UE 803 - Data Science for NLP

Lecture 16: Hugging Face Transformers Library

Claire Gardent - CNRS / LORIA

Neural Classification

Outline

- Classifying with BERT
- HF Transformers library
- Classifying with the HF Pipelines
- Classifying using a Pretrained Model
- Fine-tuning a Pretrained HF Classification Model

BERT Classifier

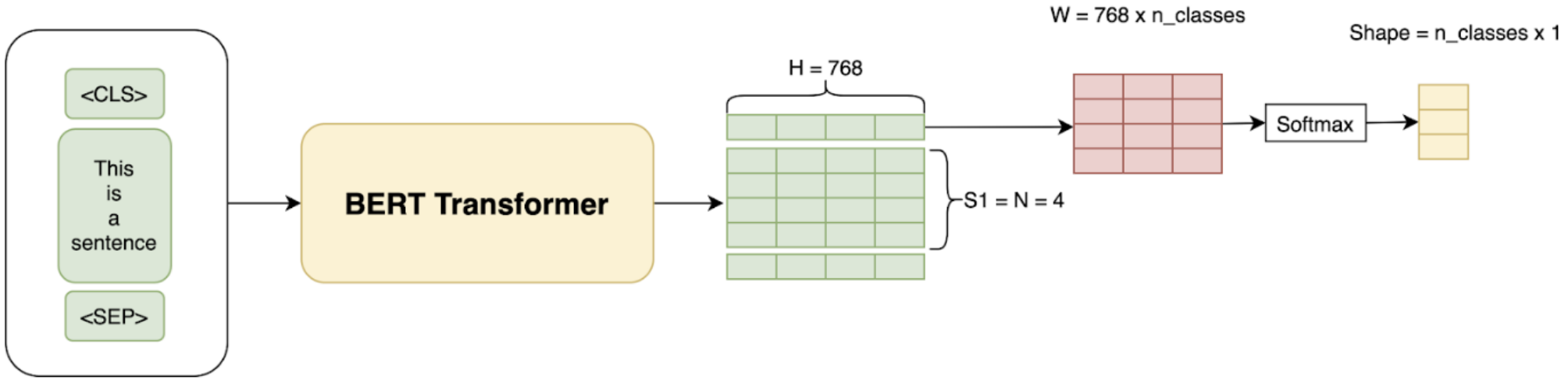
BERT

- A Transformer Neural Network
- Trained on large amount of text to predict masked tokens (and next sentence fit)
- ***An encoder which maps sequences of input tokens to sequences of vector representations (embeddings)***
- The first (CLS) token is often taken to encode the whole input sequence

BERT Classifier

- BERT + Classification Layer
- Fine-tuned on classification data
- ***Maps the encoding of the CLS token to a probability distribution over the set of target classes***

BERT Classifier



The 🤗 Transformers library

The 🤗 Transformers library

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

Provides the functionality to create and use thousands of pretrained models shared by companies and individual researchers

📝 Text: text classification, information extraction, question answering, summarization, translation, and text generation in over 100 languages.

🖼️ Images: image classification, object detection, and segmentation.

🗣️ Audio: speech recognition and audio classification.

🐙 Multimodal: table question answering, optical character recognition, information extraction from scanned documents, video classification, and visual question answering.

Three ways to use a model from the 🤗 Transformers library

- Pipeline, use an existing model as is
  Black-box use
- Use a pre-trained model
  The behaviour of the model at various intermediate stages (tokenization, padding, truncation etc.) can be inspected.
- Fine tune a pre-trained model on some data
  Re-train a pre-trained model in a supervised way on a given task.

Pipeline

Pipeline

- Black box use of an existing mode. The `pipeline()` method automatically loads a default model and tokenizer which can perform your task.
- The pipeline() function connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer
- In the example below, "sentiment-analysis" is a binary classification model.

```
from transformers import pipeline

classifier = pipeline("sentiment-analysis")
classifier("This film is great.")
```

```
[{'label': 'POSITIVE', 'score': 0.9598047137260437}]
```

Some Pipelines

Pipelines

- Audio Classification
- Automatic Speech Recognition
- Image Classification
- Image Segmentation
- Object Detection
- Question Answering
- Summarization
- Text Classification
- Text Generation
- Text2Text Generation
- Token Classification
- Translation

Using Pre-Trained Models

Loading pretrained instances with an AutoClass

- An `AutoClass` automatically infer and load the correct architecture from a given checkpoint (set of weights).
- The `from_pretrained` method lets you quickly load a pretrained model for a given architecture.

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassific

# Choose a pretrained model (checkpoint, model weights)
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

# Select the corresponding tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
# Select the corresponding model
model = AutoModelForSequenceClassification.from_pretrained(checkpoint
```

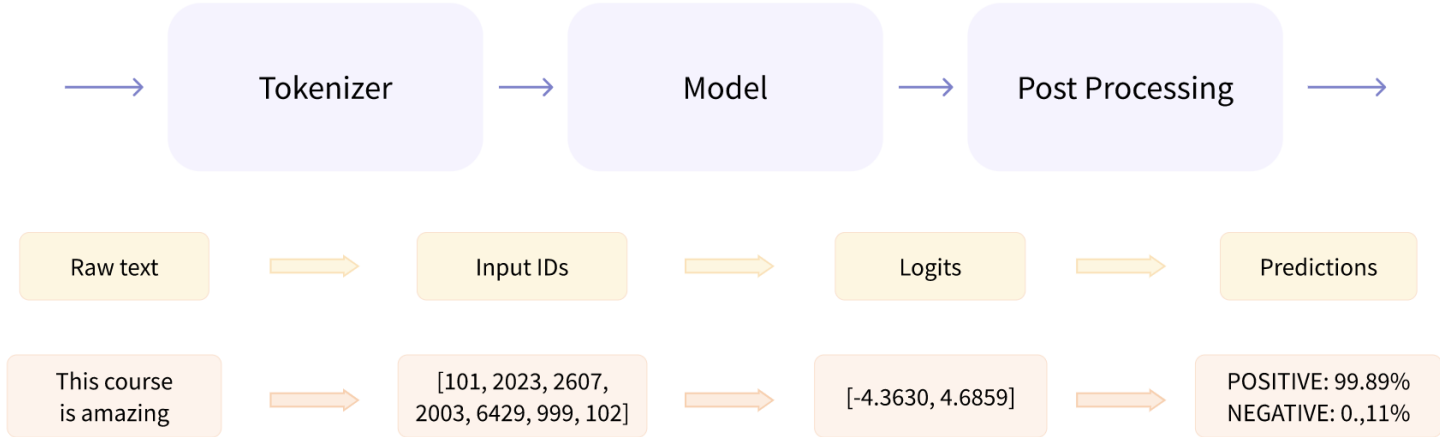
This creates a tokenizer and a sequence classification model initialised with (distil) BERT weights

Using a Pre-Trained Model

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassific
# Choose a pretrained model (checkpoint)
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
# Select the corresponding tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
# Select the corresponding model
model = AutoModelForSequenceClassification.from_pretrained(checkpoint
# Input
sequences = ["I like Yoga.", "I also like walking." ]
# Tokenize the input
tokens = tokenizer(sequences, padding=True, return_tensors="pt")
# Run the model on the tokenized input
output = model(**tokens)
```

We use the pre-trained model "as is": we do not train it

Neural Processing



Three main steps

- Tokenize (Encode)
- Apply the model to the tokenized data
- Map the model output (logits) to a prediction (predicted class)

Tokenizing (Encoding)

The preprocessing needs to be done in exactly the same way as when the model was pretrained

- Use the checkpoint name of our model
- Use the `AutoTokenizer` class and its `from_pretrained()` method
  The `AutoTokenizer` class will grab the proper `tokenizer` class in the library based on the checkpoint name

```
import torch
from transformers import AutoTokenizer, AutoModelForSequenceClassific

# Choose a pretrained model (checkpoint)
checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

# Select the corresponding tokenizer
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# Select the corresponding model
model = AutoModelForSequenceClassification.from_pretrained(checkpoint
```

What does the Tokenizer do ?

Tokenizing

- Converts tokens to integers (indices)
- Maps inputs of different length to vectors of the same length
    - Padding, Truncation, Attention mask
    - Creating batches, dynamic padding
- Outputs a tensor which can be given as input to the corresponding pre-trained model

A tokenizer converts your input into a format that can be processed by the model

Converting Tokens to Integers

Models cannot process raw text directly, so the first step is to convert the text inputs into numbers

- Split the input into words or subwords that are called tokens
- Map each token to an integer

From Tokens to Integers and back

```
from transformers import AutoTokenizer
# Create a tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

# Input sentence and Tokenize
sequence = "Using a Transformer network is simple"
tokens = tokenizer.tokenize(sequence)
>> ['Using', 'a', 'transform', '##er', 'network', 'is', 'simple']

# Map the tokens to integers (indices)
ids = tokenizer.convert_tokens_to_ids(tokens)
>> [7993, 170, 11303, 1200, 2443, 1110, 3014]

# Decoding : Integers -> Tokens
decoded_string = tokenizer.decode(ids)
>> 'Using a Transformer network is simple'
```

Padding and Masking

The input to a Transformer model is a ***batch*** where multiple sentences are encoded into vectors of the same length

The tokenizer maps tokens to integer but also ***pads the input*** and creates an ***attention mask***

This ensures that although input sentences are of various length, their vector representations are all of the same length

- Sentences are converted to lists of identifiers
- ***Padding*** (adding the padding token ID) ensures that all identifiers lists are the *same length* .
  N.B. padding all the samples to the maximum length is not efficient: it's better to pad the samples when building a batch (dynamic padding)
- The ***attention mask*** tells the attention layers to *ignore the padding tokens*
  Attention masks have the exact same shape as the input IDs vectors filled with 0s and 1s: 1s indicate the corresponding tokens should be attended to, and 0s indicate the corresponding tokens should not be attended to

Padding and Masking

```
# Input
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!"]

# tokenize
tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="

# Output
{    'input_ids': tensor([
      [  101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662,
         2607,  2026,  2878,  2166,  1012,   102],
      [  101,  1045,  5223,  2023,  2061,  2172,   999,   102,     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, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0] ])
}
```

Padding

```
# Will pad the sequences up to the maximum sequence length
model_inputs = tokenizer(sequences, padding="longest")

# Will pad the sequences up to the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, padding="max_length")

# Will pad the sequences up to the specified max length
model_inputs = tokenizer(sequences, padding="max_length", max_length=
```

Truncation

If there are length limits (from the model or specified), sentences must be truncated to respect these limits.

```
sequences = ["I've been waiting for a HuggingFace course my whole lif

# Truncate sequences that are longer than the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, truncation=True)

# Truncate sequences that are longer than the specified max length
model_inputs = tokenizer(sequences, max_length=8, truncation=True)
```

Special Tokens

Some models add special tokens to the input (here: [CLS] and [SEP]). The tokenizer knows which ones are expected and will deal with this for you.

Example: One token ID is added at the beginning, and one at the end. Using the decode method, we can see which tokens these are.

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
sequence = "I like yoga."

# Tokenize
tokens = tokenizer(sequence)

# Decode
tokenizer.decode(tokens["input_ids"])
>>
"[CLS] I like yoga. [SEP]"
```

Tokenizer Output

Given some inputs, the tokenizer generally outputs a tensor with three dimensions:

- Batch size: The number of inputs processed at a time (2 in our example).
- Sequence length: The (fixed) length of the input numerical representation (16 in our example).
- Hidden size: The size of the vectors representing each token (***embeddings***)

```
outputs = model(**inputs)
# Sequence of hidden-states at the output of the last layer.
print(outputs.last_hidden_state.shape)
# 2 inputs, 16 dimension vector per instance, embedding size: 768
torch.Size([2, 16, 768])
```

Model Output (Binary Classifier)

Transformer models usually output ***logits*** i.e., scores.
This can be transformed into a probability distribution using a softmax layer

```
outputs = model(**inputs)
# Logits: the raw scores output by the last layer of the model
print(outputs.logits)
>> tensor([[-1.5607,  1.6123], [ 4.1692, -3.3464]] )

# From logits to Probabilities
import torch
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
>> tensor([[4.0195e-02, 9.5980e-01], [9.9946e-01, 5.4418e-04]] )

# To know which position corresponds to which label
model.config.id2label
>> {0: 'NEGATIVE', 1: 'POSITIVE'}
```

Using a Dataset from the 🤗 Transformers library

Loading a dataset

https://huggingface.co/datasets

Download and cache the dataset, by default in ~/.cache/huggingface/dataset

```
from datasets import load_dataset
raw_datasets = load_dataset("mrpc")
>> DatasetDict({
       train: Dataset({
          features: ['sentence1', 'sentence2', 'label', 'idx'],
          num_rows: 3668
       })
       validation: Dataset({
          features: ['sentence1', 'sentence2', 'label', 'idx'],
          num_rows: 408
       })
       test: Dataset({
          features: ['sentence1', 'sentence2', 'label', 'idx'],
          num_rows: 1725
       })
```

Looking at the Data

```
raw_train_dataset = raw_datasets["train"][0]

{'idx': 0,
 'label': 1,
 'sentence1': 'Amrozi accused his brother , whom he called " the witn
 'sentence2': 'Referring to him as only " the witness " , Amrozi accu
```

Using the `features` attribute, we can inspect the structure of the dataset.

```
raw_train_dataset.features
>>
{'sentence1': Value(dtype='string', id=None),
 'sentence2': Value(dtype='string', id=None),
 'label': ClassLabel(num_classes=2, names=['not_equivalent', 'equival
 'idx': Value(dtype='int32', id=None)}
```

Fine-Tuning a Pre-Trained Model

Fine-Tuning a Pre-Trained Model

- Load the data
- Tokenize
- Define the evaluation metrics
- Train and evaluate

Loading the data

MRPC Dataset

- 3,668 pairs of sentences in the training set, 408 in the validation set, and 1,725 in the test set
- Paraphrasing dataset: each sentence pairs is labelled as paraphrase vs. not paraphrase pair

```
from datasets import load_dataset
# Load the dataset
load_dataset("mrpc")

  >>>
DatasetDict({
    train: Dataset({
        features: ['idx', 'label', 'sentence1', 'sentence2'],
        num_rows: 3668
    })
 ...
})
```

Tokenizing pairs of sentences

To keep the data as a dataset, we can use the `Dataset.map()` method which applies a function to each element of the dataset.
The tokenizer tokenizes two sentences for each data instance.

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding
raw_datasets = load_dataset("mrpc")
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

# Function which tokenizes pairs of sentences
def tokenize_function(dataset):
    return tokenizer(dataset["sentence1"], dataset["sentence2"], trunc

# Tokenize the dataset
raw_datasets.map(tokenize_function, batched=True)
```

Dynamic Padding

- When using the default collate function, the size of the input representation is the same for all batches.
- ***Dynamic padding*** pads all samples to the length of the longest element in the batch. This avoids having inputs with a lot of padding and speeds up training.
- The `DataCollatorWithPadding` function can be used to apply dynamic padding

```
from transformers import DataCollatorWithPadding

data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Preprocessing Summary Code

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("mrpc")
checkpoint = "bert-base-uncased"

# Tokenize pairs of sentences
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], trun

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True

# Create batches using dynamic padding
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Training

Define a TrainingArguments class

Parameter: a directory where the trained model will be saved.

```
from transformers import TrainingArguments

training_args = TrainingArguments("saved-model")
```

Define a model

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint
```

Training

Define a Trainer

```
from transformers import Trainer

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
)
```

Train

Fine-tune the model (train on your data)

```
trainer.train()
```

Evaluation

To report the validation loss and metrics at the end of each epoch in addition to the training loss, update the trainer with `compute_metric` and `evaluation_strategy`

```
training_args = TrainingArguments("saved-model",
   evaluation_strategy="epoch")
model = AutoModelForSequenceClassification.from_pretrained(checkpoint

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
    compute_metrics=compute_metrics,
)
trainer.train()
```

Evaluation

Define a `compute_metrics()` function which (i) takes as input the correct labels and the predictions made by the model (the logits output for each element of the evaluation dataset) and (ii) applies a metric

```
def compute_metrics(eval_preds):
# Select a metric
  metric = load_metric("mrpc")
# Get the results and the expected labels
    logits, labels = eval_preds
# take the index with the maximum value on the second axis
    predictions = np.argmax(logits, axis=-1)
# apply the metric
    return metric.compute(predictions=predictions, references=labels)
```

Merci!