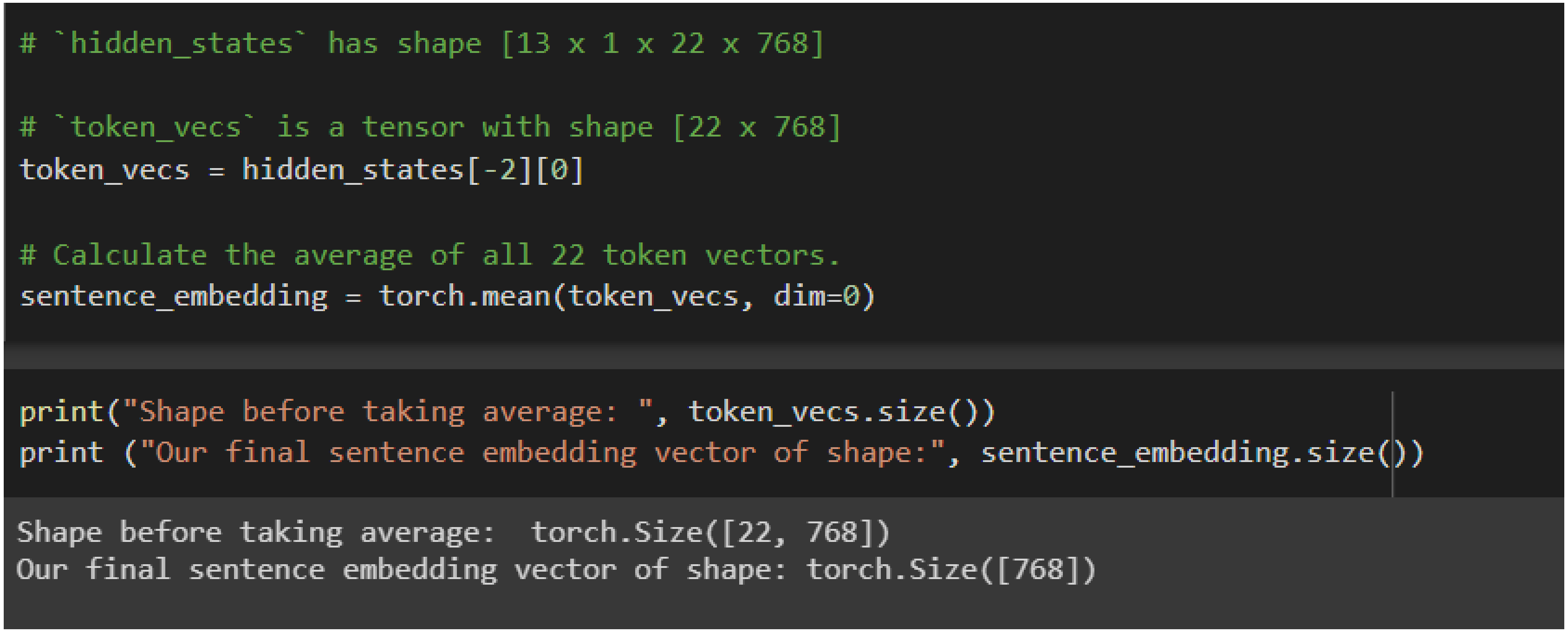


```
from scipy.spatial.distance import cosine

# Calculate the cosine similarity between the word bank
# in "bank robber" vs "river bank" (different meanings).
diff_bank = 1 - cosine(bank2, bank3)

# Calculate the cosine similarity between the word bank
# in "bank robber" vs "bank vault" (same meaning).
same_bank = 1 - cosine(bank2, bank1)

print('Vector similarity for  *similar*  meanings:  %.2f' % same_bank)
print('Vector similarity for *different* meanings:  %.2f' % diff_bank)
```

```
Vector similarity for  *similar*  meanings:  0.94
Vector similarity for *different* meanings:  0.69
```

Representing a sentence

- The vector for [CLS] can be used as a sequence approximate
- Alternatively, **average** the second to last hidden layer of each token producing a single 768 length vector



```
# `hidden_states` has shape [13 x 1 x 22 x 768]

# `token_vecs` is a tensor with shape [22 x 768]
token_vecs = hidden_states[-2][0]

# Calculate the average of all 22 token vectors.
sentence_embedding = torch.mean(token_vecs, dim=0)
```

```
print("Shape before taking average: ", token_vecs.size())
print ("Our final sentence embedding vector of shape:", sentence_embedding.size())
```

```
Shape before taking average:  torch.Size([22, 768])
Our final sentence embedding vector of shape: torch.Size([768])
```

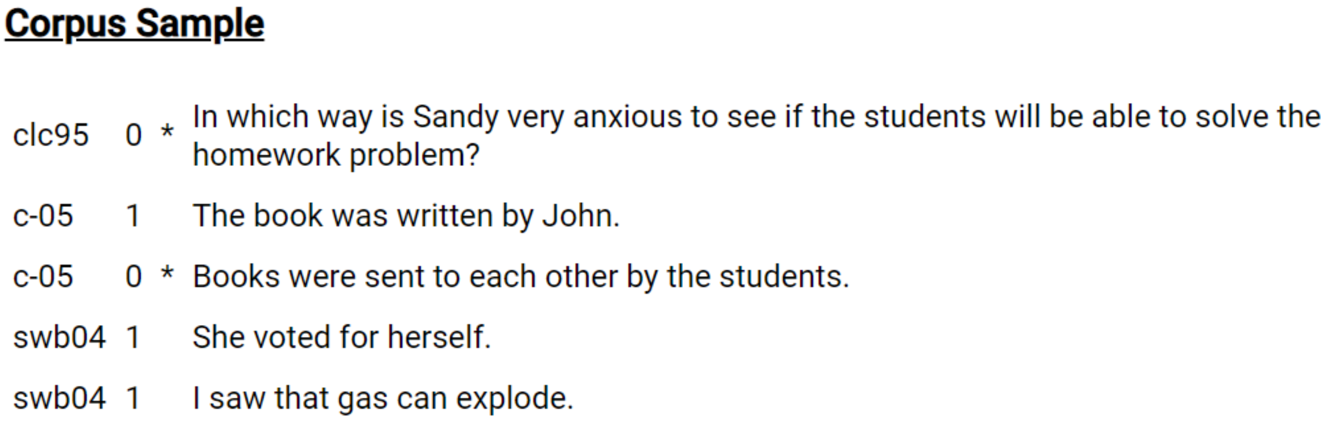
Exercise

- Create a Colab notebook to do the following task:
  - Given three sentences
    - “What’s the time now in Singapore?”
    - “What is the weather in Seattle today?”
    - “Apple is looking at buying the U.K. startup for $1 billion.”
  - Use BERT to extract sentence vector for each example above
    - Use the hidden states from second to last layer
    - Average each token in the sentence
  - And compute the cosine similarities between the 3 sentences.
- Enter your results into quiz “TPML Day 4 – BERT Exercise”.

FINE-TUNING BERT

The CoLA dataset

- The Corpus of Linguistic Acceptability (CoLA)
- 10657 sentences from 23 linguistics publications (9594 sentences publicly available as training and development sets)
- expertly annotated for acceptability (grammaticality) by their original authors (0=unacceptable, 1=acceptable).
- https://nyu-mll.github.io/CoLA/

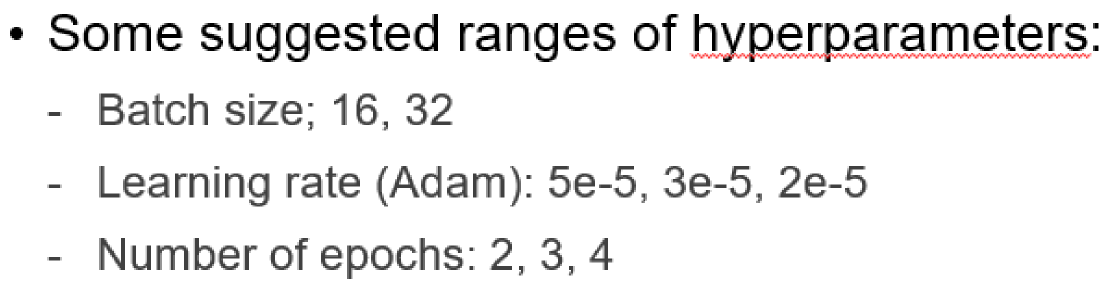


Corpus Sample

clc95	0	*	In which way is Sandy very anxious to see if the students will be able to solve the homework problem?
c-05	1		The book was written by John.
c-05	0	*	Books were sent to each other by the students.
swb04	1		She voted for herself.
swb04	1		I saw that gas can explode.

A typical PyTorch model training process

1. Prepare our data inputs and labels
2. Load data onto the GPU for acceleration
3. Define model and commit to GPU
4. Set hyperparameters (learning rate, optimizer, loss fuction, number of epochs…)



- Some suggested ranges of hyperparameters:
  - Batch size; 16, 32
  - Learning rate (Adam): 5e-5, 3e-5, 2e-5
  - Number of epochs: 2, 3, 4

A typical PyTorch model training process

5. Put the model in training mode
6. In each pass:
   a. Clear out the gradients calculated in the previous pass.
   b. Forward pass (feed input data through the network)
   c. Backward pass (backpropagation)
   d. Tell the network to update parameters with optimizer.step()
   e. Track variables for monitoring progress

Given *input_ids*, *attention_mask*, and *labels:*

```
model.zero_grad()
outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
loss = outputs.loss
loss.backward()
optimizer.step()
```

BERT Input Requirements

- So we’ve seen:
  - Token IDs
  - Segment IDs (also known as “token type id”)
- BERT also requires:
  - Mask IDs (also known as “attention mask”)
    - To distinguish **tokens** and **paddings** in a sequence
  - Positional Embeddings
    - Token positions in the sequence (automatically created as absolute positional embeddings in *Transformers*)

tokenizer.encode_plus

- The tokenizer.encode_plus function combines multiple steps for us:
  1. Split the sentence into tokens.
  2. Add the special [CLS] and [SEP] tokens.
  3. Map the tokens to their IDs.
  4. Pad or truncate all sentences to the same length.
  5. Create the attention masks which explicitly differentiate real tokens from [PAD] tokens.

```
print(tokenizer.encode_plus("This is my dog.", max_length = 15, padding = True))
```

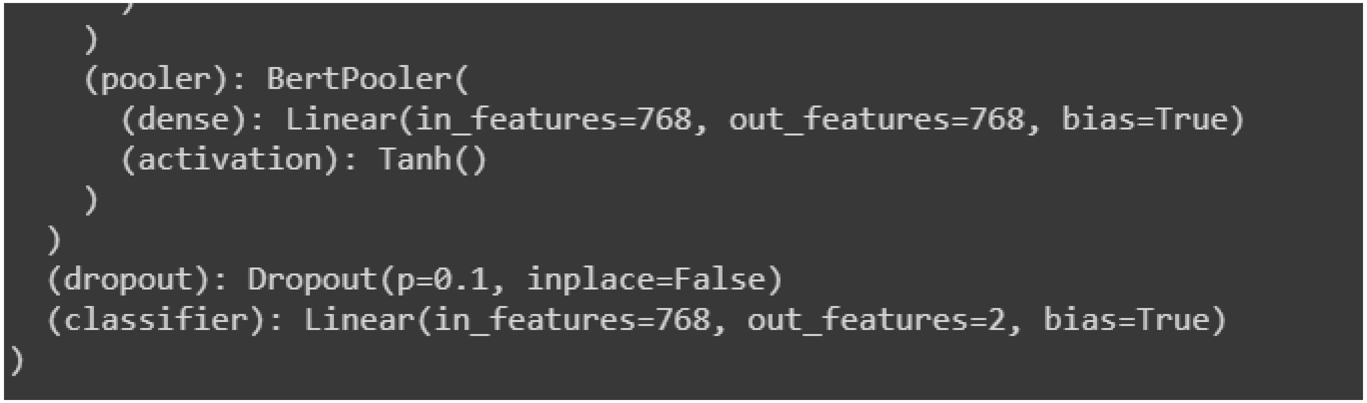
```
{'input_ids': [101, 2023, 2003, 2026, 3899, 1012, 102, 0, 0, 0, 0, 0, 0, 0, 0],
'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
'attention_mask': [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]}
```

Useful classes from Transformers

- BERT with a task-specific layer on top
- Different class for different task
  - BertModel
  - BertForPreTraining
  - BertForMaskedLM
  - BertForNextSentencePrediction
  - BertForSequenceClassification - The one we'll use.
  - BertForTokenClassification
  - BertForQuestionAnswering

BertForSequenceClassification

- BERT + one single linear layer for classification



```
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

- Other details, let’s see in the codes.

References

- Transformers documentations (https://huggingface.co/transformers/index.html)
- BERT tutorials, By Chris McCormick and Nick Ryan (http://mccormickml.com/)

Summarization Exercise

- 2 options:
  - Option 1: Build a model using fine-tuning to generate title based on report content
    - Using the dataset osha.txt oOccupational hazard and accident reports)
    - No header, 3 columns(“id”, “title”/”text”, “report”/”ctext”)
  - Option 2: Build a model using fine-tuning to generate summary for news
    - Using the dataset from Day 3 summarization workshop
    - Compare fine-tuned model’s performance with that of last week.
- Submission: submit a pdf file converted from your notebook (code and results): yournamme-summarization.pdf
  - with your name at the beginning of the document
  - displaying generated text
    - Option 1: generated titles versus actual titles for 50 reports
    - Option 2: generated summaries versus actual summaries for 50 news. Your discussion of comparison of performance of models.