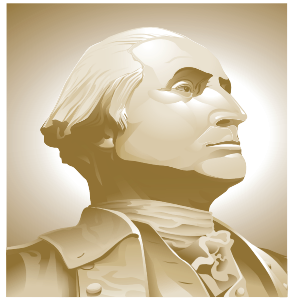
Transformer

Natural Language Processing
Lecture 9



THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

State of Art Machine Learning

Context

- The aim of NLP tasks is **not only to understand single words** individually, but to be able to **understand the context of those words**
  - **Classifying whole sentences**: Getting the sentiment of a review, detecting if an email is spam, determining if a sentence is grammatically correct or whether two sentences are logically related or not
  - **Classifying each word in a sentence:** Identifying the grammatical components of a sentence (noun, verb, adjective), or the named entities (person, location, organization)
  - **Generating text content**: Completing a prompt with auto-generated text, filling in the blanks in a text with masked words
  - **Extracting an answer from a text**: Given a question and a context, extracting the answer to the question based on the information provided in the context
  - **Generating a new sentence from an input text:** Translating a text into another language, summarizing a text

Why is it Challenging?

- Computers don't process information in the same way as humans.
- For example, when we read the sentence "I am hungry," we can easily understand its meaning.
- Similarly, given two sentences such as "I am hungry" and "I am sad," we're able to easily determine how similar they are.
- For machine learning (ML) models, such tasks are more difficult.

- The text needs to be processed in a way that enables the model to learn from it.
- And because language is complex, we need to think carefully about how this processing must be done.
- We want to solve these challenges by introducing new architecture **Transformer**.

Transformers Are Everywhere

- Transformer models are used to solve all kinds of NLP tasks, like the ones mentioned in the previous slide.
- The most basic object in the transformers library is pipeline.
- It connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer
- There are three main steps involved when you pass some text to a pipeline:

1- The text is preprocessed into a format the model can understand.

2- The preprocessed inputs are passed to the model.

3- The predictions of the model are post-processed, so you can make sense of them.

Some of the Currently Available Pipelines Are:

- **feature-extraction** (get the vector representation of a text)
- **fill-mask**
- **ner** (named entity recognition)
- **question-answering**
- **sentiment-analysis**
- **summarization**
- **text-generation**
- **translation**
- **zero-shot-classification**

Sentiment Analysis

- By default, this pipeline selects a particular pretrained model that has been fine-tuned for sentiment analysis in English. The model is downloaded and cached when you create the classifier object. If you rerun the command, the cached model will be used instead and there is no need to download the model again.

```
1 from transformers import pipeline
2 classifier = pipeline("sentiment-analysis")
3 print(classifier("I've been waiting to listen to this course my whole life."))
4 print(classifier("I've been waiting to listen to this terrible course my whole life."))
5 print(classifier("I am not sure what I am doing but seems I am doing a heck of job"))
6 print(classifier("How are you doing?.I am not too bad"))
```

Zero-shot classification

- We'll start by tackling a more challenging task where we need to classify texts that haven't been labelled. This is a common scenario in real-world projects because annotating text is usually time-consuming and requires domain expertise. For this use case, the zero-shot-classification pipeline is very powerful

```
1 | from transformers import pipeline
2 | classifier = pipeline("zero-shot-classification")
3 |
4 | print(classifier("This is a course about the Transformers library",
5 |                  candidate_labels=["education", "politics", "business"],))
6 |
7 | classifier("This session is about the machine learning and artifical inteligence",
8 |            candidate_labels=["education", "politics", "business", "data science"],)
```

- it allows you to specify which labels to use for the classification, so you don't have to rely on the labels of the pretrained model. You've already seen how the model can classify a sentence as positive or negative using those two labels — but it can also classify the text using any other set of labels you like.l

Text Generation

- Now let's see how to use a pipeline to generate some text. The main idea here is that you provide a prompt and the model will auto-complete it by generating the remaining text. This is similar to the predictive text feature that is found on many phones. Text generation involves randomness, so it's normal if you don't get the same results as shown below.

```
1 | from transformers import pipeline
2 |
3 | generator = pipeline("text-generation")
4 | print(generator("In this course, we will teach you how to"))
5 | print(generator("I am tired of listening to this brownbag session about natural language pro
6 |           num_return_sequences = 1, max_length  = 100 ))
```

Using any model from the Hub

- The previous examples used the default model for the task at hand, but you can also choose a particular model from the Hub to use in a pipeline for a specific task — say, text generation. Go to the Model Hub and click on the corresponding tag on the left to display only the supported models for that task.

```
1 | from transformers import pipeline
2 |
3 | generator = pipeline("text-generation", model="distilgpt2")
4 | print(generator("In this course, we will teach you how to"))
5 | print(generator("I am tired of listening to this brownbag session about natural language pro
6 |           num_return_sequences = 1, max_length  = 100 ))
```

Mask Filling

- The next pipeline you'll try is fill-mask. The idea of this task is to fill in the blanks in a given text: The top_k argument controls how many possibilities you want to be displayed. Note that here the model fills in the special $< mask >$ word, which is often referred to as a mask token. Other mask-filling models might have different mask tokens, so it's always good to verify the proper mask word when exploring other models.

```
1 | from transformers import pipeline
2 |
3 | unmasker = pipeline("fill-mask")
4 | print(unmasker("This session will teach you all about <mask> models.", top_k=2))
5 | print(unmasker("I <mask> my life.", top_k=5))
```

Named Entity Recognition

- Named entity recognition (NER) is a task where the model has to find which parts of the input text correspond to entities such as persons, locations, or organizations. Let's look at an example. We pass the option grouped_entities=True in the pipeline creation function to tell the pipeline to regroup together the parts of the sentence that correspond to the same entity.

```
1 from transformers import pipeline
2
3 ner = pipeline("ner", grouped_entities=True)
4 print(ner("My name is Amir and I work at GWU in District of Columbia office."))
```

Question Answering

- The question-answering pipeline answers questions using information from a given context:

```
from transformers import pipeline
question_answerer = pipeline("question-answering")
print(question_answerer(
    question="Where do I work?",
    context="My name is Amir and I work at CL in in District of Columbia office."))
print(question_answerer(
    #question="Where is the capital of France?",
    #question="What is population Paris?",
    question="What is GDP of Paris?",
    context="""
    Paris (French pronunciation: (About this soundlisten)) is the capital and most
    populous city of France, with an estimated population of 2,175,601 residents as
    of 2018,in an area of more than 105 square kilometres (41 square miles).[4] Since
    the 17th century, Paris has been one of Europe's major centres of finance,
    diplomacy, commerce, fashion, gastronomy, science, and arts. The City of Paris is
    the centre and seat of government of the region and province of le-de-France, or
    Paris Region, which has an estimated population of 12,174,880, or about 18 percent
    of the population of France as of 2017.[5] The Paris Region had a GDP of 709 billion
    (808 billion)in 2017.[6] According to the Economist Intelligence Unit Worldwide Cost
    of Living Survey in 2018, Paris was the second most expensive city in the world,
     after Singapore and ahead of Zurich, Hong Kong, Oslo, and Geneva.[7]"""
))
```

Summarization

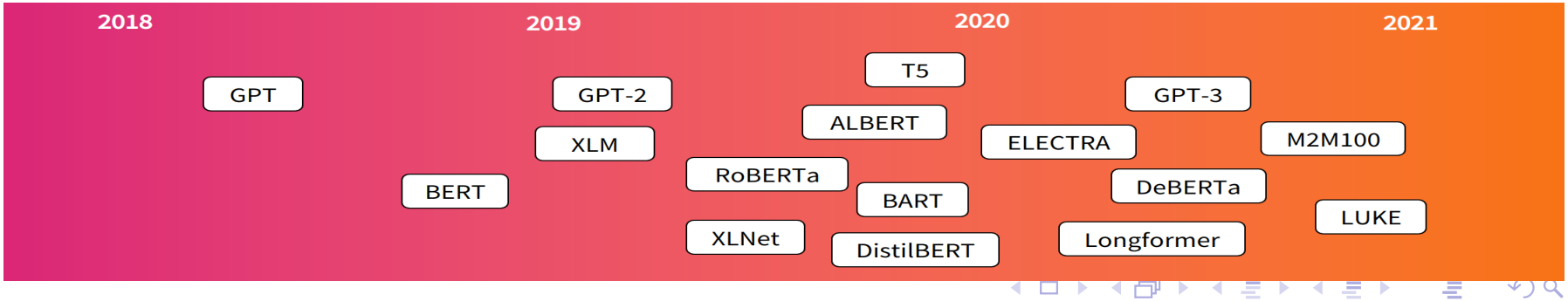
- Summarization is the task of reducing a text into a shorter text while keeping all (or most) of the important aspects referenced in the text. Here's an example:

```
from transformers import pipeline
summarizer = pipeline("summarization")
summarizer("""
    America has changed dramatically during recent years. Not only has the number of
    graduates in traditional engineering disciplines such as mechanical, civil,
    electrical, chemical, and aeronautical engineering declined, but in most of
    the premier American universities engineering curricula now concentrate on
    and encourage largely the study of engineering science. As a result, there
    are declining offerings in engineering subjects dealing with infrastructure,
    the environment, and related issues, and greater concentration on high
    technology subjects, largely supporting increasingly complex scientific
    developments. While the latter is important, it should not be at the expense
    of more traditional engineering.

    Rapidly developing economies such as China and India, as well as other
    industrial countries in Europe and Asia, continue to encourage and advance
    the teaching of engineering. Both China and India, respectively, graduate
    six and eight times as many traditional engineers as does the United States.
    Other industrial countries at minimum maintain their output, while America
    suffers an increasingly serious decline in the number of engineering graduates
    and a lack of well-educated engineers.""",max_length= 100)
```

A Bit of Transformer History

- Here are some reference points in the (short) history of Transformer models:
- The Transformer architecture was introduced in June 2017.
- The focus of the original research was on translation tasks. This was followed by the introduction of several influential models, including:
  - GPT-like (also called auto-regressive Transformer models)
  - BERT-like (also called auto-encoding Transformer models)
  - BART/T5-like (also called sequence-to-sequence Transformer models)



Transformers Are Language Models?

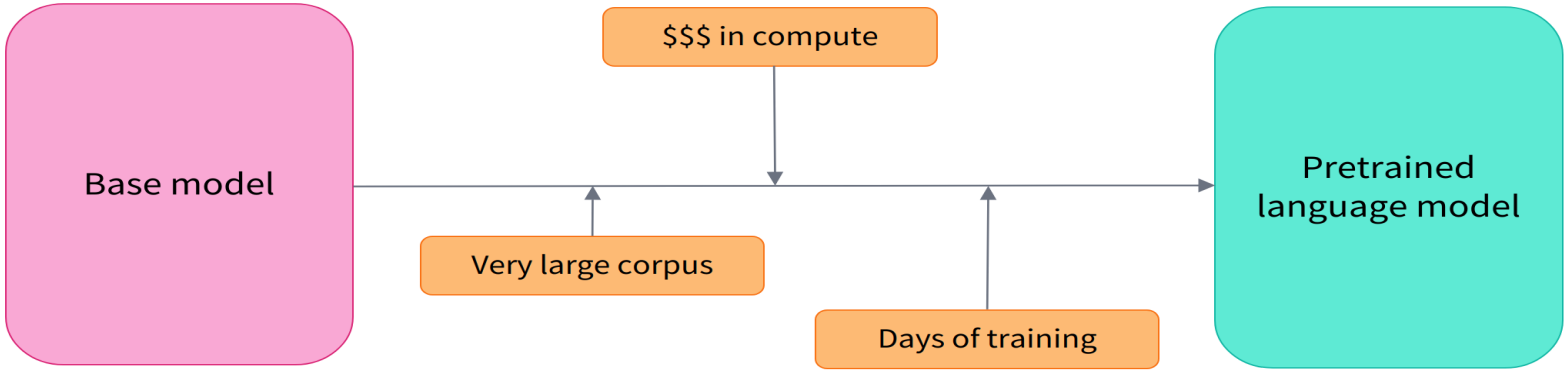
- All the Transformer models mentioned above (GPT, BERT, BART, T5, etc.) have been trained as language models.
- This means they have been trained on large amounts of raw text in a self-supervised fashion.
- Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model.
- That means that humans are not needed to label the data,

Transformers Are Big Models

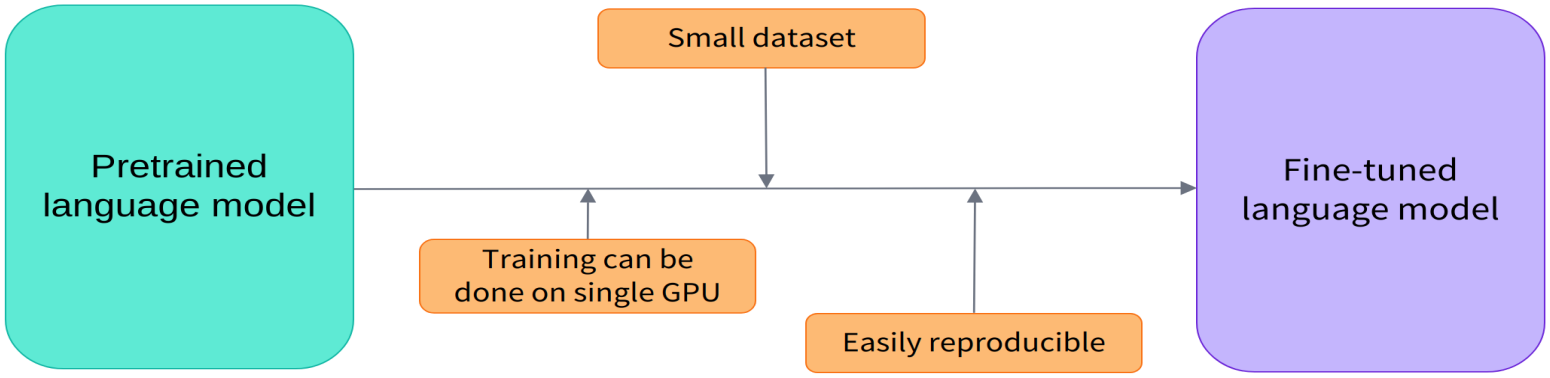
- Apart from a few outliers (like DistilBERT), the general strategy to achieve better performance is by increasing the models' sizes as well as the amount of data they are pretrained on.
- Unfortunately, training a model, especially a large one, requires a large amount of data.
- Self-supervised learning is a type of training in which the objective is automatically computed from the inputs of the model.
- This becomes very costly in terms of time and compute resources.
- Imagine if each time a research team, a student organization, or a company wanted to train a model, it did so from scratch. This would lead to huge, unnecessary global costs!

Transfer Learning

- Pretraining is the act of training a model from scratch.

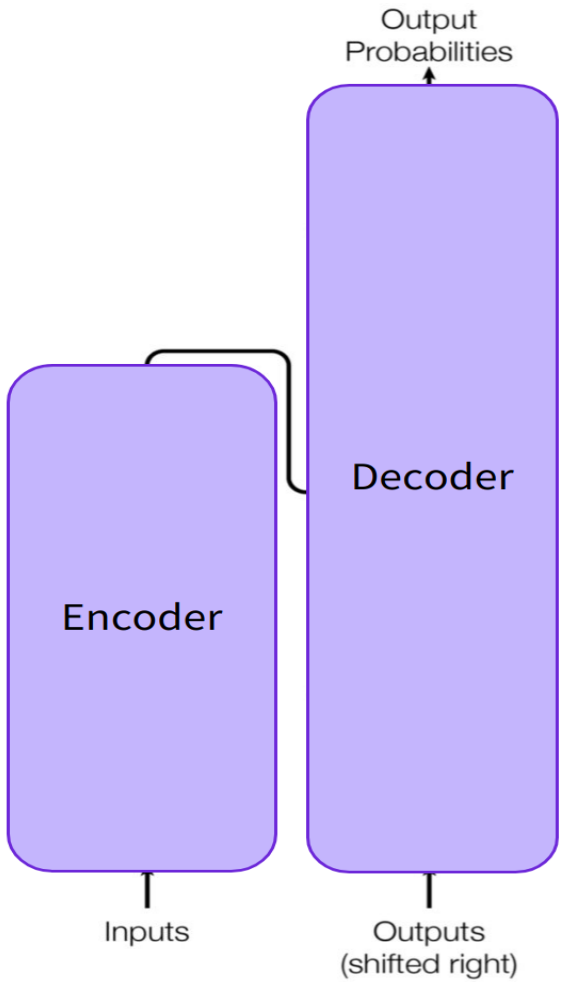


- Fine-tuning, on the other hand, is the training done after a model has been pretrained.



General Architecture

- The model is primarily composed of two blocks:
  - **Encoder**: The encoder receives an input and builds a representation of it (its features). This means that the model is optimized to acquire understanding from the input.
  - **Decoder**: The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence. This means that the model is optimized for generating outputs.



- Each of these parts can be used independently, depending on the task:

- **Encoder-only models**: Good for tasks that require understanding of the input, such as sentence classification and named entity recognition.

- **Decoder-only models**: Good for generative tasks such as text generation.

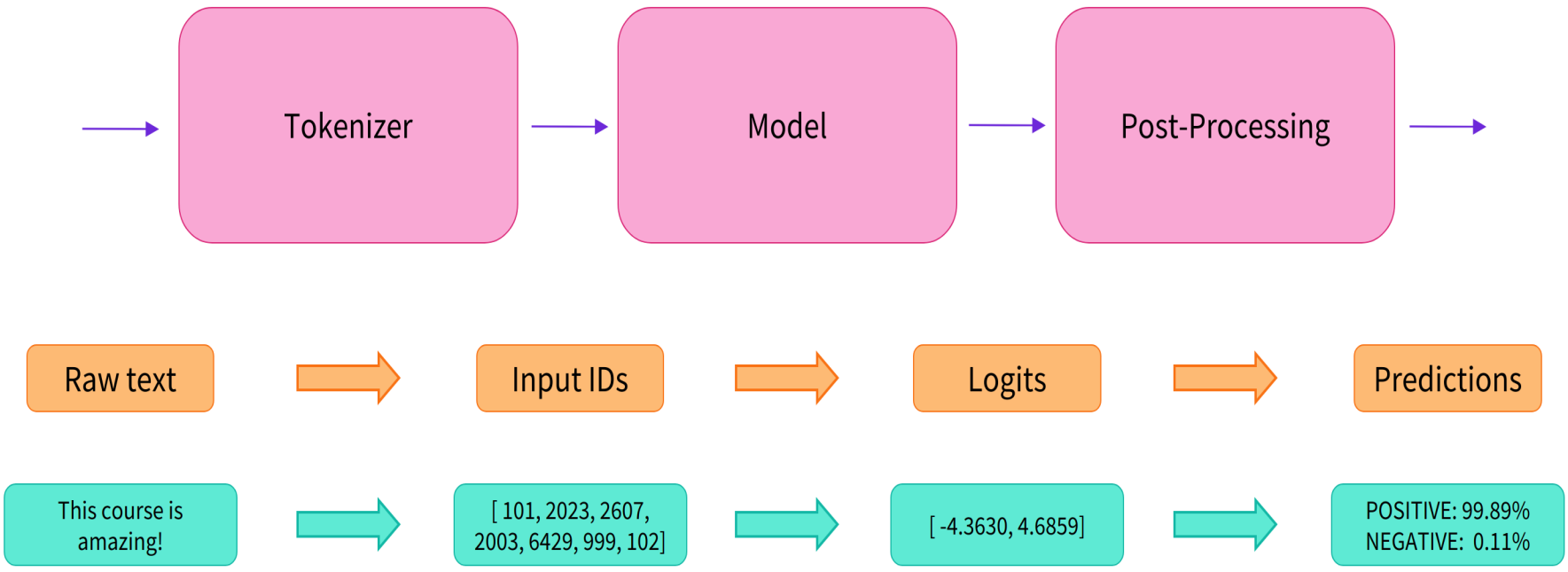
- **Encoder-decoder models or sequence-to-sequence models**: Good for generative tasks that require an input, such as translation or summarization.

Attention Layers

- A key feature of Transformer models is that they are built with special layers called attention layers.
- All you need to know is that this layer will tell the model to pay specific attention to certain words in the sentence you passed it.
- The same concept applies to any task associated with natural language.
- A word by itself has a meaning, but that meaning is deeply affected by the context, which can be any other word (or words) before or after the word being studied.

Behind the Pipeline

- The pipeline groups together three steps: preprocessing, passing the inputs through the model, and postprocessing:



Preprocessing with a Tokenizer

- Like other neural networks, Transformer models can't process raw text directly, so the first step of our pipeline is to convert the text inputs into numbers that the model can make sense of.

  - Splitting the input into words, subwords, or symbols (like punctuation) that are called tokens

  - Mapping each token to an integer

  - Adding additional inputs that may be useful to the model

Preprocessing with a Tokenizer

- Once we have the tokenizer, we can directly pass our sentences to it and we'll get back a dictionary that's ready to feed to our model! The only thing left to do is to convert the list of input IDs to tensors.
- You can use Transformers without having to worry about which ML framework is used as a backend; it might be PyTorch or TensorFlow orsome models.
- However, Transformer models only accept tensors as input. If this is your first time hearing about tensors, you can think of them as NumPy arrays instead.

Preprocessing with a Tokenizer

- A NumPy array can be a scalar (0D), a vector (1D), a matrix (2D), or have more dimensions. It's effectively a tensor; other ML frameworks' tensors behave similarly, and are usually as simple to instantiate as NumPy arrays.

```
1 | from transformers import AutoTokenizer
2 |
3 | checkpoint = "distilbert-base-uncased"
4 | tokenizer = AutoTokenizer.from_pretrained(checkpoint)
5 |
6 | raw_inputs = "I've been waiting for a this brownbag session."
7 | inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
8 | print(inputs)
9 | print(tokenizer.tokenize(raw_inputs))
```

- The output itself is a dictionary containing two keys, input_ids and attention_mask.

Going Through the Model

- We can download our pretrained model the same way we did with our tokenizer. The vector output by the Transformer module is usually large. It generally has three dimensions:

- **Batch size**: The number of sequences processed at a time (1 in our example).

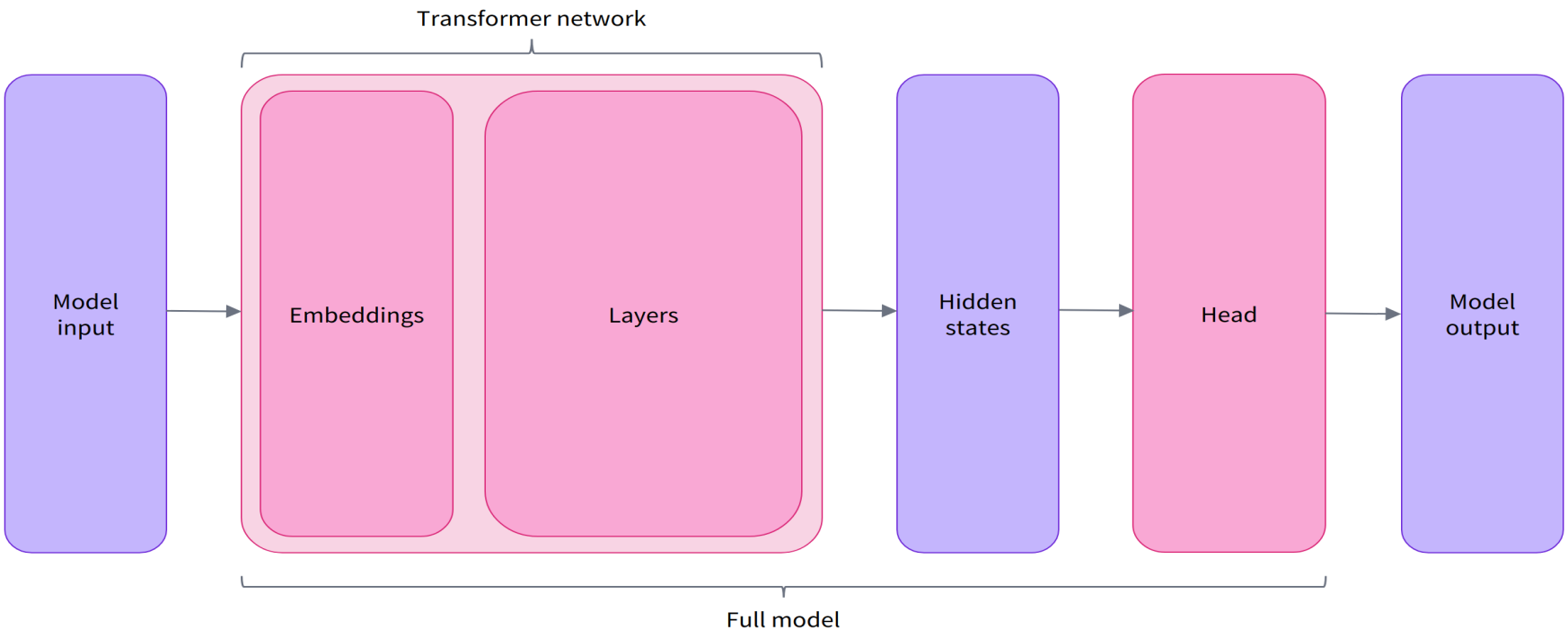
- **Sequence length**: The length of the numerical representation of the sequence (16 in our example).

- **Hidden size**: The vector dimension of each model input.

Model heads: Making Sense Out of Numbers

- The model heads take the high-dimensional vector of hidden states as input and project them onto a different dimension. They are usually composed of one or a few linear layers:

- The output of the Transformer model is sent directly to the model head to be processed.



Transformer Heads

- There are many different architectures available in Transformers, with each one designed around tackling a specific task.

- **Model (retrieve the hidden states)**
- **ForCausalLM**
- **ForMaskedLM**
- **ForMaskedLM**
- **ForMultipleChoice**
- **ForQuestionAnswering**
- **ForSequenceClassification**
- **ForTokenClassification**

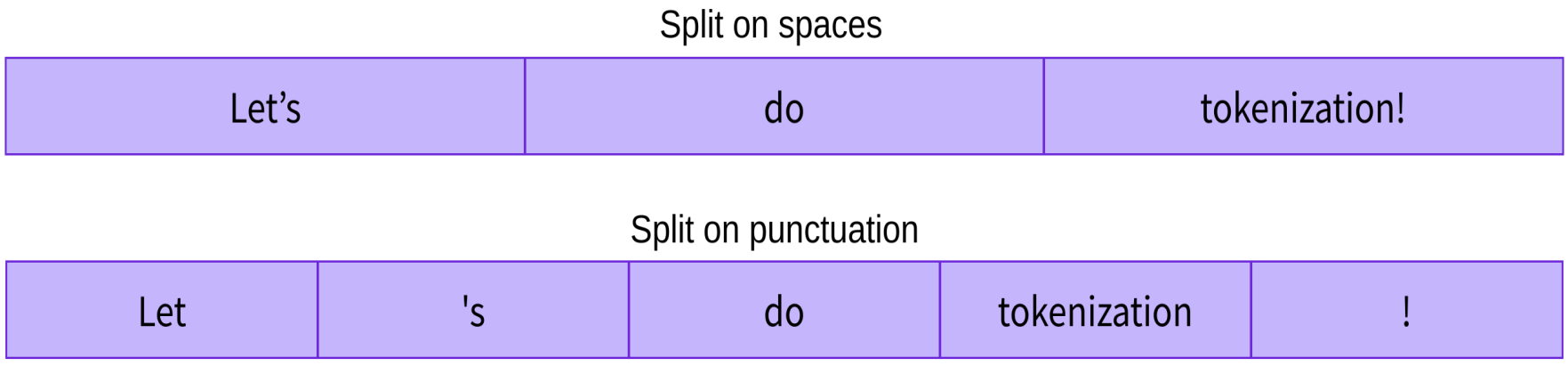
Transformer Head

- For our example, we will need a model with a sequence classification head (to be able to classify the sentences as positive or negative). So, we won't actually use the AutoModel class, but AutoModelForSequenceClassification:

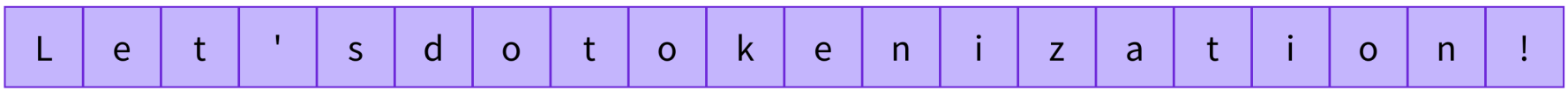
```
1  from transformers import AutoModelForSequenceClassification, AutoTokenizer
2  import torch
3
4  checkpoint = "distilbert-base-uncased"
5  tokenizer = AutoTokenizer.from_pretrained(checkpoint)
6  model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
7  raw_inputs = "I've been waiting for a this brownbag session."
8  inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
9  outputs = model(**inputs)
10 print(outputs.logits.shape)
11 print(outputs.logits)
12
13 predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
14 print(predictions)
15 print(model.config.id2label)
```

More on Tokenizers

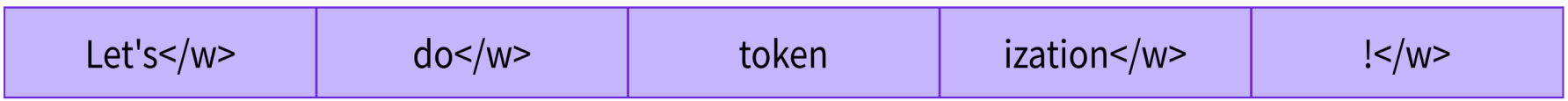
- Word Based Tokenization



- Character Based Tokenization



- Subword Tokenization



More on Tokenizers

- **Byte-level BPE**, as used in GPT-2
- **WordPiece**, as used in BERT
- **SentencePiece or Unigram**, as used in several multilingual models

Encoding and Decoding

- The tokenization process is done by the tokenize method of the tokenizer.The conversion to input IDs is handled by the convert_tokens_to_ids tokenizer method. Decoding is going the other way around: from vocabulary indices, we want to get a string. This can be done with the decode method as follows.

```
1  from transformers import AutoTokenizer
2  tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
3  sequence = "Using a Transformer network is simple"
4
5  tokens = tokenizer.tokenize(sequence)
6  print(tokens)
7  ids = tokenizer.convert_tokens_to_ids(tokens)
8  print(ids)
9  decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])
10 print(decoded_string)
```

GLUE

GLUE

- GLUE is centered on nine English sentence understanding tasks, which cover a broad range of domains, data quantities, and difficulties. Source
- Leaderboard
- Dataset for Download

- **SINGLE-SENTENCE TASKS** : CoLA ,SST-2

- **SIMILARITY AND PARAPHRASE TASKS** : MRPC , QQP, STS-B

- **INFERENCE TASKS**: MNLI, QNLI, RTE, WNLI

- The Corpus of Linguistic Acceptability (Warstadt et al., 2018) consists of English acceptability judgments drawn from books and journal articles on linguistic theory. Each example is a sequence of words annotated with whether it is a grammatical English sentence. Following the authors, we use Matthews correlation coefficient (Matthews, 1975) as the evaluation metric, which evaluates performance on unbalanced binary classification and ranges from -1 to 1, with 0 being the performance of uninformed guessing. We use the standard test set, for which we obtained private labels from the authors.

SST-2

- The Stanford Sentiment Treebank (Socher et al., 2013) consists of sentences from movie reviews and human annotations of their sentiment. The task is to predict the sentiment of a given sentence. We use the two-way (positive/negative) class split, and use only sentence-level labels.

MRPC

- The Microsoft Research Paraphrase Corpus (Dolan & Brockett, 2005) is a corpus of sentence pairs automatically extracted from online news sources, with human annotations for whether the sentences in the pair are semantically equivalent. Because the classes are imbalanced (68% positive), we follow common practice and report both accuracy and F1 score.

QQP

- The Quora Question Pairs dataset is a collection of question pairs from the community question-answering website Quora. The task is to determine whether a pair of questions are semantically equivalent. As in MRPC, the class distribution in QQP is unbalanced (63% negative), so we report both accuracy and F1 score. We use the standard test set, for which we obtained private labels from the authors.

STS-B

- The Semantic Textual Similarity Benchmark (Cer et al., 2017) is a collection of sentence pairs drawn from news headlines, video and image captions, and natural language inference data. Each pair is human-annotated with a similarity score from 1 to 5; the task is to predict these scores.

MNLI

- The Multi-Genre Natural Language Inference Corpus (Williams et al., 2018) is a crowdsourced collection of sentence pairs with textual entailment annotations. Given a premise sentence and a hypothesis sentence, the task is to predict whether the premise entails the hypothesis (entailment), contradicts the hypothesis (contradiction), or neither (neutral). The premise sentences are gathered from ten different sources, including transcribed speech, fiction, and government reports.We use the standard test set, for which we obtained private labels from the authors, and evaluate on both the matched (in-domain) and mismatched (cross-domain) sections.

QNLI

- The Stanford Question Answering Dataset (Rajpurkar et al. 2016) is a question-answering dataset consisting of question-paragraph pairs, where one of the sentences in the paragraph (drawn from Wikipedia) contains the answer to the corresponding question (written by an annotator). We convert the task into sentence pair classification by forming a pair between each question and each sentence in the corresponding context, and filtering out pairs with low lexical overlap between the question and the context sentence. The task is to determine whether the context sentence contains the answer to the question

- The Recognizing Textual Entailment (RTE) datasets come from a series of annual textual entailment challenges. We combine the data from RTE1 (Dagan et al., 2006), RTE2 (Bar Haim et al., 2006), RTE3 (Giampiccolo et al., 2007), and RTE5 (Bentivogli et al., 2009). Examples are constructed based on news and Wikipedia text. We convert all datasets to a two-class split, where for three-class datasets we collapse neutral and contradiction into not entailment, for consistency.

- The Winograd Schema Challenge (Levesque et al., 2011) is a reading comprehension task in which a system must read a sentence with a pronoun and select the referent of that pronoun from a list of choices. The examples are manually constructed to foil simple statistical methods: Each one is contingent on contextual information provided by a single word or phrase in the sentence. To convert the problem into sentence pair classification, we construct sentence pairs by replacing the ambiguous pronoun with each possible referent.

Transformer Full Training

```
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding
from torch.utils.data import DataLoader
from transformers import AutoModelForSequenceClassification
from transformers import AdamW
from transformers import get_scheduler
import torch
from tqdm.auto import tqdm
from datasets import load_metric

num_epochs = 3

raw_datasets = load_dataset("glue", "mrpc")
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(example):
    return tokenizer(example["sentence1"], example["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)
```

Transformer Full Training

```
tokenized_datasets = tokenized_datasets.remove_columns(["sentence1", "sentence2", "idx"])
tokenized_datasets = tokenized_datasets.rename_column("label", "labels")
tokenized_datasets.set_format("torch")
tokenized_datasets["train"].column_names

train_dataloader = DataLoader(tokenized_datasets["train"],
                              shuffle=True, batch_size=8, collate_fn=data_collator)
eval_dataloader = DataLoader(tokenized_datasets["validation"],
                             batch_size=8, collate_fn=data_collator)

for batch in train_dataloader:
    break
{k: v.shape for k, v in batch.items()}


model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)
outputs = model(**batch)
print(outputs.loss, outputs.logits.shape)
optimizer = AdamW(model.parameters(), lr=5e-5)
```

Transformer Full Training

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

device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
model.to(device)
device

progress_bar = tqdm(range(num_training_steps))

model.train()
for epoch in range(num_epochs):
    for batch in train_dataloader:
        batch = {k: v.to(device) for k, v in batch.items()}
        outputs = model(**batch)
        loss = outputs.loss
        loss.backward()

        optimizer.step()
        lr_scheduler.step()
        optimizer.zero_grad()
        progress_bar.update(1)
```

Transformer Full Training

```
metric = load_metric("glue", "mrpc")
model.eval()
for batch in eval_dataloader:
    batch = {k: v.to(device) for k, v in batch.items()}
    with torch.no_grad():
        outputs = model(**batch)

    logits = outputs.logits
    predictions = torch.argmax(logits, dim=-1)
    metric.add_batch(predictions=predictions, references=batch["labels"])

metric.compute()
```

Downloading Wikipeida

Downloading a Wikipedia Dump

- Wikipedia dumps are freely available in multiple formats in many languages.
- For the English language Wikipedia, a full list of all available formats of the latest dump can be found HERE
- As we're primarily interested in text data
- We'll download such a dump (that contains solely pages and articles) in a compressed XMLformat
- To download the latest Wikipedia dump for the English language, for example, simply run the following command in your terminal: **`./download_wiki_dump.sh en`**

Extracting and Cleaning a Wikipedia Dump

- The Wikipedia dump we have just downloaded is not ready to be pre-processed (sentence tokenized and one sentence per line) just yet.
- First, we need to extract and clean the dump, which can easily be accomplished with WikiExtractor
- Install it using pip first
- To extract and clean the Wikipedia dump we have just downloaded, for example, simply run the following command in your terminal:
- Run **`./extract_and_clean_wiki_dump.sh enwikilatest-pages-articles.xml.bz2`**

Pre-processing a Wikipedia dump

- Now that we have successfully downloaded, extracted and cleaned a Wikipedia dump, we can begin to pre-process it.
- Practically, this means sentence-tokenizing the articles, as well as writing them one-sentence-per-line to a single text file.
- Install it using pip first then.
- It can be accomplished using Microsoft's blazingly fast BlingFire tokenizer
- Run **`python3 preprocess_wiki_dump.py enwiki-latest-pages-articles.txt`**

Pretraining a RoBERTa Model from Scratch

Step 1: Loading the dataset

- I chose to use the works of Immanuel Kant (1724-1804), the German philosopher, who was the epitome of the Age of Enlightenment.
- The idea is to introduce human-like logic and pretrained reasoning for downstream reasoning tasks.
- Project Gutenberg, `https://www.gutenberg.org`, offers a wide range of free eBooks that can be downloaded in text format.
- You can use other books if you want to create customized datasets of your own based on books.
- kant.txt provides a small training dataset to train the transformer mode.
- Run the download.sh file to download the txt file.

Step 2: Training a tokenizer

- Hugging Face's ByteLevelBPETokenizer() will be trained using kant.txt.
- A bytelevel tokenizer will break a string or word down into a sub-string or sub-word.

- There are two main advantages among many others:
  - The tokenizer can break words into minimal components. Then it will merge these small components into statistically interesting ones. For example, "smaller" and smallest" can become "small," "er," and "est."
- The chunks of strings classified as an unknown unk_token, using WorkPiece level encoding, will practically disappear.

Step 2: Training a tokenizer

- In this model, we will be training the tokenizer with the following parameters
- files=paths is the path to the dataset.
- vocab_size=52_000 is the size of our tokenizer's model length.
- min_frequency=2 is the minimum frequency threshold
- special_tokens=[] is a list of special tokens.
- $< s >$: a start token, $< pad >$: a padding token, $< /s >$: an end token, $< unk >$: an unknown token, $< mask >$: the mask token for language modeling

Step 3: Saving the Files to Disk

- The tokenizer will generate two files when trained:
  - merges.txt, which contains the merged tokenized sub-strings
  - vocab.json, which contains the indices of the tokenized sub-strings
  - The files in this example are small. You can double-click on them to view their contents.
- merges.txt contains the tokenized sub-strings as planned; vocab.json contains the indices

Step 4: Loading the Trained Tokenizer Files

- We could have loaded pretrained tokenizer files. However, we trained our own tokenizer and now are ready to load the files
  - The tokenizer can encode a sequence.
  - The tokenizer now processes the tokens to fit the BERT model variant used
  - The data for the training model is now ready to be trained.

Step 5: Defining the Configuration of the Model

- We will be pretraining a RoBERTa-type transformer model using the same number of layers and heads as a DistilBERT transformer.
- The model will have a vocabulary size set to 52,000, 12 attention heads, and 6 layers
- We will explore the configuration in more detail.
- Let's first recreate the tokenizer in our model.

Step 6: Reloading the Tokenizer in Transformers

- We are now ready to load our trained tokenizer, which is our pretrained tokenizer in RobertaTokenizer.from_pretained()
- Now that we have loaded our trained tokenizer, let's initialize a RoBERTa model from scratch.

Step 7: Initializing a Model from Scratch

- It will initialize a model from scratch and examine the size of the model.
  - The program first imports a RoBERTa masked model for language modeling.
  - The model is initialized with the configuration.
  - Take some time to go through the details of the output of the configuration before continuing.

Step 8: Step 10: Building the Dataset

- The program will now load the dataset line by line for batch training with block_size=128 limiting the length
  - The output shows that Hugging Face has invested a considerable amount of resources into optimizing the time it takes to process data
  - The program will now define a data collator to create an object for backpropagation.

Step 9: Defining a Data Collator

- We need to run a data collator before initializing the trainer.
  - A data collator will take samples from the dataset and collate them into batches.
  - The results are dictionary like objects.
  - We are preparing a batched sample process for Masked Language Modeling (MLM) by setting mlm=True
  - We also set the number of masked tokens to train mlm_probability=0.15.
  - We now initialize data_collator with our tokenizer, MLM activated.

Step 10: Initializing the Trainer

- The previous steps have prepared the information required to initialize the trainer.
- The dataset has been tokenized and loaded. Our model is built. The data collator has been created.
- For ease of training purposes, the program trains the model quickly. The number of epochs is limited to one
- The GPU comes in handy since we can share the batches and multi-process the training tasks
- Everything is ready. The trainer is launched with one line of code:
- The output displays the training process in real time showing the loss, learning rate, epoch, and steps

Exercise - Lecture 9

- Exercise 1 to 1
- Class-Ex-Lecture9.py