Fine-Tuning BERT for Question Answering on the SQuAD v1.1 Dataset

Author: *Andrea Sottana*

Codebase link here

Main sources / Bibliography:

- https://colab.research.google.com/drive/1uSlWtJdZmLrI3FCNIlUHFxwAJiSu2J0-
- https://colab.research.google.com/drive/1aWeUxtbv2mpvHkW0_0No_x7yYFfIEIg9

IMPORTANT NOTE: This presentation **does not** contain any step-by-step explanation of the code linked in the slides, which is already well documented inside each module.

If you have questions about the code or its documentation, please ask during the session or get in touch with me later!

The code linked is fully modularized, and modules cannot be run directly. All the scripts to run the modules are here https://github.com/AI-Core/Practical-ML-DS/tree/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/scripts

The task and the dataset

- SQuAD (Stanford Question Answering Dataset) is a reading comprehension dataset, consisting of questions posed by crowdworkers on a set of Wikipedia articles
- We'll be using v1.1 which contains 100,000+ question-answer pairs on 500+ articles
- Each question is attached to a "context", i.e. a passage of text containing the answer, and the algorithm will learn to predict where the answer starts and ends inside the context
- The latest v2.0 also contains 50,000 unanswerable questions written adversarially by crowdworkers to look similar to answerable ones, to also train the algorithm to determine when no answer is supported by the paragraph and abstain from answering; we'll not go into this today

SQuAD structure

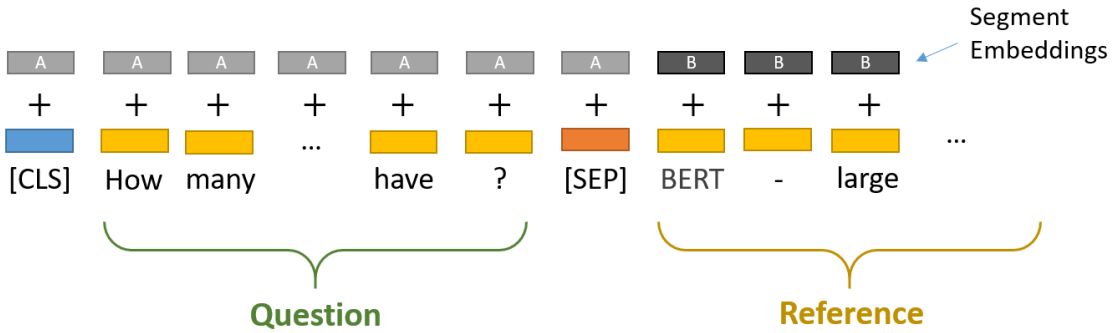
- The SQuAD dataset is a dictionary. Every “paragraph” has multiple questions. Each question has a labelled answer and the index in the context where the answer starts. This is to ensure, if the answer appear multiple times, that we know which one is the correct label.

Question:	The Basilica...is beside to which structure?
Answer:	the Main Building
Context:	Architecturally, the school has a Catholic character. Atop **the Main Building**'s gold dome is a golden statue of the Virgin Mary. Immediately in front of **the Main Building** and facing it, is a copper statue of Christ with arms upraised with the legend "Venite Ad Me Omnes". Next to **the Main Building** is the Basilica of the Sacred Heart.

- “*the Main Building*” appears 3 times, so the start index will help us label the answer correctly. In the case above, the third occurrence is the actual answer
- The test dataset has multiple acceptable answers for each question

Inputs and outputs of the model

- For each question + context pair, the input is a tokenized sequence starting with one “[CLS]” token, followed by the question tokens, followed by one “[SEP]” token, followed by the context tokens, followed by one “[SEP]” tokens, followed by as many “[PAD]” tokens are necessary to reach the maximum sequence length, if applicable.
- The model outputs two softmax probability distributions for each token being the start and the end token of the answer respectively.
- We then calculate the argmax to find the start and end predictions.
- There are slightly more complicated rules in the unusual circumstance where the most likely predicted end token precedes the start token, but we’ll not look at this case today.



Question: `How many parameters does BERT-large have?`

Reference Text: `BERT-large is really big... it has 24 layers and an embedding size of 1,024, for a total of 340M parameters! Altogether it is 1.34GB, so expect it to take a couple minutes to download to your Colab instance.`

In [1]:
```
from transformers import BertTokenizer
```

In [2]:
```
tokenizer = BertTokenizer.from_pretrained("bert-base-cased", do_lower_case=False)
```

In [3]:
```
question = "How many parameters does BERT-large have?"
print(question)
```

```
How many parameters does BERT-large have?
```

In [4]:
```
context = "BERT-large is really big... it has 24-layers and an embedding size of 1,024, for a total of 340M param
print(context)
```

```
BERT-large is really big... it has 24-layers and an embedding size of 1,024, for a total of 340M parameters! Al
together it is 1.34GB, so expect it to take a couple minutes to download to your Colab instance.
```

In [5]:
```
'''A dictionary containing the sequence pair and additional information. There are 3 keys, each value
is a torch.tensor of shape (1, max_len) and can be converted to just (max_len) by applying .squeeze():
- 'input_ids': the ids of each token of the encoded sequence pair, with padding at the end
- 'token_type_ids': 1 for token positions in  answer text, 0 elsewhere (i.e. in question and padding)
- 'attention_mask': 1 for non "[PAD]" token, 0 for "[PAD]" tokens.'''
encoded_dict = tokenizer.encode_plus(
    question,
    context,
    add_special_tokens=True,  # Add '[CLS]' and '[SEP]' tokens
    max_length=150,
    padding='max_length',  # Pad or truncates sentences to `max_length`
    truncation=True,
    return_attention_mask=True,  # Construct attention masks.
    return_tensors='pt',  # Return pytorch tensors.
)
```

In [6]:
```
encoded_dict.keys()
```

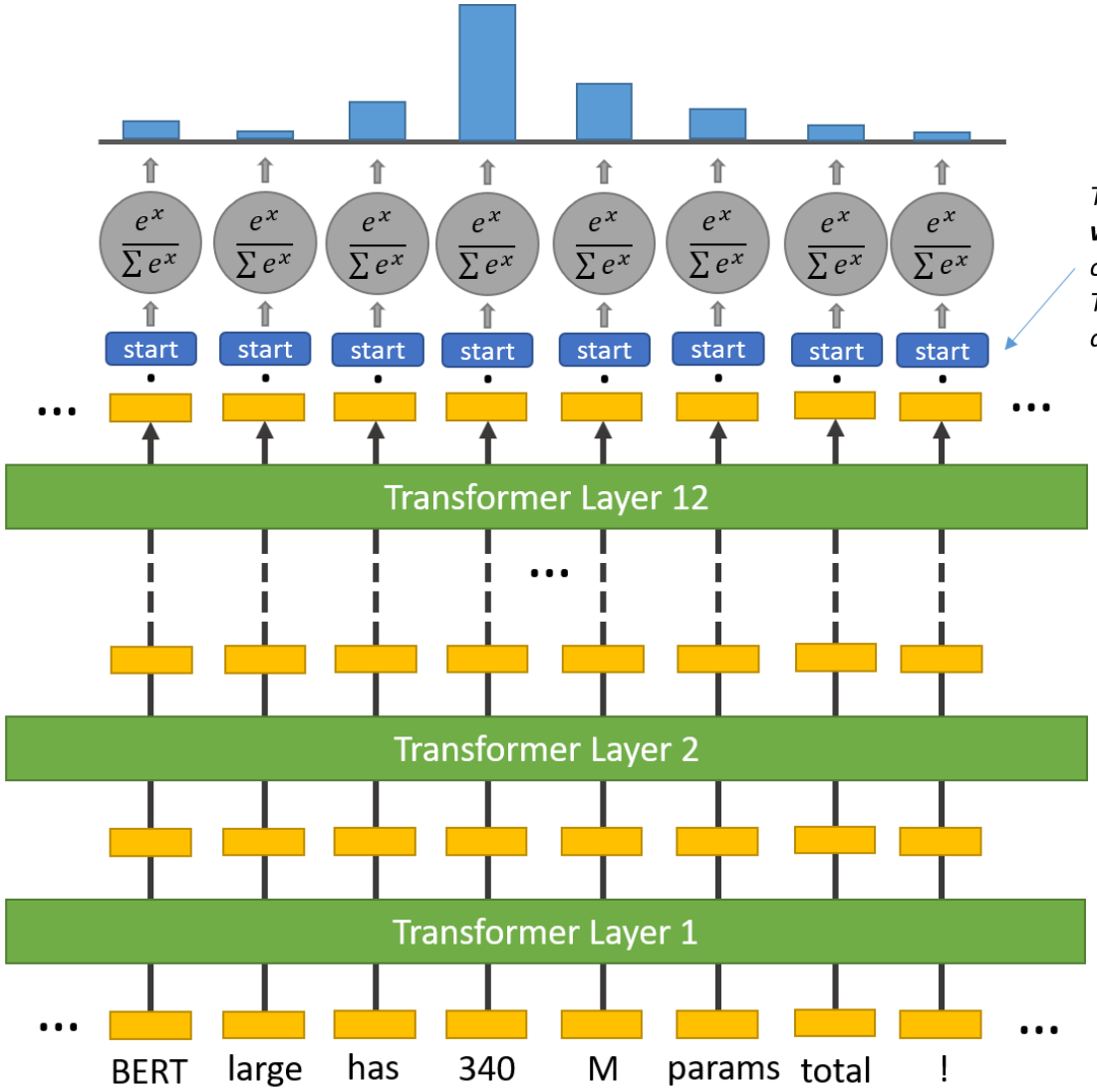
Out[6]:
```
dict_keys(['input_ids', 'token_type_ids', 'attention_mask'])
```

In [7]:
```
encoded_dict["input_ids"]
```

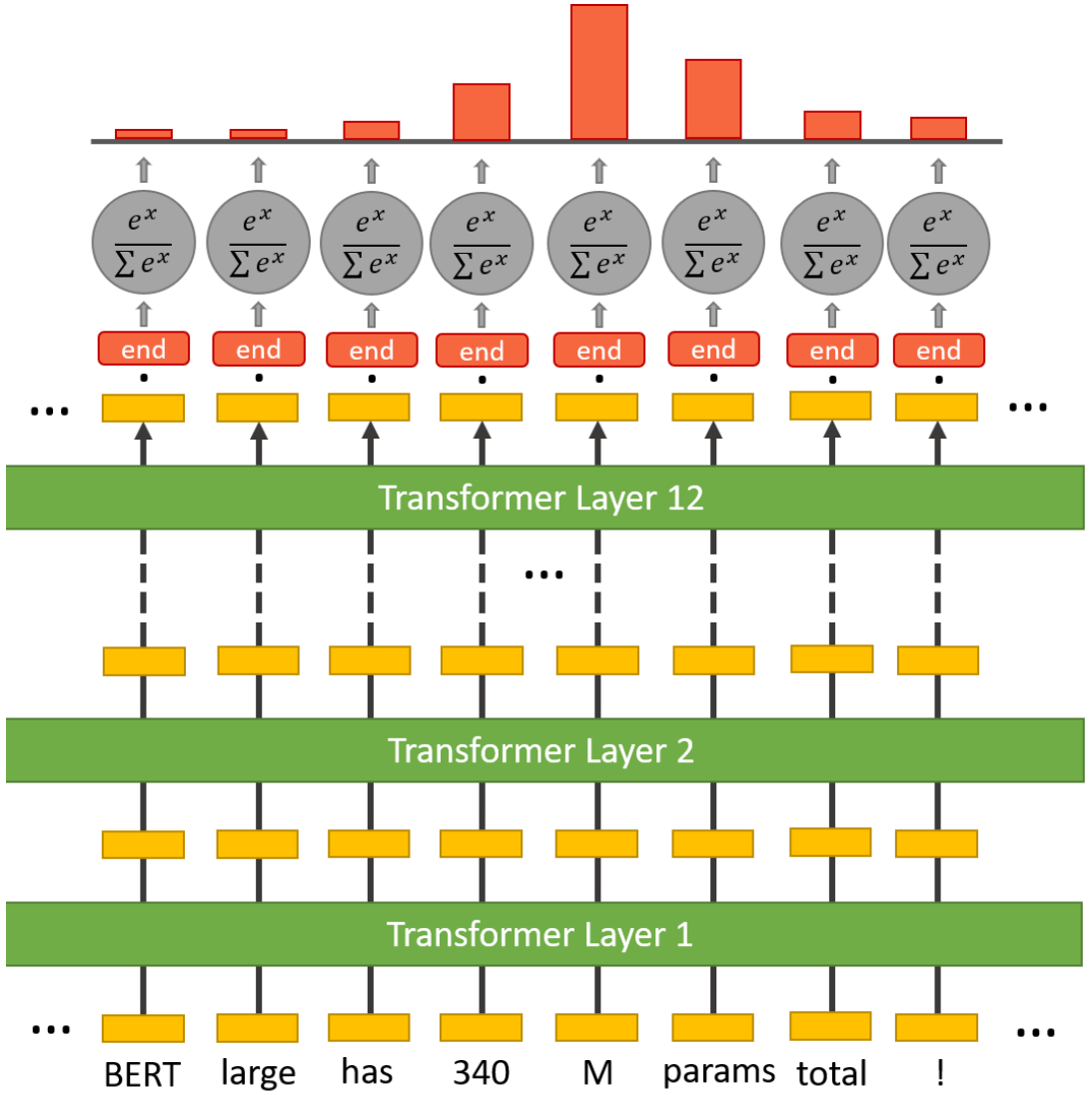
Out[7]:
```
tensor([[  101,  1731,  1242, 11934,  1674,   139,  9637,  1942,   118,  1415,
          1138,   136,   102,   139,  9637,  1942,   118,  1415,  1110,  1541,
          1992,   119,   119,   119,  1122,  1144,  1572,   118,  8798,  1105,
          1126,  9712,  4774,  3408,  2060,  1104,   122,   117,  5507,  1527,
           117,  1111,   170,  1703,  1104, 16984,  2107, 11934,   106, 17762,
         16609,  4679,  1122,  1110,   122,   119,  3236, 13745,   117,  1177,
          5363,  1122,  1106,  1321,   170,  2337,  1904,  1106,  9133,  1106,
          1240, 17492,  1830,  5374,   119,   102,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
             0,     0,     0,     0,     0,     0,     0,     0,     0,     0]])
```

In [8]:
```
tokenizer.decode(encoded_dict["input_ids"].squeeze())
```

Out[8]:
```
'[CLS] How many parameters does BERT - large have? [SEP] BERT - large is really big... it has 24 - layers and a
n embedding size of 1, 024, for a total of 340M parameters! Altogether it is 1. 34GB, so expect it to take a co
uple minutes to download to your Colab instance. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [P
AD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PA
D] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]'
```



*This length 768 vector is the **weights** for the start token classifier.*
*The **same weights** are applied to **every position**.*



- Start: “340”; end: “M” → predicted answer: “340 M”

The tools we need

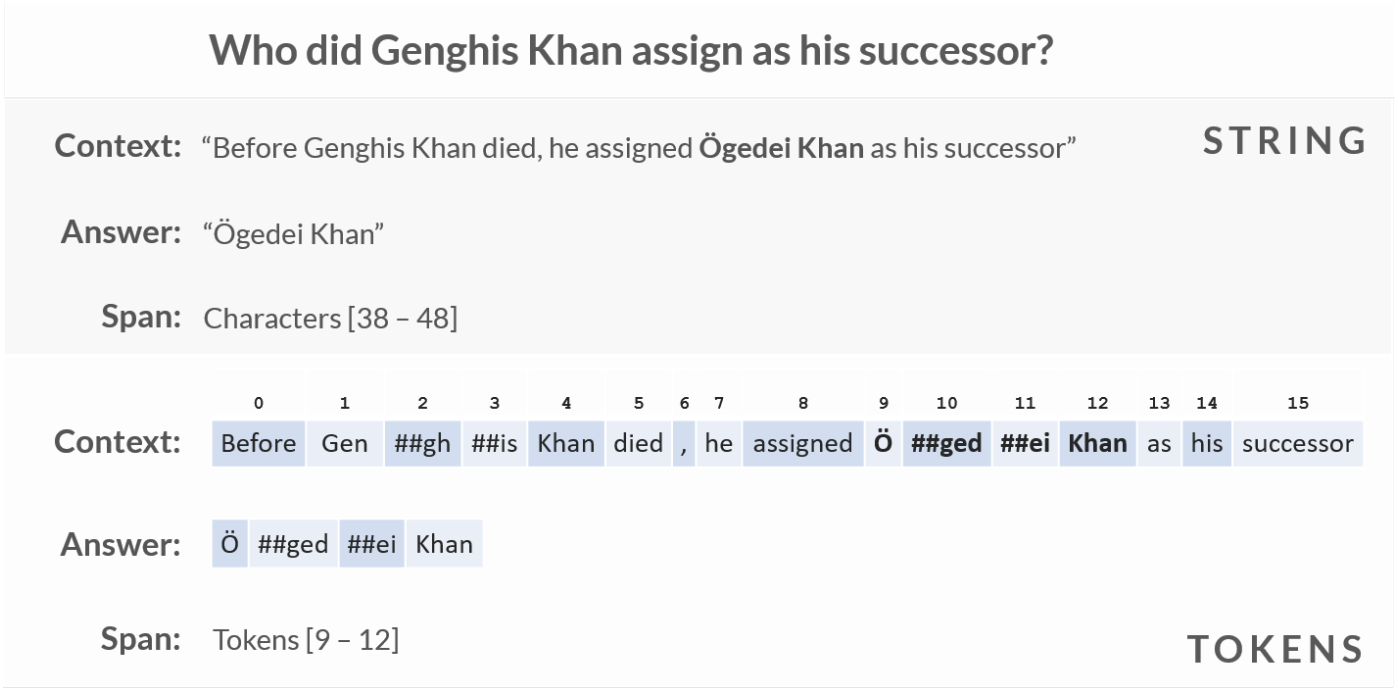
- We'll use the HuggingFace transformers python library, the most widely accepted and powerful PyTorch interface for working with BERT. The library includes a variety of different pre-trained transformer models, the library also includes pre-built modifications of these models suited to your specific task
- Check out their website if interested, anyone can open source a trained model here (lots of university and big tech companies have published a range of open source models)
- Beyond BERT, a number of other transformer-based model architectures are available

Today's lesson topics

- We'll instantiate the `BertForQuestionAnswering` model from a `bert-base-cased` model, which is a BERT model pre-trained on self-supervised tasks: Masked Language Modelling and Next Sentence Prediction. The pre-trained `bert-base-cased` weights have not been fine-tuned for a specific downstream task, but the `BertForQuestionAnswering` model already has the correct algorithmic structure for question answering (in terms of dealing with inputs and outputs)
- The BERT model is great at making sense of text, but we need to further train it on our specific downstream task (question answering) before it will perform well
- We'll then go through all the pre-processing of the dataset and write the fine-tuning function.
- We'll then use the test dataset to compare our F1 score with that of the fine-tuned model for question answering, namely `bert-large-uncased-whole-word-masking-finetuned-squad`
- In real-real life, assuming you have your custom dataset, you may wish to load your `BertForQuestionAnsering` model using the `bert-large-uncased-whole-word-masking-finetuned-squad` weights instead of `bert-base-cased`, and run a further fine-tuning based on your custom dataset

First challenge: keeping track of tokens

- The SQuAD dataset gives us the location of the answer in terms of its start character index in the context string. But BERT needs the location in terms of the input tokens!
- BERT's tokenization method is custom; for example, it preserves punctuation as tokens, and it will break some words into subwords. If there are multiple adjacent whitespaces, this information will be lost. Because of that, the SQuAD dataset can't tell us the indices of the tokens, we have to determine that ourselves.



- How do we (programmatically) identify, say, which of those 15 tokens contain characters 38 - 48 of the original string, i.e. the answer?

- I followed the strategy detailed in one of the bibliography references, although there may be better strategies to explore (BERT might maintain a mapping for each BERT token back to its character offset in the original context string).

- Below are the steps we'll follow:
  - **Step 1**: Feed the **answer** string (not the full context) into the BERT tokenizer to determine the number of tokens it breaks into.
  - **Step 2**: In the context, replace the answer with a string made up of [MASK] tokens, matching the number of tokens in the answer. I use the [MASK] token because it has its own entry in the vocabulary and shouldn't ever appear in normal text.
  - **Step 3**: Feed the modified text into the BERT tokenizer to tokenize and encode everything.
  - **Step 4**: Locate the [MASK] tokens in the encoded result. This is easy, since the [MASK] token has a specific ID and won't appear anywhere else in the sequence.
  - **Step 5**: Record the start and end indices of the answer (i.e. the first and last [MASK] tokens which will be our labels), and finally "repair" the encoded sequence by switching the MASK token IDs out for the original answer token IDs.

How to train the model on a GPU

- Access to adequate computational resources is perhaps the main blocker for personal ML projects
- Use a GPU where possible. You can use a GPU for free using Google Colab, although the interface is not exactly user friendly; it requires everything to be run from a notebook (but you can still upload .py modules to your Google Drive and then call them from your notebook), and it requires your computer to stay active with the Google Colab windows open for the whole duration of the training
- Open a notebook, then go to `Edit` → `Notebook settings` → `Hardware accelerator` → select "GPU" and click Save.
- Check you GPU is available as below. Note: your GPU name might differ

```
import torch
torch.cuda.is_available()
```

```
True
```

```
torch.cuda.get_device_name()
```

```
'Tesla T4'
```

- All the code I wrote will automatically identify a GPU if available and will put all tensors on a GPU, else will use a CPU. If you write your own model, you must explicitly set your tensors on a cuda device if you want to use a GPU. Unfortunately, PyTorch only supports Nvidia GPUs.

Using Google Drive for saving, storing and loading models

- Connecting the notebook to your Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

- Enter your Google details when prompted, then use `'/content/drive/My Drive'` as your home Google Drive directory and treat it as a local path in the notebook.
- Training on a single Google Colab's GPU takes around 1-2hrs per epoch, on a standard CPU it might take 30-50 times longer

CODE BREAK 1 – Preprocessing

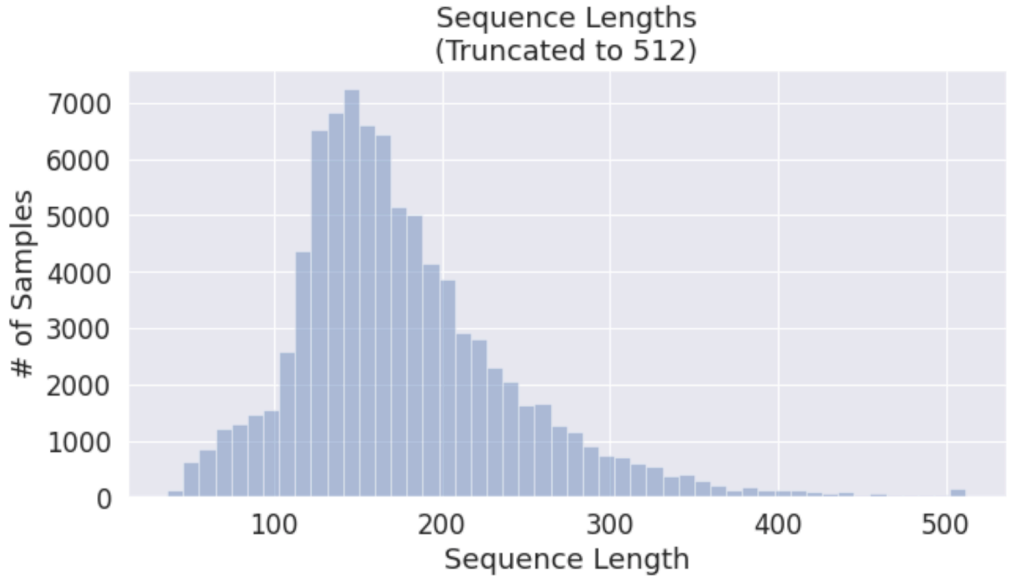
- Let's start looking at some helper functions https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/utils.py

- Code for tokenizing and encoding the dataset: https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/preprocess_dataset.py

- Full and exhaustive documentation is provided in the code, let's get our hands on it!

Choosing max sequence length

- Normally, BERT takes a sequence length up to 512 tokens.
- However, if most sequences are much shorter, reducing the maximum sequence length might be preferable to speed up training. If a sequence is longer, it will be truncated
- Do some statistics on your dataset, and consider for each possible maximum length, how many “answers” in the question + context pairs will be truncated, and therefore how many training samples will be unusable, as a function of sequence length
- Other truncation strategies may be adopted to deal with truncated answers (e.g. truncating the start of the context instead of the end), but we’ll ignore this for now



Finally, here are the number of training samples which would be impacted, given a handful of different choices of `max_len`.

```
How many comments will be truncated?

max_len = 128  -->   69,082 of  87,599  (78.9%)  will be truncated
max_len = 256  -->    9,812 of  87,599  (11.2%)  will be truncated
max_len = 300  -->    4,568 of  87,599  ( 5.2%)  will be truncated
max_len = 384  -->    1,087 of  87,599  ( 1.2%)  will be truncated
max_len = 512  -->      135 of  87,599  ( 0.2%)  will be truncated
```

Things to consider

- Training time is quadratic with max_len (the maximum sequence length). This means that max_len = 512 will take 4x longer to train than max_len = 256, and 16x longer than max_len = 128 !
- Truncating the samples to a shorter length will presumably hurt accuracy, due to the loss of information. How much it hurts depends on the dataset, though.
- The combination of max_len and batch_size needs to fit within the memory limits of our GPU. For a Tesla K80 (which has 12GB of RAM), with batch_size = 16, the maximum length we can use (without running of memory) is about max_len = 400.
- I've not yet found a resource which lets you calculate how your GPU RAM affects the sequence length you can use, so for now I decided to go with the suggested value of the bibliography's source, i.e. we'll be using a maximum sequence length of 384.
- Something else you may wish to consider is the train/validation split. Today we'll split the training dataset into 95% train / 5% validation. The test dataset is completely separate.

Hyper-parameters

- For the purposes of fine-tuning, the authors of the BERT paper recommend choosing from the following values:
  - Batch size: 16, 32
  - Learning rate (Adam): 5e-5, 3e-5, 2e-5
  - Number of epochs: 2, 3, 4. This is also a parameter to the "learning rate scheduler" in the section.
- However, some recent papers on applications of BERT to legal text found that lots of work blindly adopts the hyper-parameters recommended in the original BERT paper, but actually a broader hyperparameter search can lead to substantially better performance
- Due to practical constraints, we won't do a hyperparameter search (training a model on Google Colab took me 4 hours, imagine multiplying that for all the possible permutations of batch size, learning rate and number of epochs!)
- We'll use these values:
  - Batch size: 16
  - Learning rate: 2e-5
  - Epochs: 3

Some optimizers concepts

- We'll be using the AdamW optimizer
- The AdamW optimizer, on top of the learning rate, also requires and epsilon parameter, i.e. "a very small number to prevent any division by zero in the implementation" (from here). We'll use eps = 1e-8 is
- You can find the creation of the AdamW optimizer in run_glue.py here.
- The learning rate scheduler is responsible for updating the learning rate over the course of the training. Generally speaking, you want the learning rate to gradually get smaller and smaller so that training makes gradually finer adjustments to the weights.
- This decay needs to happen *across all of the training epochs*, so this is why we also need to specify the number of epochs we want to train for when we use the learning rate scheduler.

- A core difference between the training/validation and the test dataset is that the former contains one answer per question, whereas the latter contains multiple valid answers, so we need to be flexible in how we score the model

    - e.g. question: "*Where did Super Bowl 50 take place?*"
    - Possible valid answers:
        - *Santa Clara, California*
        - *Levi's Stadium*
        - *Levi's Stadium in the San Francisco Bay Area at Santa Clara, California.*

- There is no need to randomly shuffle the samples when in prediction mode as the weights are not being updated based on the results

CODE BREAK 2 – Fine/tuning loop and predictions

- Let's build the fine-tuning loop; using Google Colab and one GPU, your model might take 3-6 hrs to train for 3 epochs.
  https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/fine_tuning.py

- After the training is done, we can load the model and run the predictions on the test dataset
  https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/prediction_loop.py

  - **For practical purposes, you can download and use this model (431MB) which I have already trained on Google Colab:**
    https://drive.google.com/file/d/1N9e91W4EqfL_L6t7AmF0gU0yhsb9IJRF/view?usp=sharing

- And here we can score our model
  https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/scores.py

A note on loading and saving models in PyTorch

- When saving models in PyTorch, it's preferable to save the dictionary containing all the weights, as opposed to saving the model as is.
- This is the **non-ideal** way to do it, by saving the model directly

Saving `model` (must be instance of `torch.nn.Module`).

```
import torch
torch.save(model, "my_model.pt")
```

Loading `model` (must be instance of `torch.nn.Module`).

```
model = torch.load("my_model.pt")
```

- **Note:** If you saved your model when it was on a cuda/GPU device (e.g. on Google Colab) and then load it on a device with only a CPU (e.g. your laptop) you will have to add an extra parameter as follows

```
model = torch.load("my_model.pt", map_location=torch.device('cpu'))
```

- The disadvantage of the approach on the previous slide is that the serialized data is bound to the specific classes and the exact directory structure used when the model is saved. The reason for this is because pickle (a library which torch uses in the background to save and load models) does not save the model class itself. Rather, it saves a path to the file containing the class, which is used during load time. Because of this, your code can break in various ways when used in other projects or after refactors.
- For more details, see https://pytorch.org/tutorials/beginner/saving_loading_models.html
- The best way to do it is by saving the dictionary with the weights, as done below, so it is portable

Saving `model` (must be instance of `torch.nn.Module`).

```
import torch
torch.save(model.state_dict(), "my_model.pt")
```

Loading `model` (must be instance of `torch.nn.Module`).
Note how you need to instantiate an un-trained (or un-tuned) model first, then load its weights.

```
from transformers import BertForQuestionAnswering
model = BertForQuestionAnswering.from_pretrained("bert-base-cased")
model.load_state_dict(torch.load("my_model.pt"))
```

`model.load_state_dict` is an inplace function, and if you put a `print` statement around it, it should print `<All keys matched successfully>`

- **Note:** In case you are saving the latest checkpoint to continue training later, you need to save the optimizer's state as well. You can call `optimizer.state_dict()` and use the same code as above

Check for overfitting

- Our training code will store a dictionary, which in my case looks like this

```
{'epoch_1': {'training_loss': 1.3099741377829537,
  'valid_loss': 0.9752135873385713,
  'valid_accuracy': 0.7172894526339847,
  'training_time': '0:51:17',
  'valid_time': '0:01:54'},
 'epoch_2': {'training_loss': 0.7837005815446401,
  'valid_loss': 0.9346368016991798,
  'valid_accuracy': 0.7318592160895898,
  'training_time': '0:51:10',
  'valid_time': '0:01:54'},
 'epoch_3': {'training_loss': 0.5739485899948434,
  'valid_loss': 1.0036209920996721,
  'valid_accuracy': 0.7338589875442807,
  'training_time': '0:51:09',
  'valid_time': '0:01:54'}}
```

- We can plot training and validation loss as a function of epochs to check for overfitting

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

# Use plot styling from seaborn.
sns.set(style='darkgrid')

# Increase the plot size and font size.
sns.set(font_scale=1.2)
plt.rcParams["figure.figsize"] = (10,6)

# Plot the learning curve.
epochs = [1, 2, 3]
training_loss = [stats[f"epoch_{i}"]["training_loss"] for i in epochs]
valid_loss = [stats[f"epoch_{i}"]["valid_loss"] for i in epochs]
plt.plot(epochs, training_loss, 'b-o', label="Training")
plt.plot(epochs, valid_loss, 'g-o', label="Validation")

# Label the plot.
plt.title("Training & Validation loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.show()
```

Scoring our model – part 1

- There are two standard approaches to scoring results on the SQuAD benchmark.
- The first approach is the *exact match*.
- We count the number of predicted start and end tokens that are equal to the correct ones. For this reason, there are actually two "points" for every sample. For every correctly predicted start token we score 1 point, and the same for every correctly predicted end token, so each prediction can score 0, 1 or 2.
- To handle the multiple possible answers, we score our predictions against each of the possible answers separately, and pick whichever answer best matches our prediction (**not** the sum!)
- For example, if a question has 4 valid answers and our exact match score for each valid answer is `[0, 2, 1, 2]`, we will take 2 – i.e. `max([0, 2, 1, 2])` – as the exact match score for that answer, and not 7.
- The final exact match score will be the total of all the scores divided by twice the number of questions (i.e. the maximum possible score, considering each question is scored independently for start and end tokens)

Scoring our model – part 2

- The second approach is less rigid and is the *F1 score,* allowing us to give the model some credit for predicting a span which partially intersects the correct one. Here is the original code.
- For each predicted sample, $F1 = \frac{2 \cdot precision \cdot recall}{precision+recall}$
- The recall is the proportion of predicted token which overlap with the correct answer tokens, and the precision is the proportion of correctly predicted token out of all predicted tokens; example
    - Context: "*Before Genghis Khan died, he assigned Ögedei Khan as his successor.*"
    - Question: *"Who did Genghis Khan assigned as his successor?"*
    - Correct answer: "*Ögedei Khan*" – i.e. tokens 9 - 12: `['Ö', '##ged', '##ei', 'Khan']`
    - Predicted answer: "*he assigned Ögedei*" – i.e. tokens 7 - 11: `['he', 'assigned', 'Ö', '##ged', '##ei']`
    - 3 of the predicted tokens overlap with the 4 correct tokens, so recall = 3/4 = 0.75.
    - 3 of the 5 predicted tokens are correct, so we get precision = 3/5 = 0.6.
    - Exact match score would be 0, but F1 score is 0.667
- To handle the multiple possible answers per question sample, we calculate the F1 score for each of the possible answers separately, and pick the highest score as the F1 score for that sample
- The final F1 score will be the average of all the best F1 scores for each prediction

A quick comparison

My results

- Exact match score: 83.8%
- Average F1: 0.864

Fully Pre-Tuned BERT-large model

`bert-large-uncased-whole-word-masking-finetuned-squad`

- Exact match score: 88.8%
- Average F1: 0.914

Original BERT paper

Table 2:

$BERT_{LARGE}$ (Ens.+TriviaQA)

- Exact match score: 87.4%
- Average F1: 0.932

- Our model is definitely not performing badly at all. While a 3-6 hrs training on one GPU, or several days on a CPU (not recommended!) might seem a lot to you, this fine-tuning is actually extremely computationally inexpensive by research/academic standards compared to the resources required to pre-train BERT from randomly initialized weights. This shows how well BERT can perform on downstream tasks with just a few epochs of fine-tuning and how versatile it is.
- Worth bearing in mind that the BERT score above is obtained by using both an ensemble and some "dataset augmentation", so it's not really a fair comparison to our own simpler fine-tuning process, which nonetheless is really good.

CODE BREAK 3 – Build a Chatbot

- Chatbot code https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/modules/user_interface.py

- Now let's open this script and try playing with the chatbot! https://github.com/AI-Core/Practical-ML-DS/blob/master/Chapter%205.%20NLP/Module%204.%20BERT%20and%20HuggingFace/bert_for_question_answering/scripts/chatbot_script.py

Any Questions?