Header

# 6. Natural Language Processing

In this tutorial, we'll take a detour away from stand-alone pieces of data such as still images, to data that is dependent on other data items in a sequence. For our example, we'll use text sentences. Language is naturally composed of sequence data, in the form of characters in words, and words in sentences. Other examples of sequence data include stock prices and weather data over time. Videos, while containing still images, are also sequences. Elements in the data have a relationship with what comes before and what comes after, and this fact requires a different approach.

## 6.1 Objectives

- Use a tokenizer to prepare text for a neural network
- See how embeddings are used to identify numerical features for text data

## 6.2 BERT

BERT, which stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers, was a ground-breaking model introduced in 2018 by Google.

BERT is simultaneously trained on two goals:

- Predict a missing word from a sequence of words
- Predict a new sentence after a sequence of sentences

Let's see BERT in action with these two types of challenges.

## 6.3 Tokenization

Since neural networks are number crunching machines, let's turn text into numerical tokens. Let's load BERT's tokenizer:

In [1]:
```
import torch
from transformers import BertTokenizer, BertModel, BertForMaskedLM, BertForQuestionAnswering
tokenizer = BertTokenizer.from_pretrained('bert-base-cased')

#BertTokenizer: It converts raw text (sentences or documents) into tokens
#BertModel: It provides pretrained transformer encodings of text but does not include a classification head
#BertForMaskedLM: Random words in a sentence are masked (replaced with [MASK]) and The model learns to predict the masked word
#BertForQuestionAnswering: extract a span of text from a context passage that answers a given question.
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/file_download.py:1132: FutureWarning: `resume_download` is deprecated a
nd will be removed in version 1.0.0. Downloads always resume when possible. If you want to force a new download, use `force_dow
nload=True`.
  warnings.warn(
HBox(children=(FloatProgress(value=0.0, description='tokenizer_config.json', max=49.0, style=ProgressStyle(des…
HBox(children=(FloatProgress(value=0.0, description='vocab.txt', max=213450.0, style=ProgressStyle(description…
HBox(children=(FloatProgress(value=0.0, description='tokenizer.json', max=435797.0, style=ProgressStyle(descri…
HBox(children=(FloatProgress(value=0.0, description='config.json', max=570.0, style=ProgressStyle(description_…
```

The BERT `tokenizer` can encode multiple texts at once. We will later test BERT's memory, so let's give it information and a question about that information. Feel free to come back here later and try a different combination of sentences.

In [2]:
```
text_1 = "I understand equations, both the simple and quadratical."
text_2 = "What kind of equations do I understand?"

# Tokenized input with special tokens around it (for BERT: [CLS] at the beginning and [SEP] at the end)
indexed_tokens = tokenizer.encode(text_1, text_2, add_special_tokens=True)
indexed_tokens
```

Out[2]:
```
[101,
 146,
 2437,
 11838,
 117,
 1241,
 1103,
 3014,
 1105,
 186,
 18413,
 21961,
 1348,
 119,
 102,
 1327,
 1912,
 1104,
 11838,
 1202,
 146,
 2437,
 136,
 102]
```

If we count the number of tokens, there are more tokens than words in our sentences. Let's see why that is. We can use convert_ids_to_tokens to see what was used as tokens.

In [3]:
```
tokenizer.convert_ids_to_tokens([str(token) for token in indexed_tokens])
```

Out[3]:
```
['[CLS]',
 'I',
 'understand',
 'equations',
 ',',
 'both',
 'the',
 'simple',
 'and',
 'q',
 '##uad',
 '##ratic',
 '##al',
 '.',
 '[SEP]',
 'What',
 'kind',
 'of',
 'equations',
 'do',
 'I',
 'understand',
 '?',
 '[SEP]']
```

There are two reasons why the indexed list is longer than our origincal input:

1. The `tokenizer` adds `special_tokens` to represent the start ( `[CLS]` ) of a sequence and separation ('[SEP]`) between sentences.
2. The `tokenizer` can break a word down into multiple parts.

From a linguistic perspective, the second one is interesting. Many languages have word roots, or components that make up a word. For instance, the word "quadratic" has the root "quadr" which means "4". Rather than use word roots as defined by a language, BERT uses a WordPiece model to find patterns in how to break up a word. The BERT model we will be using today has `28996` token vocabulary.

If we want to decode our encoded text directly, we can. Notice the `special_tokens` have been added in.

In [4]:
```
tokenizer.decode(indexed_tokens)
```

Out[4]: '[CLS] I understand equations, both the simple and quadratical. [SEP] What kind of equations do I understand? [SEP]'

## 6.4 Segmenting Text

In order to use the BERT model for predictions, it also needs a list of `segment_ids` . This is a vector the same length as our tokens and represents which segment belongs to each sentence.

Since our `tokenizer` added in some `special_tokens` , we can use these special tokens to find the segments. First, let's define which index correspnds to which special token.

In [5]:
```
cls_token = 101
sep_token = 102
```

Next, we can create a `for` loop. We'll start with our `segment_id` set to `0` , and we'll increment the `segment_id` whenever we see the [SEP] token. For good measure, we will return both the `segment_ids` and `indexd_tokens` as tensors as we will be feeding these into the model later.

In [6]:
```
def get_segment_ids(indexed_tokens):
    segment_ids = []
    segment_id = 0
    for token in indexed_tokens:
        if token == sep_token:
            segment_id += 1
        segment_ids.append(segment_id)
    segment_ids[-1] -= 1  # Last [SEP] is ignored
    return torch.tensor([segment_ids]), torch.tensor([indexed_tokens])
```

Let's test it out. Does each number correctly correspond to the first and second sentence?

In [7]:
```
segments_tensors, tokens_tensor = get_segment_ids(indexed_tokens)
segments_tensors
```

Out[7]: tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])

## 6.4 Text Masking

Let's start with the focus BERT has on words. To train for word embeddings, BERT masks out a word in a sequence of words. The mask is its own special token:

In [8]:
```
tokenizer.mask_token
```

Out[8]: '[MASK]'

In [9]:
```
tokenizer.mask_token_id
```

Out[9]: 103

Let's take our two sentences from before and mask out the position at index `5`. Feel free to return here to change the index to see how it changes the results!

In [10]:
```
masked_index = 5
```

Next, we'll apply the mask and verify it appears in our sequence of setences.

In [11]:
```
indexed_tokens[masked_index] = tokenizer.mask_token_id
tokens_tensor = torch.tensor([indexed_tokens])
tokenizer.decode(indexed_tokens)
```

Out[11]: '[CLS] I understand equations, [MASK] the simple and quadratical. [SEP] What kind of equations do I understand? [SEP]'

Then, we will load the model used to predict the missing word: `modelForMaskedLM`.

In [12]:
```
masked_lm_model = BertForMaskedLM.from_pretrained("bert-base-cased")
```

```
HBox(children=(FloatProgress(value=0.0, description='model.safetensors', max=435755784.0, style=ProgressStyle(…
```

```
Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight', 'cls.seq_relationship.bias', 'cls.seq_relationship.weight']
- This IS expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

Just like with other PyTorch modules, we can check the architecture.

In [13]:
```
masked_lm_model
```

Out[13]:
```
BertForMaskedLM(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(28996, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-11): 12 x 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)
            (intermediate_act_fn): GELUActivation()
          )
          (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)
          )
        )
      )
    )
  )
  (cls): BertOnlyMLMHead(
    (predictions): BertLMPredictionHead(
```

```
        (transform): BertPredictionHeadTransform(
          (dense): Linear(in_features=768, out_features=768, bias=True)
          (transform_act_fn): GELUActivation()
          (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
        )
        (decoder): Linear(in_features=768, out_features=28996, bias=True)
      )
    )
  )
```

Can you spot the section labeled `word_embeddings`? These are the embeddings BERT learned for each token.

In [14]:
```python
embedding_table = next(masked_lm_model.bert.embeddings.word_embeddings.parameters())
embedding_table
```

Out[14]:
```
Parameter containing:
tensor([[-0.0005, -0.0416,  0.0131,  ..., -0.0039, -0.0335,  0.0150],
        [ 0.0169, -0.0311,  0.0042,  ..., -0.0147, -0.0356, -0.0036],
        [-0.0006, -0.0267,  0.0080,  ..., -0.0100, -0.0331, -0.0165],
        ...,
        [-0.0064,  0.0166, -0.0204,  ..., -0.0418, -0.0492,  0.0042],
        [-0.0048, -0.0027, -0.0290,  ..., -0.0512,  0.0045, -0.0118],
        [ 0.0313, -0.0297, -0.0230,  ..., -0.0145, -0.0525,  0.0284]],
       requires_grad=True)
```

We can verify there is an embedding of size `768` for each of the `28996` tokens in BERT's vocabulary.

In [15]:
```python
embedding_table.shape
```

Out[15]:
```
torch.Size([28996, 768])
```

Let's test the model! Can it correctly predict the missing word in our provided sentences? We will use torch.no_grad to inform PyTorch not to calculate a gradient.

In [16]:
```python
with torch.no_grad():
    predictions = masked_lm_model(tokens_tensor, token_type_ids=segments_tensors)
predictions
```

Out[16]:
```
MaskedLMOutput(loss=None, logits=tensor([[[ -7.3832,  -7.2504,  -7.4539,  ...,  -6.0597,  -5.7928,  -6.2133],
         [ -6.7681,  -6.7896,  -6.8317,  ...,  -5.4655,  -5.4048,  -6.0683],
         [ -7.7323,  -7.9597,  -7.7348,  ...,  -5.7611,  -5.3566,  -4.3361],
         ...,
         [ -6.1213,  -6.3311,  -6.4144,  ...,  -5.8884,  -4.1157,  -3.1189],
         [-12.3216, -12.4479, -11.9787,  ..., -10.6539,  -8.7396, -11.0487],
         [-13.4115, -13.7876, -13.5183,  ..., -10.6359, -11.6582, -10.9009]]]), hidden_states=None, attentions=None)
```

This is a little bit hard to read, let's look at the `shape` to get a better sense of what's going on.

In [17]:
```
predictions[0].shape
```

Out[17]:
```
torch.Size([1, 24, 28996])
```

The `24` is our number of tokens, and the `28996` are the predictions for every token in BERT's vocabulary. We'd like to find the highest value accross all the token in the vocabulary, so we can use torch.argmax to find it.

In [18]:
```
# Get the predicted token
predicted_index = torch.argmax(predictions[0][0], dim=1)[masked_index].item()
predicted_index
```

Out[18]:
```
1241
```

Let's see what token `1241` corresponds to:

In [19]:
```
predicted_token = tokenizer.convert_ids_to_tokens([predicted_index])[0]
predicted_token
```

Out[19]:
```
'both'
```

What do you think? Is it correct?

In [20]:
```
tokenizer.decode(indexed_tokens)
```

Out[20]:
```
'[CLS] I understand equations, [MASK] the simple and quadratical. [SEP] What kind of equations do I understand? [SEP]'
```

# 6.5 Question and Answering

While word masking is interesting, BERT was designed for more complex problems such as sentence prediction. It is able to accomplish this by building on the Attention Transformer architecture.

We will be using a different version of BERT for this section, which has its own tokenizer. Let's find a new set of tokens for our sample sentences.

In [21]:
```
text_1 = "I understand equations, both the simple and quadratical."
text_2 = "What kind of equations do I understand?"

question_answering_tokenizer = BertTokenizer.from_pretrained('bert-large-uncased-whole-word-masking-finetuned-squad')
indexed_tokens = question_answering_tokenizer.encode(text_1, text_2, add_special_tokens=True)
segments_tensors, tokens_tensor = get_segment_ids(indexed_tokens)
```

```
HBox(children=(FloatProgress(value=0.0, description='tokenizer_config.json', max=48.0, style=ProgressStyle(des…
HBox(children=(FloatProgress(value=0.0, description='vocab.txt', max=231508.0, style=ProgressStyle(description…
HBox(children=(FloatProgress(value=0.0, description='tokenizer.json', max=466062.0, style=ProgressStyle(descri…
HBox(children=(FloatProgress(value=0.0, description='config.json', max=443.0, style=ProgressStyle(description_…
```

Next, let's load the `question_answering_model` .

In [22]:
```
question_answering_model = BertForQuestionAnswering.from_pretrained("bert-large-uncased-whole-word-masking-finetuned-squad")
```

```
HBox(children=(FloatProgress(value=0.0, description='model.safetensors', max=1340622760.0, style=ProgressStyle…
```

```
Some weights of the model checkpoint at bert-large-uncased-whole-word-masking-finetuned-squad were not used when initializing B
ertForQuestionAnswering: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight']
- This IS expected if you are initializing BertForQuestionAnswering from the checkpoint of a model trained on another task or w
ith another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertForQuestionAnswering from the checkpoint of a model that you expect to be ex
actly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).
```

We can feed in our tokens and segments, just like when we were masking out a word.

In [23]:
```
# Predict the start and end positions logits
with torch.no_grad():
    out = question_answering_model(tokens_tensor, token_type_ids=segments_tensors)
out
```

Out[23]:
```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[-5.5943, -4.2960, -5.2682, -1.2511, -6.8350, -0.3992,  2.2274,  2.4654,
         -6.6066,  2.5014, -4.4613, -4.8040, -7.8383, -5.5944, -4.7833, -6.9730,
         -7.1477, -5.2967, -7.4825, -6.7737, -6.8806, -8.6612, -5.5944]]), end_logits=tensor([[-0.7409, -5.3478, -4.2317, -0.0275, -2.6293, -5.9589, -2.8828,  2.7770,
         -4.8512, -2.2092, -2.2413,  4.4412, -0.7181, -0.7411, -3.8988, -5.3865,
         -5.0452, -4.4974, -6.3098, -5.5938, -5.5562, -5.3034, -0.7412]]), hidden_states=None, attentions=None)
```

The `question_answering_model` and answering model is scanning through our input sequence to find the subsequence that best answers the question. The higher the value, the more likely the start of the answer is.

In [24]:
```python
out.start_logits
```

Out[24]:
```
tensor([[-5.5943, -4.2960, -5.2682, -1.2511, -6.8350, -0.3992,  2.2274,  2.4654,
         -6.6066,  2.5014, -4.4613, -4.8040, -7.8383, -5.5944, -4.7833, -6.9730,
         -7.1477, -5.2967, -7.4825, -6.7737, -6.8806, -8.6612, -5.5944]])
```

Similarly, the higher the value in `end_logits`, the more likely the answer will end on that token.

In [25]:
```python
out.end_logits
```

Out[25]:
```
tensor([[-0.7409, -5.3478, -4.2317, -0.0275, -2.6293, -5.9589, -2.8828,  2.7770,
         -4.8512, -2.2092, -2.2413,  4.4412, -0.7181, -0.7411, -3.8988, -5.3865,
         -5.0452, -4.4974, -6.3098, -5.5938, -5.5562, -5.3034, -0.7412]])
```

We can then use torch.argmax to find the `answer_sequence` from start to finish:

In [26]:
```python
answer_sequence = indexed_tokens[torch.argmax(out.start_logits):torch.argmax(out.end_logits)+1]
answer_sequence
```

Out[26]:
```
[17718, 23671, 2389]
```

Finally, let's decode these tokens to see if the answer is correct!

In [27]:
```python
question_answering_tokenizer.convert_ids_to_tokens(answer_sequence)
```

Out[27]:
```
['quad', '##ratic', '##al']
```

In [28]:
```
question_answering_tokenizer.decode(answer_sequence)
```

Out[28]:
```
'quadratical'
```

## 6.7 Summary

Great work! You successfully used a Large Language Model (LLM) to extract answers from a sequence of sentences. Even though BERT was state-of-the-art when it was first released, many other LLMs have since broke ground. build.nvidia.com hosts many of these models to be interacted with in the browser. Go check it out and see where the state-of-the-art is today!

### 6.7.1 Clear the Memory

Before moving on, please execute the following cell to clear up the GPU memory.

In [29]:
```
import IPython
app = IPython.Application.instance()
app.kernel.do_shutdown(True)
```

Out[29]:
```
{'status': 'ok', 'restart': True}
```

### 6.7.2 Next

Congratulations, you have completed all the learning objectives of the course!

As a final exercise, and to earn certification in the course, successfully complete an end-to-end image classification problem in the assessment.

Header