# PyTorch Transformer:
# (1)Tokenizer,
# (2)Text Classification and
# (3)Question Answering

Kanru Wang

Dec 2021

# Section 1: Tokenizer for Transformer

# Reference

- Huggingface Documentation
  - Prepare input_ids, token_type_ids, attention_mask: https://huggingface.co/docs/transformers/preprocessing
  - Why no need for pre-processing before tokenization: https://huggingface.co/docs/tokenizers/python/master/pipeline.html and https://huggingface.co/docs/tokenizers/python/master/components.html
  - Train a tokenizer: https://huggingface.co/docs/tokenizers/python/latest/quicktour.html

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
batch_sentences = ["Hello I'm a tokenizer sentence example 😊",
                   "And another sentence",
                   "And the very very last one"]
for sent in batch_sentences:
    print(tokenizer.tokenize(sent))
  ['Hello', 'I', "'", 'm', 'a', 'token', '##izer', 'sentence', 'example', '[UNK]']
  ['And', 'another', 'sentence']
  ['And', 'the', 'very', 'very', 'last', 'one']
encoded_inputs = tokenizer(batch_sentences, padding=True, truncation=True, return_offsets_mapping=True, return_tensors="pt")
print(encoded_inputs)
  {'input_ids': tensor([[101, 8667,  146,  112,  182,  170, 22559, 17260, 5650,  1859,   100,   102], # for BERT, 101 is [CLS], 102 is [SEP], 0 is [PAD]
                        [101, 1262, 1330, 5650,  102,    0,     0,     0,    0,     0,     0,     0],
                        [101, 1262, 1103, 1304, 1304, 1314,  1141,   102,    0,     0,     0,     0]]),
   'token_type_ids': tensor([[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]]),
   'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                             [1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                             [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]]),  # attention_mask: tells the model not to pay attention to padding
   'offset_mapping': tensor([
   [[ 0,  0], [ 0,  5], [ 6,  7], [ 7,  8], [ 8,  9], [10, 11], [12, 17], [17, 21], [22, 30], [31, 38], [39, 40], [0, 0]],
   [[ 0,  0], [ 0,  3], [ 4, 11], [12, 20], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [0, 0]],
   [[ 0,  0], [ 0,  3], [ 4,  7], [ 8, 12], [13, 17], [18, 22], [23, 26], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [0, 0]]])}  # optional
tokenizer.batch_decode(encoded_inputs['input_ids'])
  ["[CLS] Hello I'm a tokenizer sentence example [UNK] [SEP]",
   '[CLS] And another sentence [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
   '[CLS] And the very very last one [SEP] [PAD] [PAD] [PAD] [PAD]']
```

https://huggingface.co/docs/transformers/preprocessing

```
For BERT models, the input is represented like this: [CLS] Sequence A [SEP] Sequence B [SEP]
batch_sentences = ["Hello I'm a single sentence",
                   "And another sentence",
                   "And the very very last one"]
batch_of_second_sentences = ["I'm a sentence that goes with the first sentence",
                             "And I should be encoded with the second sentence",
                             "And I go with the very last one"]
encoded_inputs = tokenizer(batch_sentences, batch_of_second_sentences)
print(encoded_inputs)
  {'input_ids': tensor([
   [101, 8667,  146,  112,  182,  170, 1423, 5650,  102,   146,  112,  182,  170, 5650, 1115, 2947, 1114, 1103, 1148, 5650, 102],
   [101, 1262, 1330, 5650,  102, 1262,  146, 1431, 1129, 12544, 1114, 1103, 1248, 5650,  102,    0,    0,    0,    0,    0,   0],
   [101, 1262, 1103, 1304, 1304, 1314, 1141,  102, 1262,   146, 1301, 1114, 1103, 1304, 1314, 1141,  102,    0,    0,    0,   0]]),
   'token_type_ids': tensor([
   [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  # token_type_ids is not required for all models
   [0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],  # token_type_ids differentiates 1st sentence & 2nd sentence
   [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]]),
   'attention_mask': tensor([
   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
   [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0]]),
   'offset_mapping': tensor([
   [[0, 0], [0, 5], [6,  7], [ 7,  8], [ 8,  9], [10, 11], [12, 18], [19, 27], [ 0,  0], [ 0,  1], [ 1,  2], [ 2,  3], [ 4,  5], [ 6, 14], [15, 19], [20, 24], [25, 29], [30, 33], [34, 39], [40, 48], [0, 0]],
   [[0, 0], [0, 3], [4, 11], [12, 20], [ 0,  0], [ 0,  3], [ 4,  5], [ 6, 12], [13, 15], [16, 23], [24, 28], [29, 32], [33, 39], [40, 48], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [0, 0]],
   [[0, 0], [0, 3], [4,  7], [ 8, 12], [13, 17], [18, 22], [23, 26], [ 0,  0], [ 0,  3], [ 4,  5], [ 6,  8], [ 9, 13], [14, 17], [18, 22], [23, 27], [28, 31], [ 0,  0], [ 0,  0], [ 0,  0], [ 0,  0], [0, 0]]])} # optional
tokenizer.batch_decode(encoded_inputs['input_ids'])
  ["[CLS] Hello I'm a single sentence [SEP] I'm a sentence that goes with the first sentence [SEP]",
   '[CLS] And another sentence [SEP] And I should be encoded with the second sentence [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]',
   '[CLS] And the very very last one [SEP] And I go with the very last one [SEP] [PAD] [PAD] [PAD] [PAD]']
```

Why no need for text pre-processing before tokenization?

When calling encode() or encode_batch(), the input text(s) go through the following pipeline:

1. Normalization (Lowercasing (for uncased models) and removing accented characters)
2. Pre-Tokenization (Splitting on spaces and punctuations, while keeping punctuations and removing spaces)
3. The Model (Splitting uncommon words into tokens in the vocabulary of the model (e.g. "tokenizers" -> "token", "##izer", "##s"), and mapping those tokens to their corresponding IDs. This part of a tokenizer needs to be trained or pretrained.) -> See next page
4. Post-Processing (Adding special tokens. E.g. [CLS], [SEP] for BERT)

No need to lower case input data for a BERT uncased model. Also, it does not make sense to lower case input data for a BERT cased model.

https://stackoverflow.com/questions/62466514

https://huggingface.co/docs/tokenizers/python/master/pipeline.html

About the model in a tokenizer…

- The model part of a tokenizer is usually pretrained, and thus the name "pretrained tokenizer".
- Training the model part of a tokenizer means it will learn merge rules by:
  - Start with all the characters present in the training corpus as tokens.
  - Identify the most common pair of tokens and merge it into one token. (No token is allowed to be bigger than any word returned by the pre-tokenizer.)
  - Repeat until the vocabulary (the number of tokens) has reached the size we want.
- Use the pretrained tokenizer associated with the pretrained model. The training data must be tokenized the same way the pre-training data (used to pre-train the model) is tokenized.
- If the Normalization or Pre-Tokenization process is changed, need to retrain the tokenizer from scratch afterward.

https://huggingface.co/docs/tokenizers/python/latest/quicktour.html

What are the methods a BertTokenizer has?

BertTokenizer inherits from PreTrainedTokenizer which inherits from PreTrainedTokenizerBase

- tokenize()

Converts a string into a sequence of tokens, using the tokenizer.

- convert_tokens_to_ids()

Converts a token string (or a sequence of tokens) into a single integer id (or a sequence of ids), using the vocabulary.

- encode()

Converts a string into a sequence of ids (integer), using the tokenizer and vocabulary. Same as doing self.convert_tokens_to_ids(self.tokenize(text)).

- encode_plus()

Tokenize and prepare for the model a sequence or a pair of sequences; while encode() only does the first four steps.

According to this article, encode_plus():

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 max length.
5. Create the attention masks which explicitly differentiate real tokens from [PAD] tokens.

- __call__()

Main method to tokenize and prepare for the model one or several sequence(s) or one or several pair(s) of sequences.

It implements encode_plus() or batch_encode_plus()

What are the methods a BertTokenizer has?

- convert_ids_to_tokens()

Converts a single index or a sequence of indices in a token or a sequence of tokens, using the vocabulary and added tokens.

This is the inverse of convert_tokens_to_ids().

- convert_tokens_to_string()

Converts a sequence of tokens in a single string. The most simple way to do it is " ".join(tokens) but we often want to remove sub-word tokenization artifacts at the same time.

This is the inverse of tokenize().

- decode()

Converts a sequence of ids in a string, using the tokenizer and vocabulary with options to remove special tokens and clean up tokenization spaces. Similar to doing self.convert_tokens_to_string(self.convert_ids_to_tokens(token_ids)).

This is the inverse of encode().

Pre-processing

String → Tokens → IDs

Post-processing

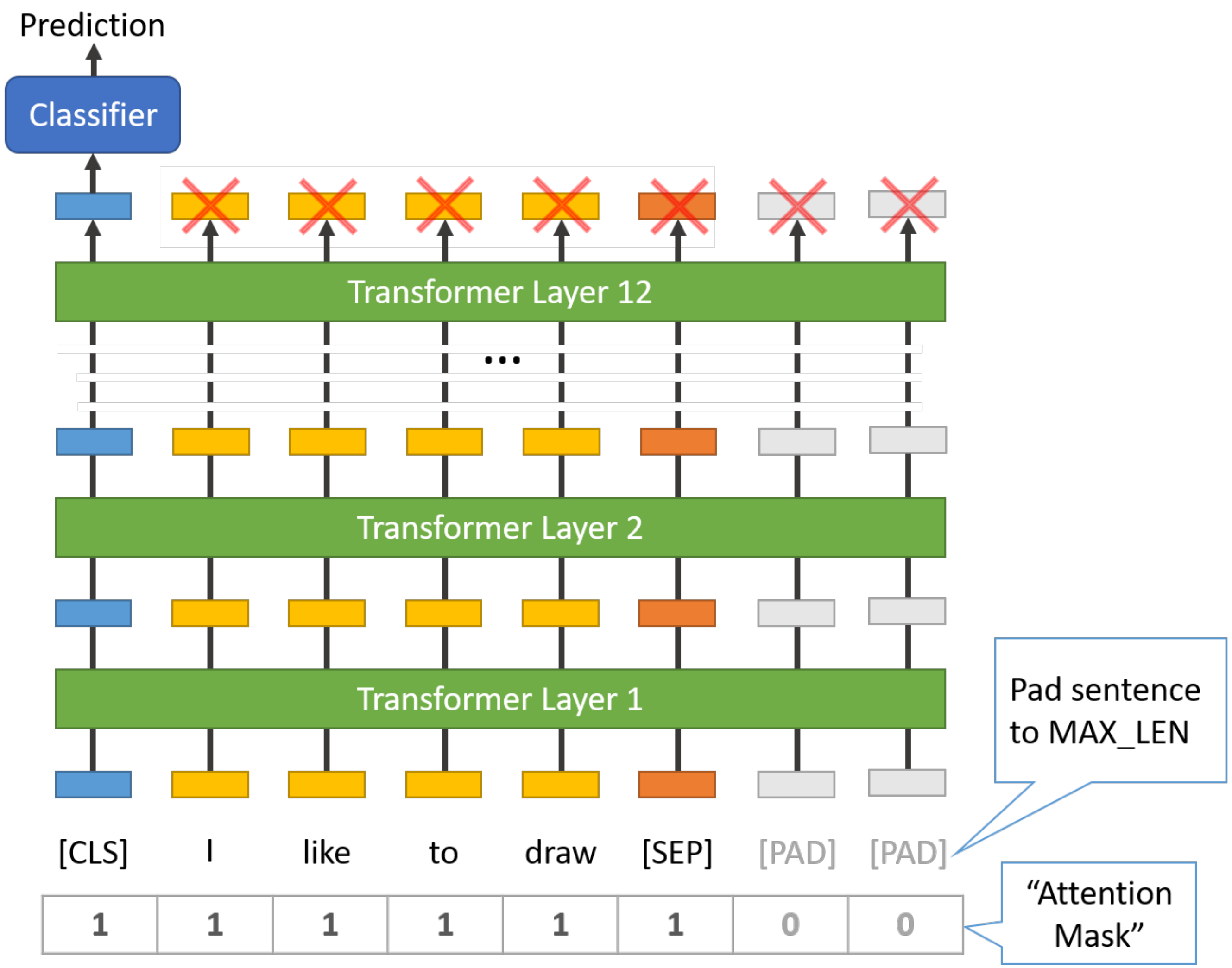
IDs → Tokens → String

https://huggingface.co/docs/transformers/main_classes/tokenizer

# Section 2: PyTorch Transformer Text Classification

Reference

- Article
  - https://mccormickml.com/2019/07/22/BERT-fine-tuning/
- Kaggle Notebook
  - https://www.kaggle.com/kanruwang/two-ways-to-prepare-pytorch-dataset-for-bert
  - https://www.kaggle.com/ahmedattia143/nlp-disaster-pytorch-bert-kfold
  - https://www.kaggle.com/nxhong93/tweet-predict1
  - https://www.kaggle.com/yaroshevskiy/bert-base-2-epochs (Using pytorch_pretrained_bert package instead of transformers package)
  - https://www.kaggle.com/yuval6967/toxic-train-bert-base-pytorch and https://www.kaggle.com/yuval6967/toxic-bert-plain-vanila and https://www.kaggle.com/abhishek/pytorch-bert-inference (Using pytorch_pretrained_bert package instead of transformers package)
- Huggingface Documentation
  - https://huggingface.co/transformers/v4.5.1/training.html and https://huggingface.co/transformers/v4.11.3/custom_datasets.html and https://huggingface.co/transformers/v4.11.3/training.html




- On the output of the final (12th) transformer, only the first embedding (corresponding to the "[CLS]" token) is used by the classifier, because BERT is trained to only use this "[CLS]" token for classification. Each of the other embeddings represents each word, not the whole sentence.
- All sentences must be padded or truncated to a single, fixed length.
- The maximum sentence length is 512 tokens.

https://mccormickml.com/2019/07/22/BERT-fine-tuning/

```python
from transformers import BertForSequenceClassification
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
```

According to this post, the from_pretrained() in below does the following:

- find the correct base model class to initialize
- use init_weights() so that layers that are not pretrained (e.g. final classification layer) still get initialized
- find the file with the pretrained weights
- overwrite the weights of the model with the pretrained weights where applicable

Below is the source code of BertForSequenceClassification

```python
class BertForSequenceClassification(BertPreTrainedModel):
    def __init__(self, config):
        super().__init__(config)
        self.num_labels = config.num_labels
        self.config = config
        self.bert = BertModel(config)
        classifier_dropout = (
            config.classifier_dropout if config.classifier_dropout is not None else config.hidden_dropout_prob
        )
        self.dropout = nn.Dropout(classifier_dropout)
        self.classifier = nn.Linear(config.hidden_size, config.num_labels)  # train this layer and fine tune the model
        self.post_init() # post_init() calls init_weights() which initializes weights
    def forward
        …
```

**Train Dataloader**

All Train text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

**Val Dataloader**

All Val text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

**Epoch**

- Train
- Val

**Test Dataloader**

All Test text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs

**Inference**

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification, AdamW, get_scheduler
from torch.utils.data import TensorDataset, DataLoader
from sklearn.model_selection import train_test_split
from datasets import load_metric

train_texts, val_texts, train_labels, val_labels = train_test_split(texts, labels, test_size=.2)

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)

train_dataset = TensorDataset(*[torch.tensor(v) for k, v in train_encodings.items()], torch.tensor(train_labels))
val_dataset = TensorDataset(*[torch.tensor(v) for k, v in val_encodings.items()], torch.tensor(val_labels))
test_dataset = TensorDataset(*[torch.tensor(v) for k, v in test_encodings.items()])

train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=16)
val_dataloader = DataLoader(val_dataset, batch_size=16)
test_dataloader = DataLoader(test_dataset, batch_size=16)
```

https://www.kaggle.com/kanruwang/two-ways-to-prepare-pytorch-dataset-for-bert

Train Dataloader

All Train text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Label

Val Dataloader

All Val text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Label

Epoch

Train

Val

Test Dataloader

All Test text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs

Inference

```
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')

model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2).to(device)

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

num_epochs = 3
num_training_steps = num_epochs * len(train_dataloader)

lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps
)
```

https://www.kaggle.com/kanruwang/two-ways-to-prepare-pytorch-dataset-for-bert

Train Dataloader

All Train text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

Val Dataloader

All Val text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

Epoch

Train

Val

Test Dataloader

All Test text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs

Inference

```
for epoch in range(num_epochs):
    model.train()  # for Batchnorm and Dropout to work properly
    for batch in train_dataloader:
        input_ids = batch[0].to(device)  # because here we use TensorDataset, the keys are lost, but positions remain
        token_type_ids = batch[1].to(device)
        attention_mask = batch[2].to(device)
        labels = batch[3].to(device)
        outputs = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids, labels=labels)
        loss = outputs.loss
        loss.backward() # computes the derivative of the loss w.r.t. each parameter using backprop
        optimizer.step()  # optimizer updates each parameter using the its gradient and the optimizer rules
        lr_scheduler.step()
        optimizer.zero_grad()  # clear old gradients from this step's loss.backward() for the next step
    model.eval()  # for Batchnorm and Dropout to work properly
    metric = load_metric("accuracy")
    for batch in val_dataloader:
        input_ids = batch[0].to(device)
        token_type_ids = batch[1].to(device)
        attention_mask = batch[2].to(device)
        labels = batch[3].to(device)
        with torch.no_grad():  # deactivate backprop to reduce memory usage
            outputs = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids, labels=labels)
        logits = outputs.logits
        predictions = torch.argmax(logits, dim=-1)
        metric.add_batch(predictions=predictions, references=labels)
    print(metric.compute())
```

https://www.kaggle.com/kanruwang/two-ways-to-prepare-pytorch-dataset-for-bert

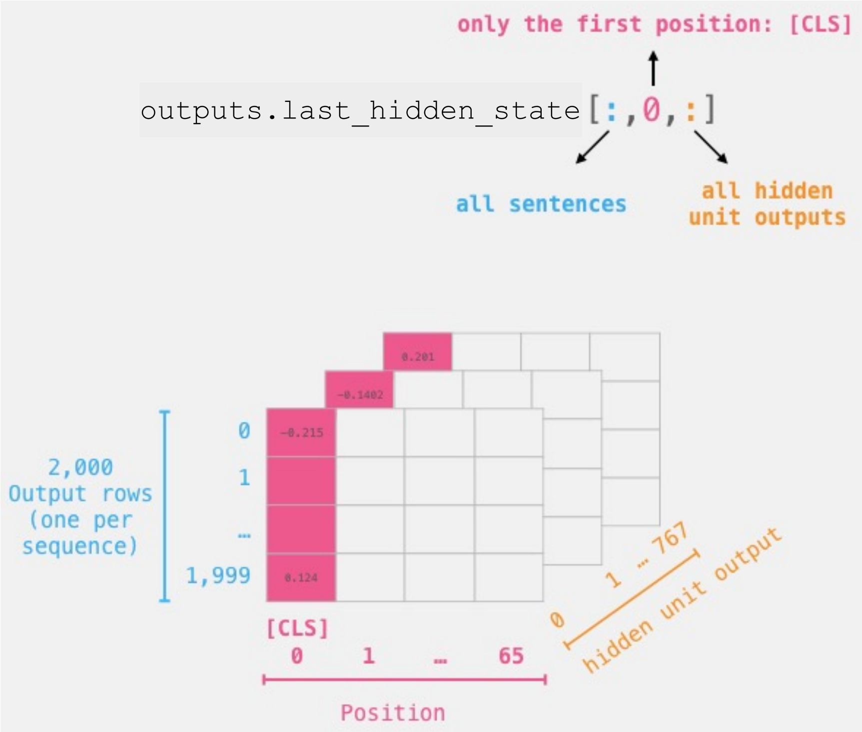
# If we want to keep the weights of the pre-trained encoder frozen…

a) If we want to keep the weights of the pre-trained encoder frozen, but tune the weights of the head layers:

```
for param in model.base_model.parameters():  # these parameters will be frozen
    param.requires_grad = False
```

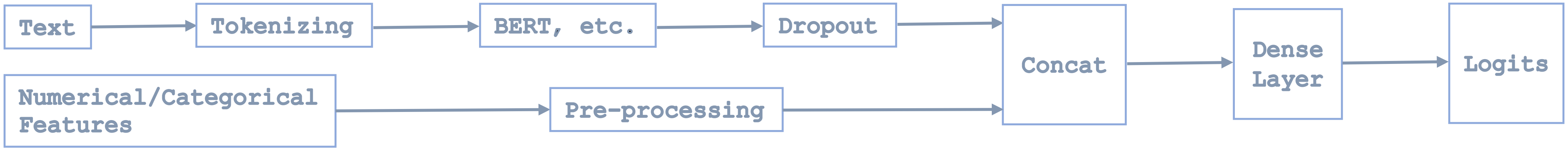
b) If we want to keep the weights of the pre-trained encoder frozen, but feed the model's output for the '[CLS]' token to a Logistic Regression or XGBoost:

```
model.eval()  # for Batchnorm and Dropout to work properly
…
with torch.no_grad():  # no gradient (of the layers within the context manager) will be calculated or stored
    outputs = model(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
    cls_token_embeddings = outputs.last_hidden_state[:, 0, :].numpy() # output: [batch size, hidden size]
…
```




- https://huggingface.co/transformers/v4.5.1/training.html#freezing-the-encoder
- http://jalammar.github.io/a-visual-guide-to-using-bert-for-the-first-time/
- https://github.com/bentrevett/pytorch-sentiment-analysis/blob/master/6%20-%20Transformers%20for%20Sentiment%20Analysis.ipynb
- https://stackoverflow.com/questions/63785319
- https://discuss.huggingface.co/t/bertmodel-forward-output-caveat-removed/983
- https://stackoverflow.com/questions/51748138

# How to combine NN outputs with numerical and categorical features




```
class CustomModel(torch.nn.Module):
    def __init__(self, model_name, num_extra_dims, num_labels, dropout=0.1):
        # num_extra_dims corresponds to the number of extra dimensions of numerical/categorical data
        super().__init__()
        self.config = AutoConfig.from_pretrained(model_name)
        self.transformer = AutoModel.from_pretrained(model_name, config=self.config)
        self.dropout = torch.nn.Dropout(dropout)
        self.classifier = torch.nn.Linear(self.transformer.config.hidden_size + num_extra_dims, num_labels) # usually 768 or 1024
    def forward(self, input_ids, extra_data, attention_mask=None, token_type_ids=None):
        # extra_data should be of shape [batch_size, dim] where dim is the # of additional numerical/categorical dimensions
        outputs = self.transformer(input_ids, attention_mask=attention_mask, token_type_ids=token_type_ids)
        cls_token_embeddings = outputs.last_hidden_state[:, 0, :] # output: [batch size, hidden size]
        cls_token_embeddings = self.dropout(cls_token_embeddings) # output: [batch size, hidden size]
        concat = torch.cat((cls_token_embeddings, extra_data), dim=-1) # output: [batch size, hidden size + num extra dims]
        logits = self.classifier(concat) # output: [batch size, num labels]
        return logits
```

https://colab.research.google.com/drive/1ZLfcB16Et9U2V-udrw8zwrfChFCIhomz
https://towardsdatascience.com/how-to-combine-textual-and-numerical-features-for-machine-learning-in-python-dc1526ca94d9
https://github.com/google-research/bert/issues/201
https://mccormickml.com/2021/06/29/combining-categorical-numerical-features-with-bert/

# Section 3: PyTorch Transformer Question Answering

# Reference

## - Article

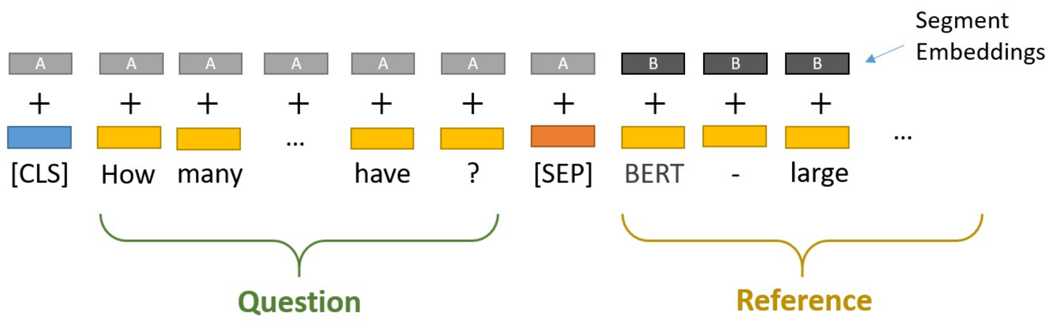
- https://mccormickml.com/2020/03/10/question-answering-with-a-fine-tuned-BERT/
- https://towardsdatascience.com/how-to-fine-tune-a-q-a-transformer-86f91ec92997 and https://gist.github.com/jamescalam/55daf50c8da9eb3a7c18de058bc139a3 (The use of DistilBertForQuestionAnswering and tokenizer's __call__. The training dataset here is the standard SQuAD dataset)
- https://towardsdatascience.com/which-flavor-of-bert-should-you-use-for-your-qa-task-6d6a0897fb24 (Less important)
- https://mccormickml.com/2021/05/27/question-answering-system-tf-idf/

## - Kaggle Notebook

- https://colab.research.google.com/github/ga642381/ML2021-Spring/blob/main/HW07/HW07.ipynb (The use of BertForQuestionAnswering and tokenizer's __call__. Also search for National Taiwan University Machine Learning course website and slides)
- https://www.kaggle.com/kanruwang/understanding-question-answering-sliding-window
- https://www.kaggle.com/davlanigan/bert-pytorch-qa2 (The use of BertForQuestionAnswering and tokenizer's encoder_plus)
- https://www.kaggle.com/zhuflower/du-reader-train (The use of BertForQuestionAnswering)
- https://www.kaggle.com/kanruwang/tweet-sentiment-roberta-pytorch-train and https://www.kaggle.com/kanruwang/tweet-sentiment-roberta-pytorch-inference and https://www.kaggle.com/kanruwang/tweet-sentiment-roberta-pytorch-understanding
- https://www.kaggle.com/kanruwang/bert-base-uncased-using-pytorch
- This task is uncommon, but the training set-up is useful: https://www.kaggle.com/takamichitoda/26th-place-solution and https://github.com/trtd56/KaggleQuest/blob/master/work/Quest_BERT-train.ipynb

## - Huggingface Documentation

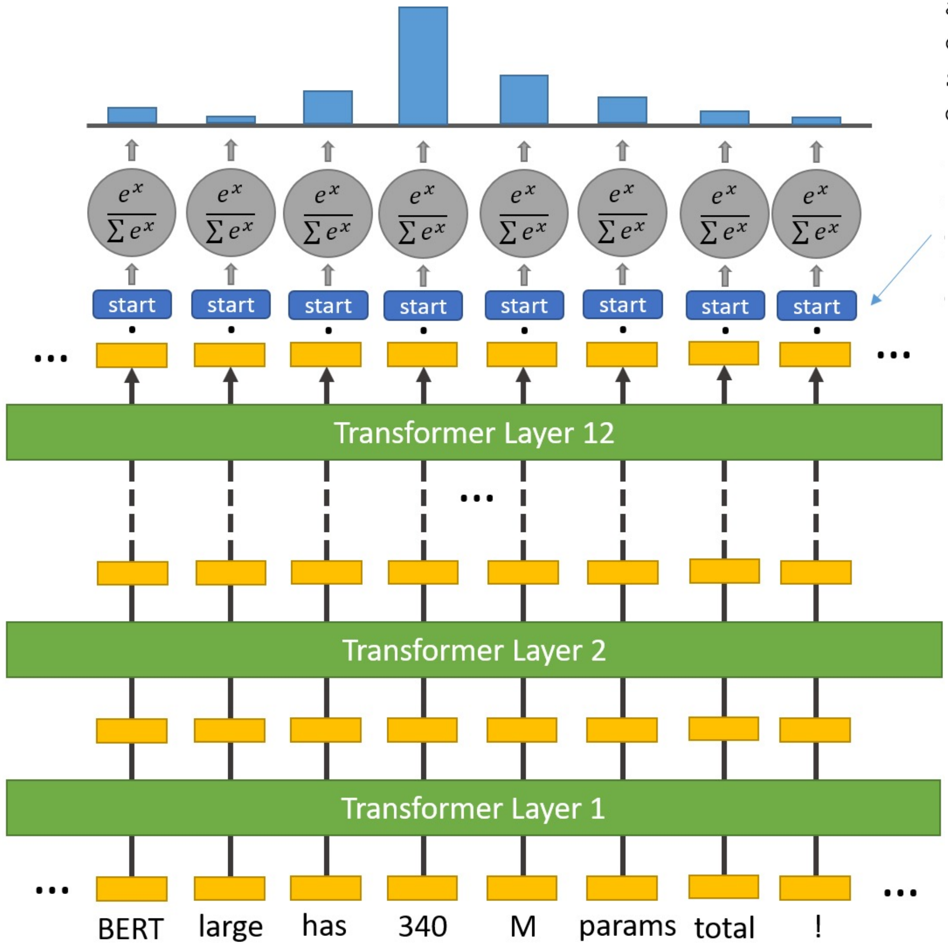
- https://github.com/huggingface/transformers/blob/master/src/transformers/models/bert/modeling_bert.py and https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertForQuestionAnswering
- https://github.com/huggingface/transformers/blob/master/src/transformers/tokenization_utils_base.py
- https://github.com/huggingface/notebooks/blob/master/examples/question_answering.ipynb (Example notebook, the use of Trainer class, and dividing a long context paragraphs into several paragraphs shorter than the max length)
- https://huggingface.co/course/chapter7/7?fw=pt

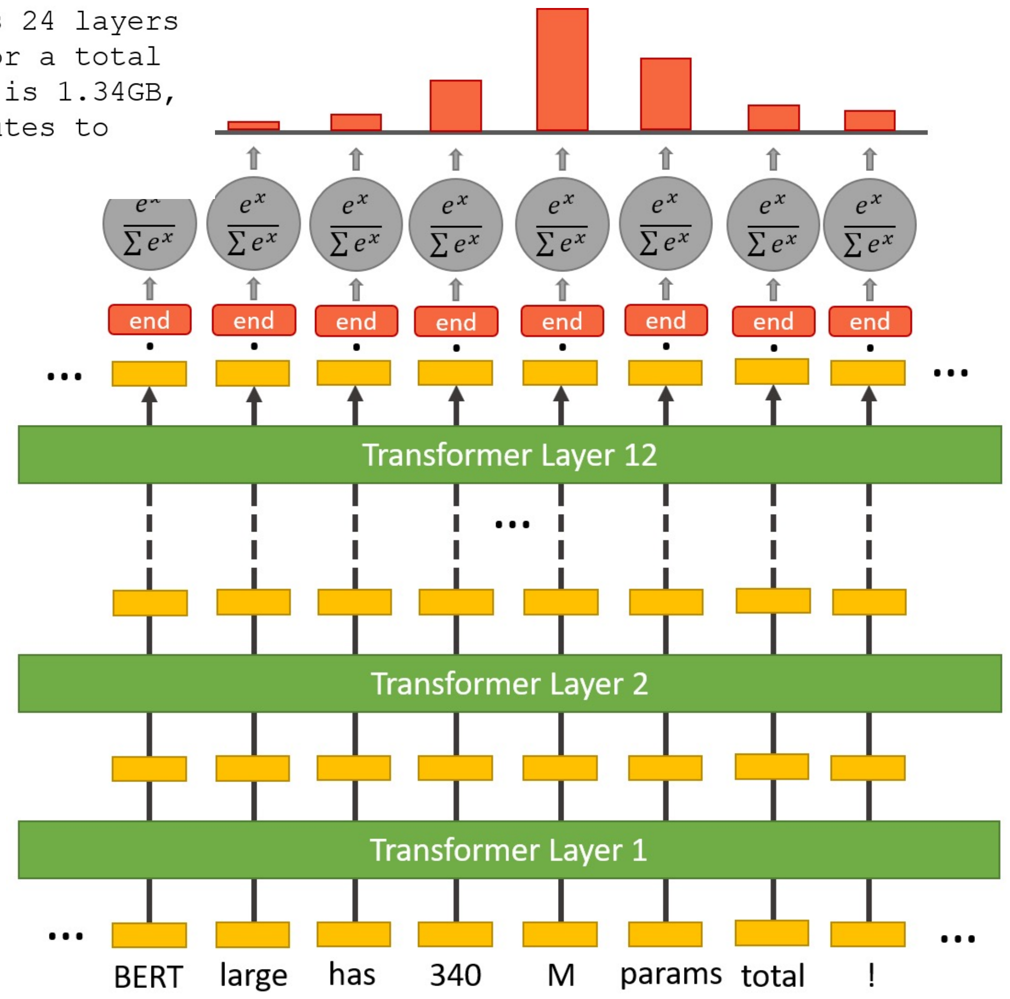



https://mccormickml.com/2020/03/10/question-answering-with-a-fine-tuned-BERT/

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

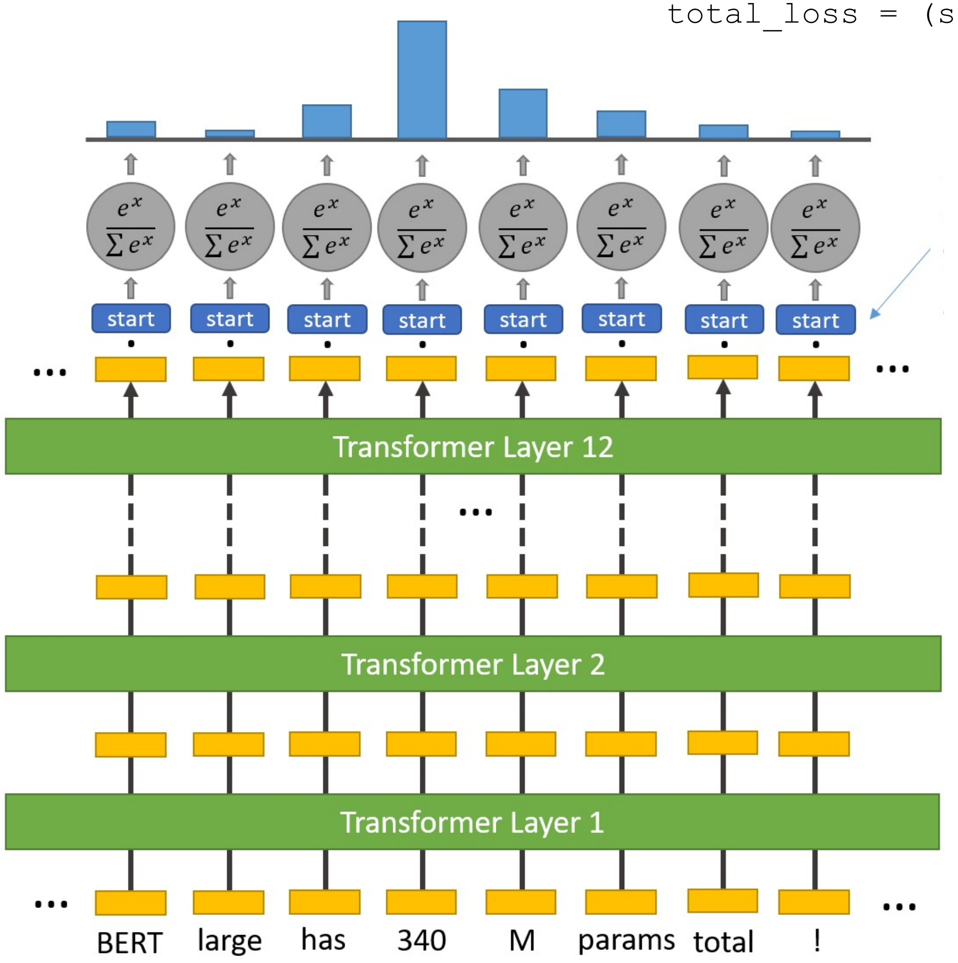
**Reference Text:** `BERT-large is really big... it has 24 layers and an embedding size of 1,024, for a total of 340M parameters! Altogether it is 1.34GB, so expect it to take a couple minutes to download to your Colab instance.`

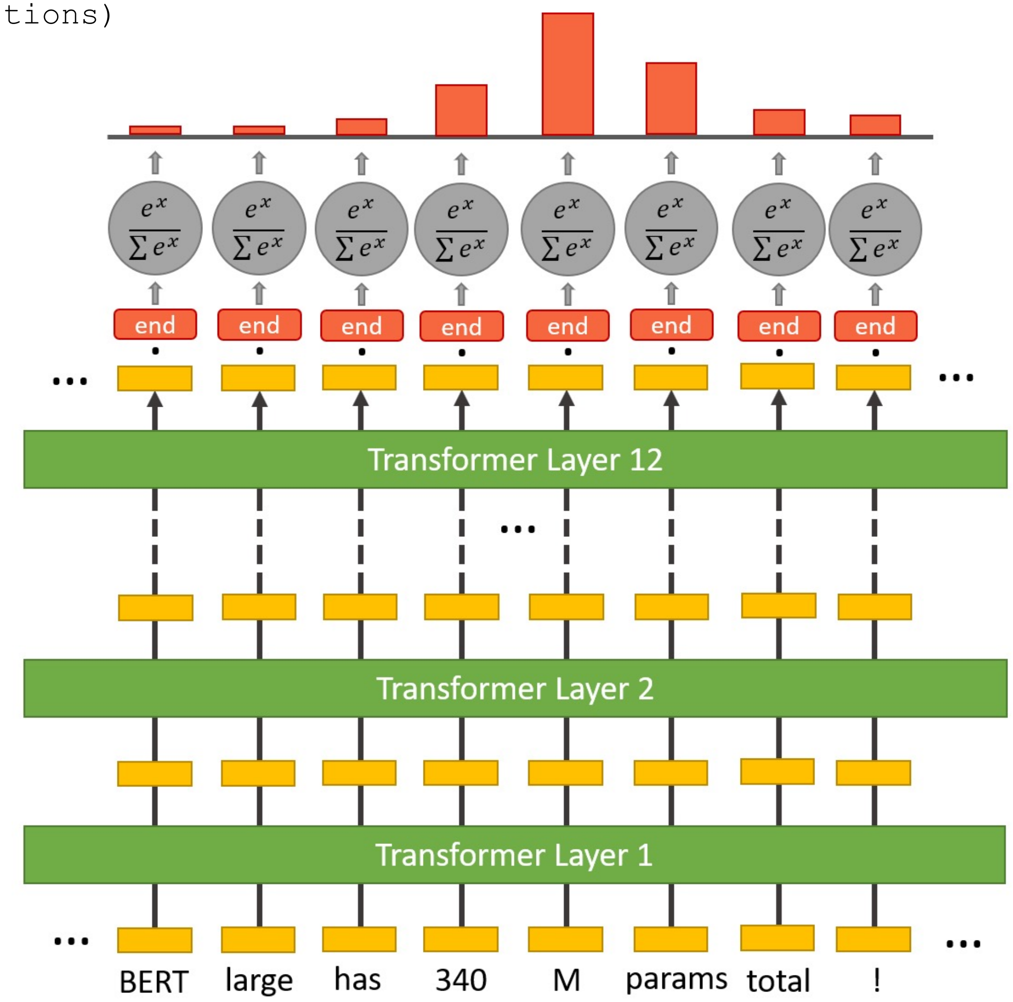


Just like a multi-class classification, here for the start position, we have
(1) a prediction vector coming out of a Softmax activation function for the start position
(2) a one-hot encoded ground truth vector for the start position
We can then calculate the Cross Entropy loss for the start position.
The same can be calculated for the end position.
Total loss is the mean of two Cross Entropy losses.

```
loss_function = CrossEntropyLoss()
start_loss = loss_function(start_logits, start_positions)
end_loss = loss_function(end_logits, end_positions)
total_loss = (start_loss + end_loss) / 2
```



Train Dataloader

All Train Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Val Dataloader

All Val Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Epoch

Train

Val

Test Dataloader

All Test Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs

Inference

```
from transformers import AutoTokenizer


tokenizer = AutoTokenizer.from_pretrained('bert-base-cased')
```

See tokenizer arguments https://huggingface.co/docs/transformers/internal/tokenization_utils

tokenizer(list of question strings, list of context strings) or

tokenizer(list of context strings, list of question strings), either order is okay.

```
train_encodings = tokenizer(train_questions, train_contexts, truncation=True, padding=True, add_special_tokens=True)
val_encodings = tokenizer(val_questions, val_contexts, truncation=True, padding=True, add_special_tokens=True)
test_encodings = tokenizer(test_questions, test_contexts, truncation=True, padding=True, add_special_tokens=True)
```

- The train_encodings here contains the input_ids, token_type_ids, attention_mask of each sample
- Sometimes there is no start_positions and end_positions, but as long as we have the answer text, we can find start_positions and end_positions from offset_mapping. See this and this.

| | |
|---|---|
| [CLS] | 101 |
| who | 2,040 |
| is | 2,003 |
| the | 1,996 |
| ac | 9,353 |
| ##as | 3,022 |
| director | 2,472 |
| ? | 1,029 |
| [SEP] | 102 |
| counter | 4,675 |
| ##feit | 21,156 |
| goods | 5,350 |

| | |
|---|---|
| problem | 3,291 |
| . | 1,012 |
| jem | 24,193 |
| ##ima | 9,581 |
| mw | 12,464 |
| ##af | 10,354 |
| ##ig | 8,004 |
| ##u | 2,226 |
| is | 2,003 |
| a | 1,037 |
| 34 | 4,090 |
| – | 1,011 |

| | |
|---|---|
| to | 2,000 |
| be | 2,022 |
| more | 2,062 |
| than | 2,084 |
| $ | 1,002 |
| 1 | 1,015 |
| . | 1,012 |
| 7 | 1,021 |
| trillion | 23,458 |
| this | 2,023 |
| year | 2,095 |
| . | 1,012 |
| [SEP] | 102 |

https://towardsdatascience.com/question-answering-with-a-fine-tuned-bert-bc4dafd45626

```
tokenizer.decode(train_encodings['input_ids'][0])
```

```
'[CLS] beyonce giselle knowles - carter ( / biːˈjɒnseɪ / bee - yon - say )
( born september 4, 1981 ) is an american singer, songwriter, record produc
er and actress. born and raised in houston, texas, she performed in various
singing and dancing competitions as a child, and rose to fame in the late 1
990s as lead singer of r & b girl - group destiny\'s child. managed by her
father, mathew knowles, the group became one of the world\'s best - selling
girl groups of all time. their hiatus saw the release of beyonce\'s debut a
lbum, dangerously in love ( 2003 ), which established her as a solo artist
worldwide, earned five grammy awards and featured the billboard hot 100 num
ber - one singles " crazy in love " and " baby boy ". [SEP] when did beyonc
e start becoming popular? [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PA
```

https://towardsdatascience.com/how-to-fine-tune-a-q-a-transformer-86f91ec92997

Train Dataloader

All Train Question Context pairs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Val Dataloader

All Val Question Context pairs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Epoch

Train

Val

Test Dataloader

All Test Question Context pairs, each has:

- Input IDs
- Attention mask
- Token type IDs

Inference

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

class QADataset(Dataset)
    """
    Instantiate the class with a tokenizer’s output object as an instance variable.
    Alternatively, can process data (e.g. tokenize and convert tokens to ids) using a QADataset object (process one
    sample each time when __getitem__ is called, or process all samples when a QADataset object is instantiated).
    """
    __init__
    __len__
    __getitem__(self, idx)
        # For Train and Val Dataset, need to return the
        # input_ids, attention_masks, token_type_ids, start_positions, end_positions of each sample.
        # For Test Dataset, need to return the
        # input_ids, attention_masks, token_type_ids of each sample.
        # Notice that token_type_ids is required for BERT.

train_set = QADataset(train_encodings)
val_set = QADataset(val_encodings)
test_set = QADataset(test_encodings)

train_loader = DataLoader(train_set, batch_size=train_batch_size, shuffle=True, pin_memory=True)
val_loader = DataLoader(val_set, batch_size=1, shuffle=False, pin_memory=True)
test_loader = DataLoader(test_set, batch_size=1, shuffle=False, pin_memory=True)
```

Train Dataloader

All Train Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Val Dataloader

All Val Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Epoch

Train

Val

Test Dataloader

All Test Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs

Inference

```
from transformer import AutoModelForQuestionAnswering, AdamW
from accelerate import Accelerator
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
if fp16_training:  # Automatic Mixed Precision offer about 1.5-3.0x speed up
    accelerator = Accelerator(fp16=True)
    device = accelerator.device
model = AutoModelForQuestionAnswering.from_pretrained('bert-base-uncased').to(device)
According to this post, the from_pretrained() in below does the following:
-  find the correct base model class to initialize
-  use init_weights() so that layers that are not pretrained (e.g. final classification layer) still get initialized
-  find the file with the pretrained weights
-  overwrite the weights of the model with the pretrained weights where applicable
Below is the source code of BertForQuestionAnswering
    class BertForQuestionAnswering(BertPreTrainedModel):
        def __init__(self, config):
            super().__init__(config)
            self.num_labels = config.num_labels
            self.bert = BertModel(config, add_pooling_layer=False)
            self.qa_outputs = nn.Linear(config.hidden_size, config.num_labels) # train this layer and fine tune the model
            self.post_init() # post_init() calls init_weights() which initializes weights
        def forward
            …
optimizer = AdamW(model.parameters(), lr=learning_rate)
if fp16_training:
    model, optimizer, train_loader = accelerator.prepare(model, optimizer, train_loader)
```

Train Dataloader

All Train Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Val Dataloader

All Val Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Start position
- End position

Epoch

Train

Val

Test Dataloader

All Test Question Context pairs, each has:
- Input IDs
- Attention mask
- Token type IDs

Inference

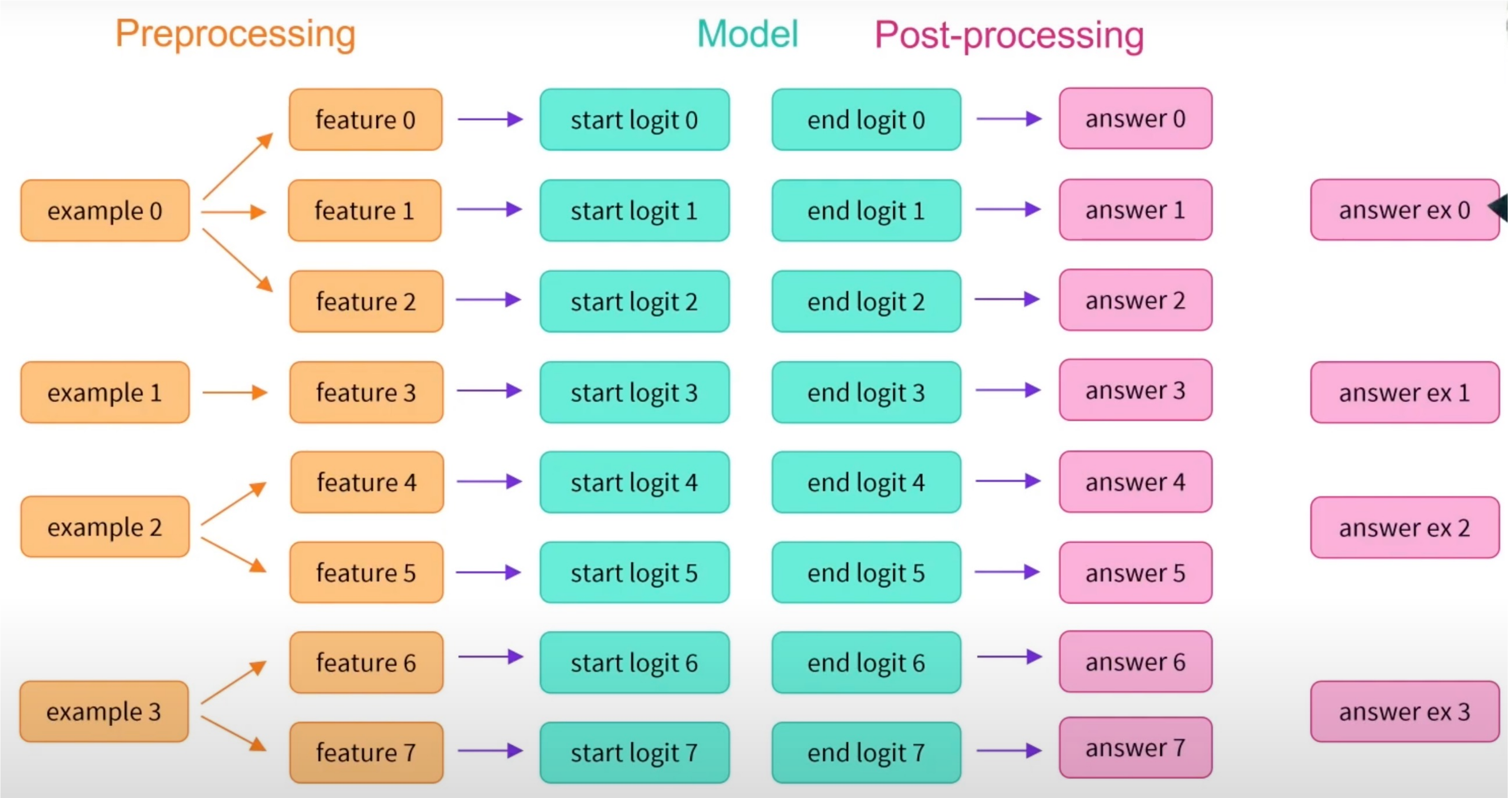
```
for epoch in range(num_epoch):
    model.train()
    for batch in tqdm(train_loader):
        optimizer.zero_grad()
        batch = {k: v.to(device) for k, v in batch.items()}  # all values to(device)
        # model args: input_ids (mandatory), token_type_ids, attention_mask, start_positions, end_positions
        outputs = model(**batch)
        # model outputs: start_logits, end_logits, loss (loss returned when start_positions/end_positions are provided)
        start_index = torch.argmax(output['start_logits'], dim=1)  # choose the most probable start position
        end_index = torch.argmax(output['end_logits'], dim=1)  # choose the most probable end position
        # prediction is correct only if both start_index and end_index are correct
        train_acc_ls.append(((start_index == batch['start_positions']) & (end_index == batch['end_positions'])).float().mean())
        train_loss_ls.append(output['loss'])
        accelerator.backward(output['loss']) if fp16_training else output['loss'].backward()
        optimizer.step()
    model.eval()
    with torch.no_grad():
        for batch in tqdm(val_loader):
            batch = {k: v.to(device) for k, v in batch.items()}
            outputs = model(**batch)
            start_index = torch.argmax(outputs['start_logits'], dim=1)
            end_index = torch.argmax(outputs['end_logits'], dim=1)
            … calculate accuracy and append to list …
```

https://www.kaggle.com/kanruwang/two-ways-to-prepare-pytorch-dataset-for-bert

Inference code is similar to validation code, but on test_loader …

```
model.eval()
with torch.no_grad():
    for batch in tqdm(test_loader):
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        start_index = torch.argmax(outputs['start_logits'], dim=1)
        end_index = torch.argmax(outputs['end_logits'], dim=1)
        tokens = tokenizer.convert_ids_to_tokens(batch['input_ids'].to(device))
        # start with the first token
        answer = tokens[start_index]
        # select the remaining answer tokens and join them with whitespace
        for i in range(start_index + 1, end_index + 1):
            # if it's a subword token, then recombine it with the previous token
            if tokens[i][0:2] == '##':
                answer += tokens[i][2:]
            # otherwise, add a space then the token
            else:
                answer += ' ' + tokens[i]
        print(answer.capitalize())
```

- Sometimes the context text is too long for the model, so a sliding window is applied to the text, cutting the text into several paragraphs, each paragraph is called a "feature".
- To avoid cutting amid an answer text, each "feature" (sub-paragraph) has some overlap with its previous and subsequent "feature".
- During training preprocessing, if a "feature" does not contain the answer or only contains part of the answer, the start_positions and end_positions are both assigned 0.




https://www.kaggle.com/kanruwang/understanding-question-answering-sliding-window

https://huggingface.co/course/chapter7/7?fw=pt

# (Bonus) Huggingface Trainer API

Reference

- Notebook
  - https://www.kaggle.com/kanruwang/huggingface-trainer-api
  - https://colab.research.google.com/drive/1-JIJlao4dI-Ilww_NnTc0rxtp-ymgDgM (from https://huggingface.co/transformers/v4.5.1/training.html)
  - https://colab.research.google.com/github/huggingface/notebooks/blob/master/examples/text_classification.ipynb
  - https://colab.research.google.com/github/huggingface/blog/blob/master/notebooks/trainer/01_text_classification.ipynb
  - https://github.com/huggingface/notebooks/blob/master/examples/question_answering.ipynb
- Huggingface Documentation
  - https://huggingface.co/docs/transformers/custom_datasets#question-answering-with-squad

**Train Dataset**

All Train text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Label

**Val Dataset**

All Val text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs
- Label

**Epoch**

Train

Val

**Test Dataset**

All Test text paragraphs, each has:
- Input IDs
- Attention mask
- Token type IDs

**Inference**

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification, TrainingArguments, Trainer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

train_texts, val_texts, train_labels, val_labels = train_test_split(texts, labels, test_size=.2)

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)

class Dataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels=None):
        self.encodings = encodings
        self.labels = labels
    def __getitem__(self, idx):
        if self.labels:
            return {**{key: torch.tensor(val[idx]) for key, val in self.encodings.items()}, "labels": self.labels[idx]}
        else:
            return {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
    def __len__(self):
        return len(self.encodings.input_ids)
train_dataset = Dataset(train_encodings, train_labels)
eval_dataset = Dataset(eval_encodings, eval_labels)
test_dataset = Dataset(test_encodings)

model = AutoModelForSequenceClassification.from_pretrained("bert-base-cased", num_labels=2)
```

https://www.kaggle.com/kanruwang/huggingface-trainer-api

**Train Dataset**

All Train text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

**Val Dataset**

All Val text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs
- Label

**Epoch**

- Train
- Val

**Test Dataset**

All Test text paragraphs, each has:

- Input IDs
- Attention mask
- Token type IDs

**Inference**

```
training_args = TrainingArguments(
    output_dir='./results',          # output directory
    evaluation_strategy='epoch',     # evaluation is done each epoch
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    gradient_accumulation_steps=1,   # defaults to 1
    learning_rate=2e-5,              # initial learning rate for AdamW optimizer
    weight_decay=0.01,               # defaults to 0
    num_train_epochs=3,              # defaults to 3
    lr_scheduler_type='linear',      # defaults to 'linear'
    warmup_steps=500,                # num of linear warmup steps from 0 to learning_rate for learning rate scheduler
    logging_dir='./logs',            # directory for storing logs
    logging_strategy='steps',        # defaults to 'steps'
    logging_steps=10,                # num of update steps between two logs
    fp16=True,                       # whether use mixed precision training, defaults to False
    dataloader_num_workers=4,        # defaults to 0
    label_smoothing_factor=0,        # defaults to 0
    optim='adamw_hf',                # defaults to 'adamw_hf'
    dataloader_pin_memory=True       # defaults to True
)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics,
)
trainer.train()
trainer.evaluate()
```

https://www.kaggle.com/kanruwang/huggingface-trainer-api