A practical introduction to Large Language Models

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

Goal of this course

- Understand how LLMs work, what they can be used for
- Being able to deploy LLMs using Hugging Face
- Practice RAG and Agents

Part 1: What is a Large Language Model?

- Language models
- Tokenization
- Embeddings
- The attention mechanism
- Encoder / decoder
- Pre-training
- Fine-tuning

Language models

What is a language model (LM)?

Input: a sentence (as a sequence of tokens)

Output: predict the next token

Basic examples:

- *Markov chain* is a LM, it gives a probabilistic distribution over the next token given the last token
- Naturally extended to *n-grams*: use the (n-1) last tokens

N-GRAMS ARE LIMITED

Number of parameters: *vocab_size* *context_length*

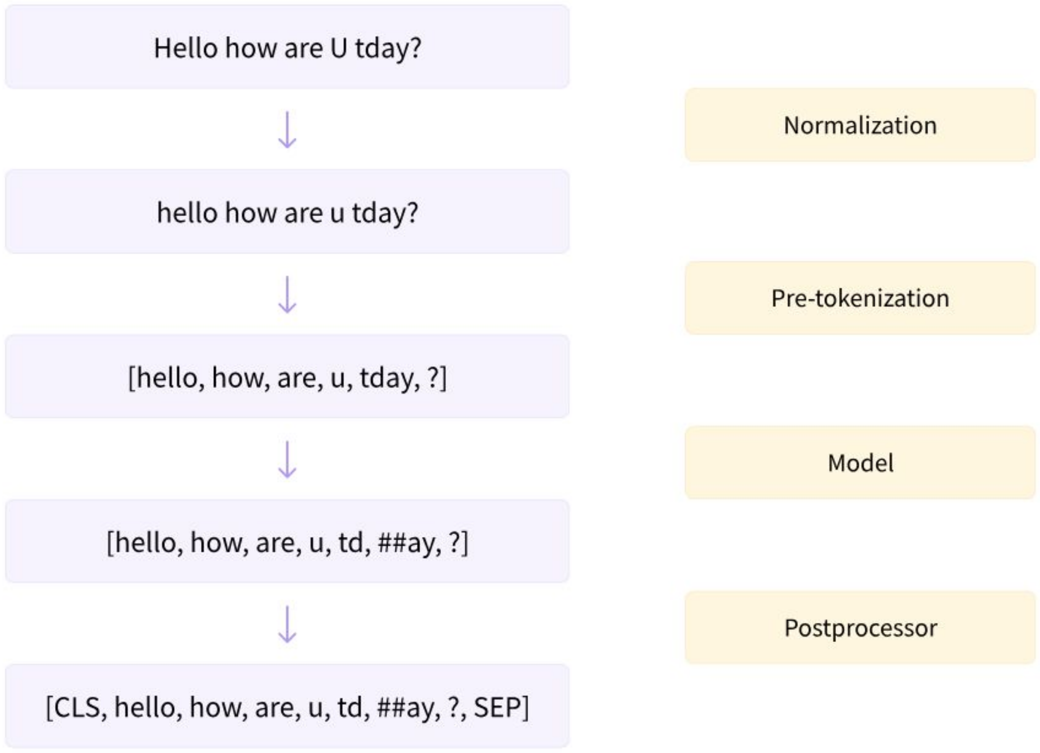
vocab_size: total number of tokens

context_length: number of tokens considered for prediction

Think of it as a very large matrix…

TOKENIZATION

Tokenization: decomposing a sentence into a sequence of tokens



Every single explanation you will ever see about Language Models use **words**, **BUT** in reality the unit object is **tokens**

WORDS != TOKENS

What it actually looks like:

```
test = "hello world"
test_encoded = tokenizer.encode(test)
test_encoded, [tokenizer.decode([x]) for x in test_encoded], tokenizer.decode(test_encoded)
```

```
([258, 285, 111, 492], ['he', 'll', 'o', ' world'], 'hello world')
```

Bottom line: at this point, we have converted a text into a sequence of integers (which represent tokens).

GPT-2 has 50,257 tokens

In practice

Tiktoken implements BPE (one of the two most popular tokenization algorithms):

https://github.com/openai/tiktoken

Embeddings

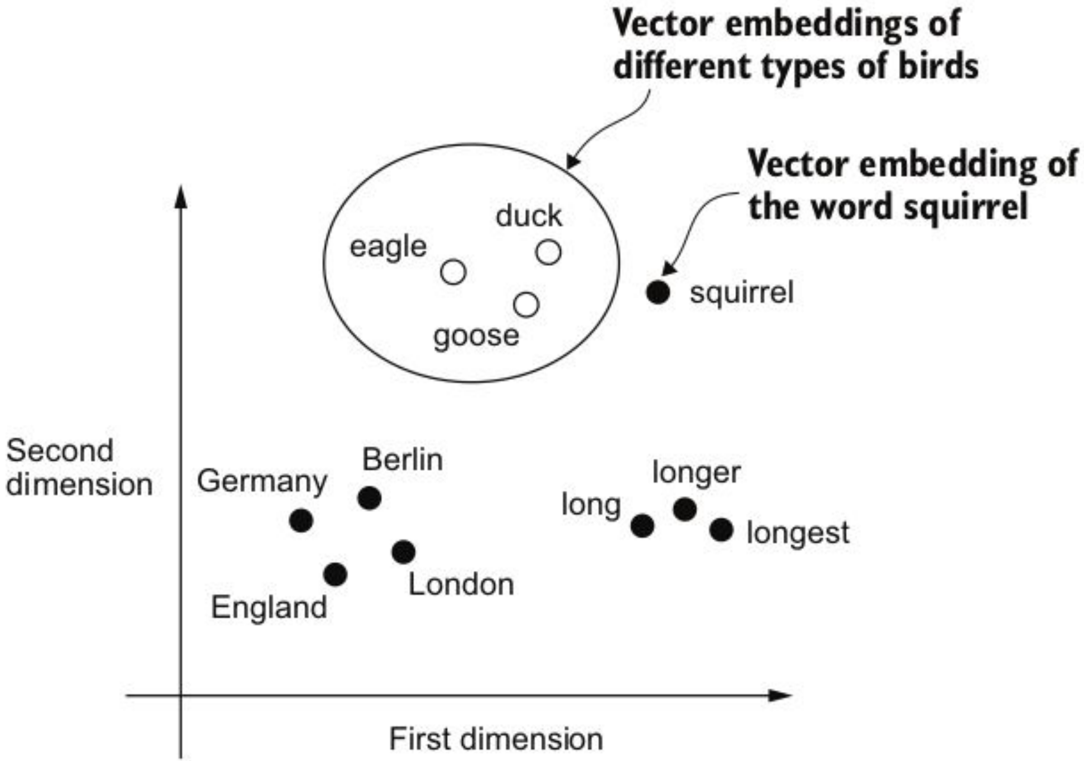
The 2003 (silent) breakthrough

Journal of Machine Learning Research 3 (2003) 1137–1155 Submitted 4/02; Published 2/03

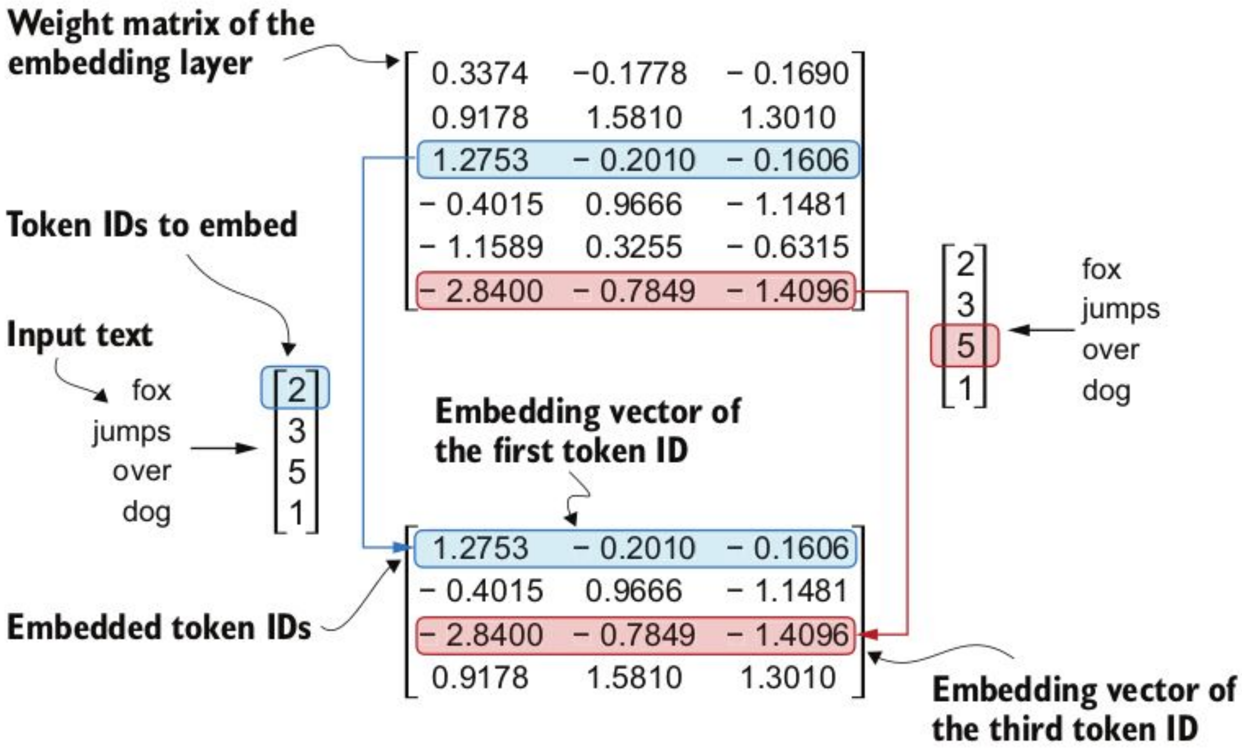
A Neural Probabilistic Language Model

Yoshua Bengio BENGIOY@IRO.UMONTREAL.CA
Réjean Ducharme DUCHARME@IRO.UMONTREAL.CA
Pascal Vincent VINCENTP@IRO.UMONTREAL.CA
Christian Jauvin JAUVINC@IRO.UMONTREAL.CA
Département d'Informatique et Recherche Opérationnelle
Centre de Recherche Mathématiques
Université de Montréal, Montréal, Québec, Canada

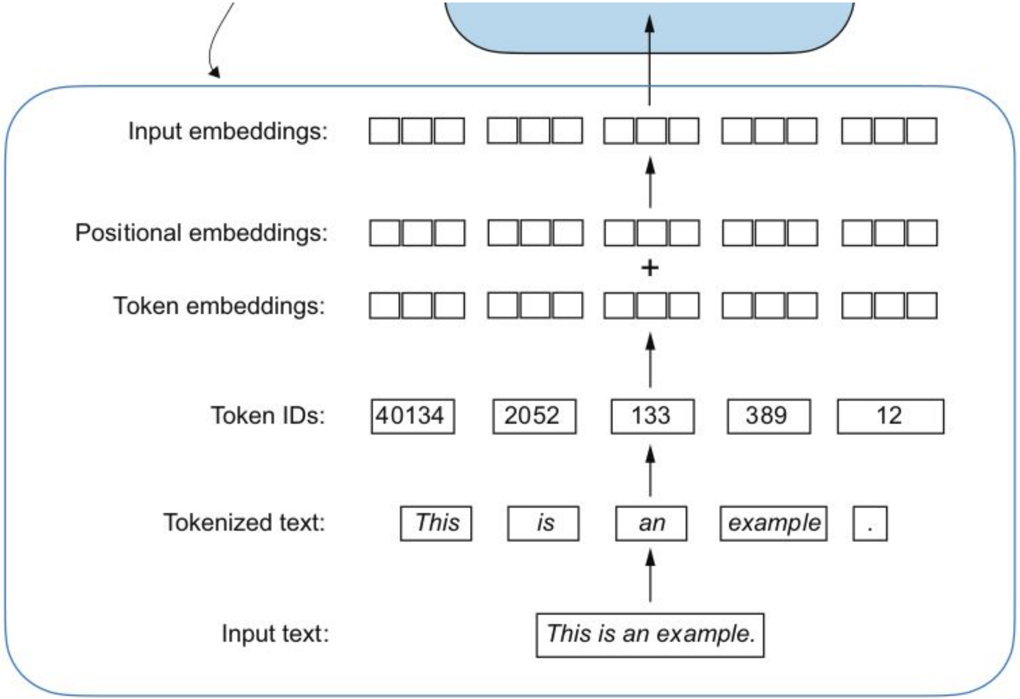
Key idea: Embeddings



From text to vectors



Bottom line: at this point, we have converted a text into a sequence of (floating point) vectors. These are (almost) the inputs for our models.



Statistics

The smallest GPT-2 models (117M and 125M parameters) use an embedding size of 768 dimensions.

The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

The attention mechanism

Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Attention is all you need

The paper came in 2017, in a wave of more and more complicated architectures around recurrent neural networks (RNNs), aiming at dealing with long contexts.

It does not do anything radically new: it says that **“attention mechanism is enough to enable long contexts”**.

A SIDE-NOTE

OpenAI scientist Noam Brown:

"The incredible progress in AI over the past five years can be summarized in one word: scale."

Recently, older architectures (LSTMs) reached similar performances as Transformers…

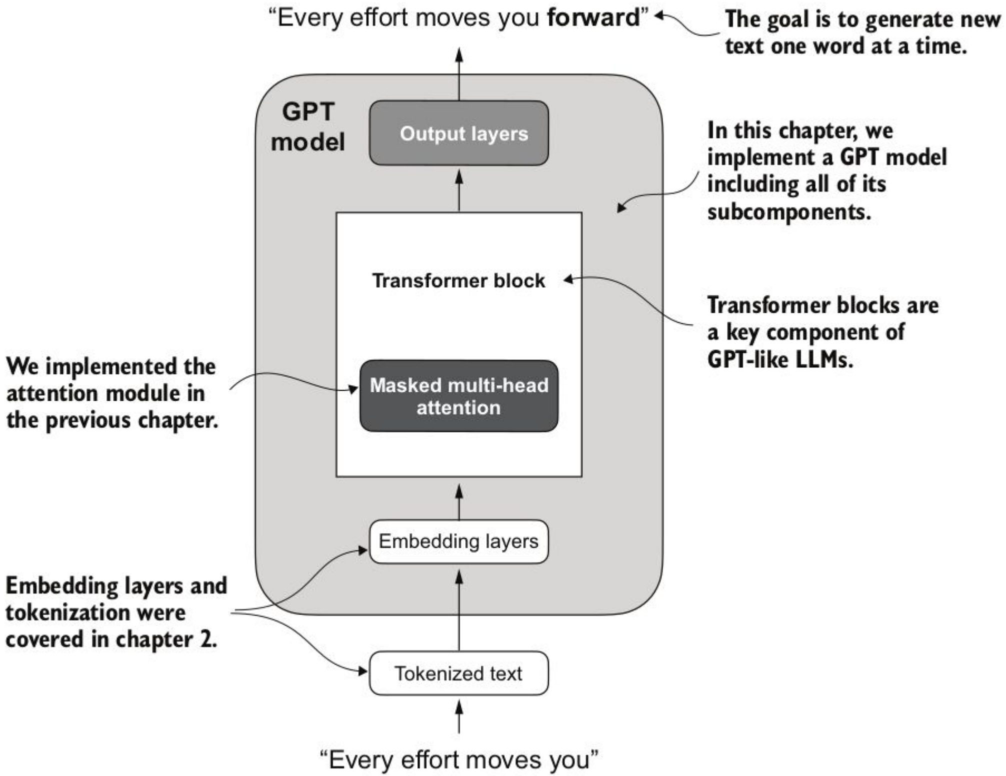
A self-attention head

Input: an embedding vector x(i) for each token i

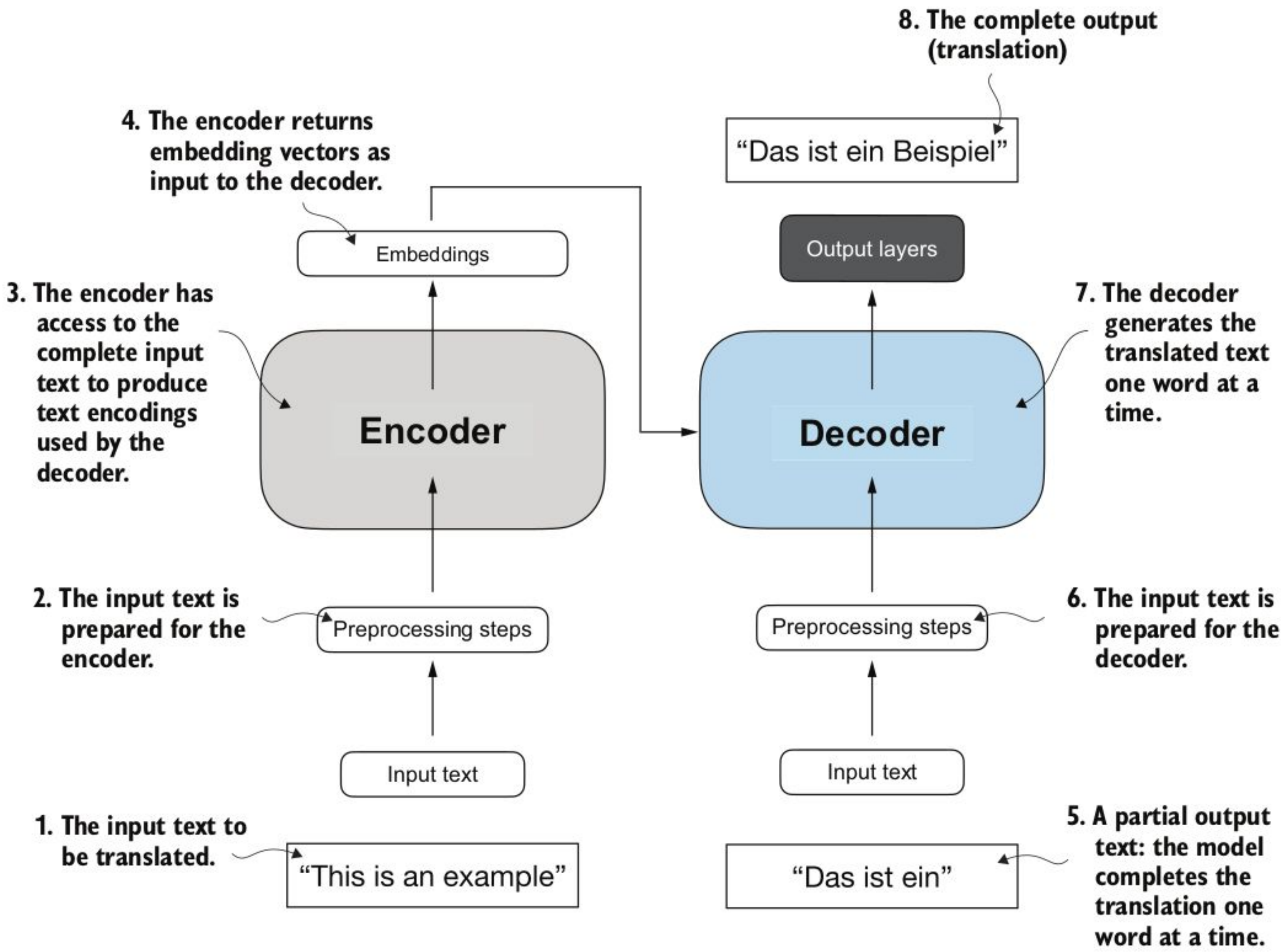
Output: a context vector z(i) for each token i

Intuition: z(i) gathers *contextual* information

Attention heads as key components in a Transformer

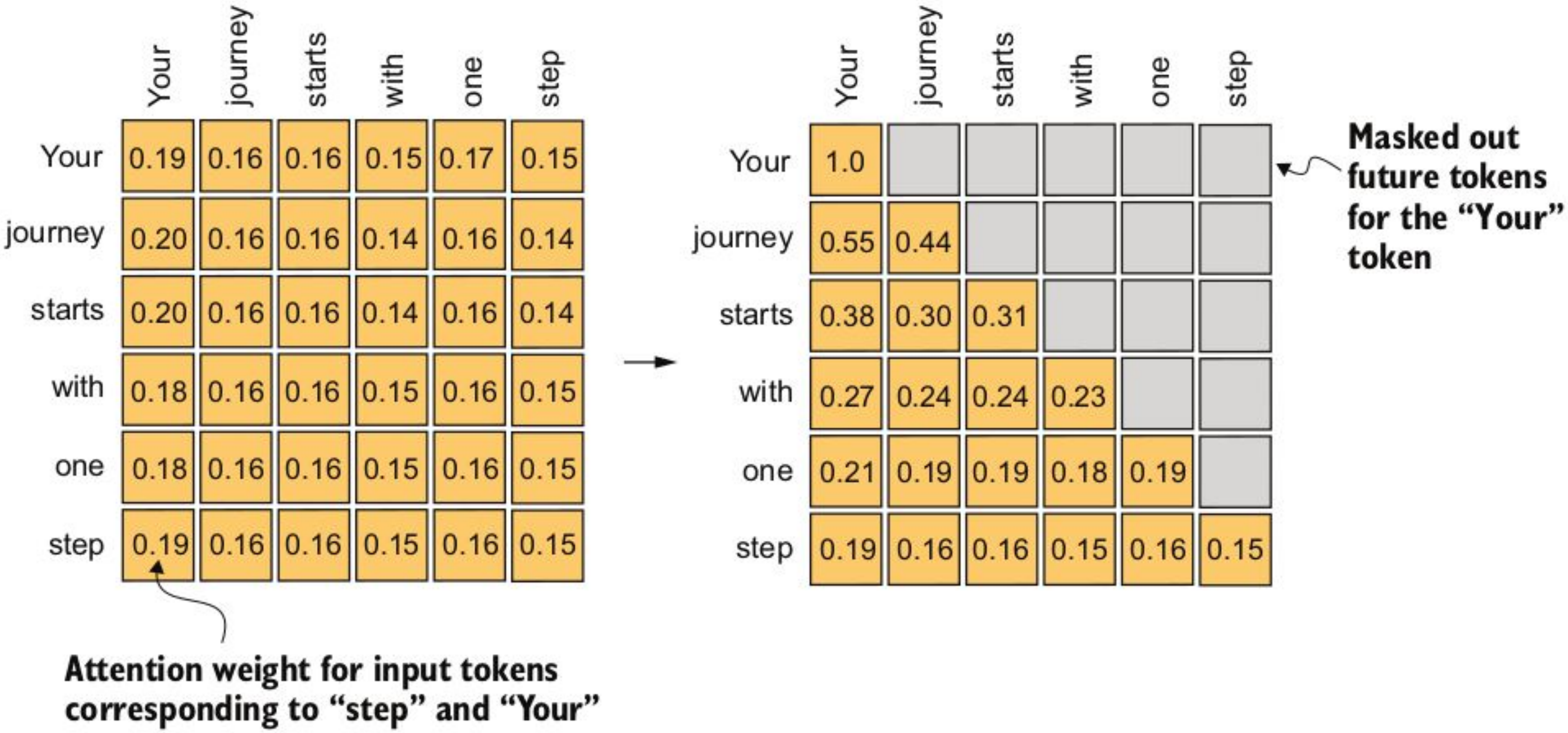


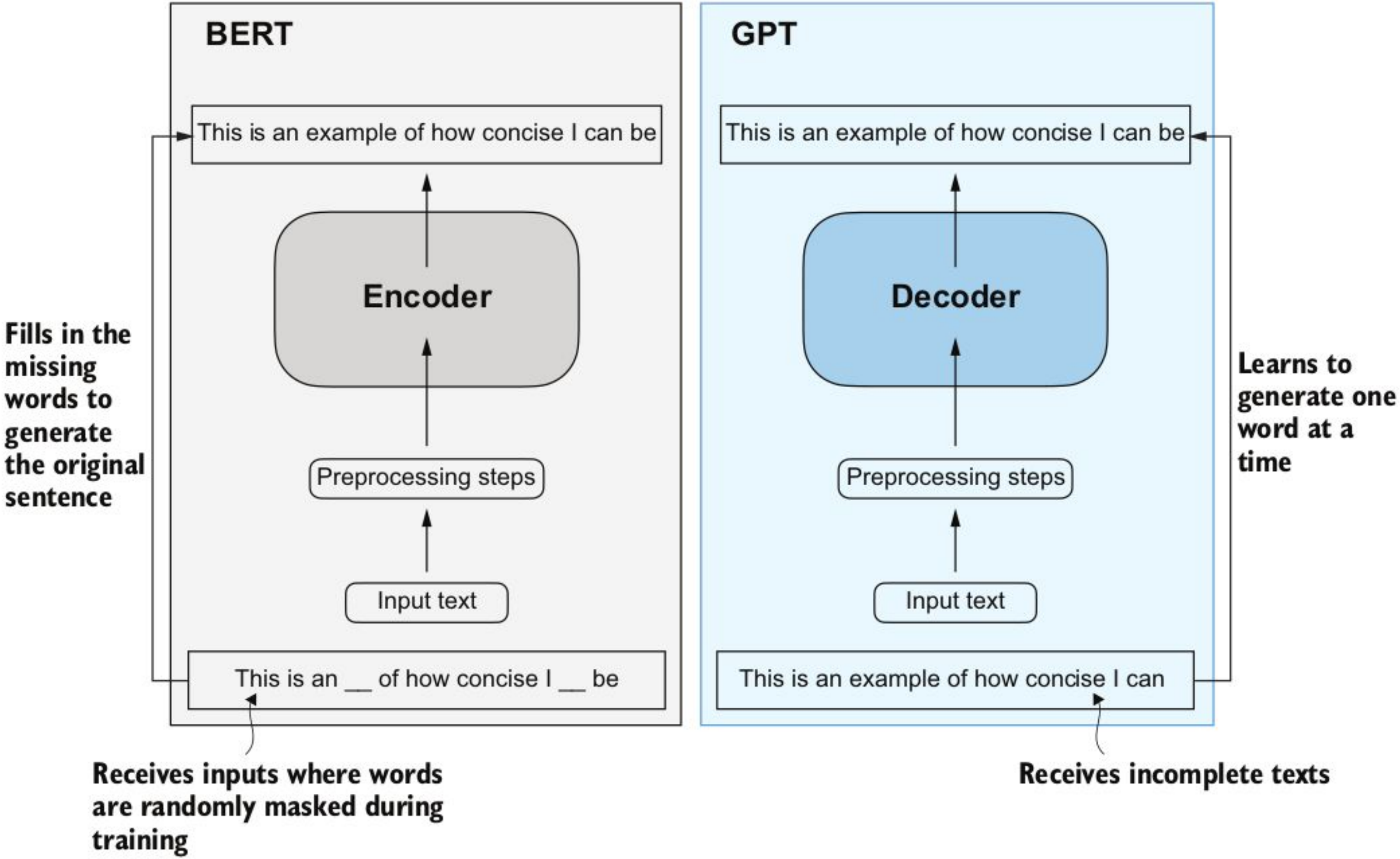
Encoder / decoder



8. The complete output (translation)
"Das ist ein Beispiel"
4. The encoder returns embedding vectors as input to the decoder.
Embeddings
Output layers
3. The encoder has access to the complete input text to produce text encodings used by the decoder.
Encoder
Decoder
7. The decoder generates the translated text one word at a time.
2. The input text is prepared for the encoder.
Preprocessing steps
Preprocessing steps
6. The input text is prepared for the decoder.
Input text
Input text
1. The input text to be translated.
"This is an example"
"Das ist ein"
5. A partial output text: the model completes the translation one word at a time.

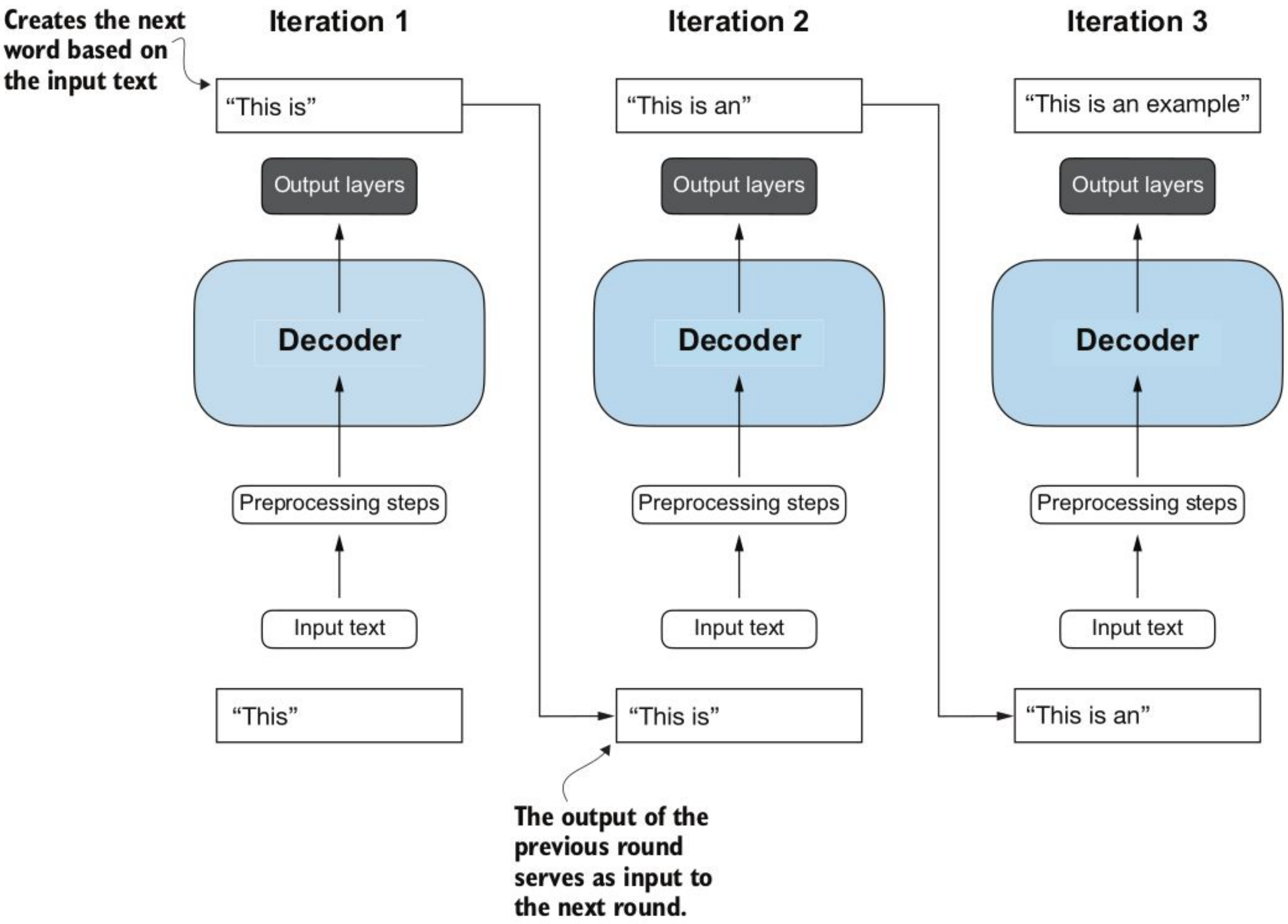
Decoders use causal attention



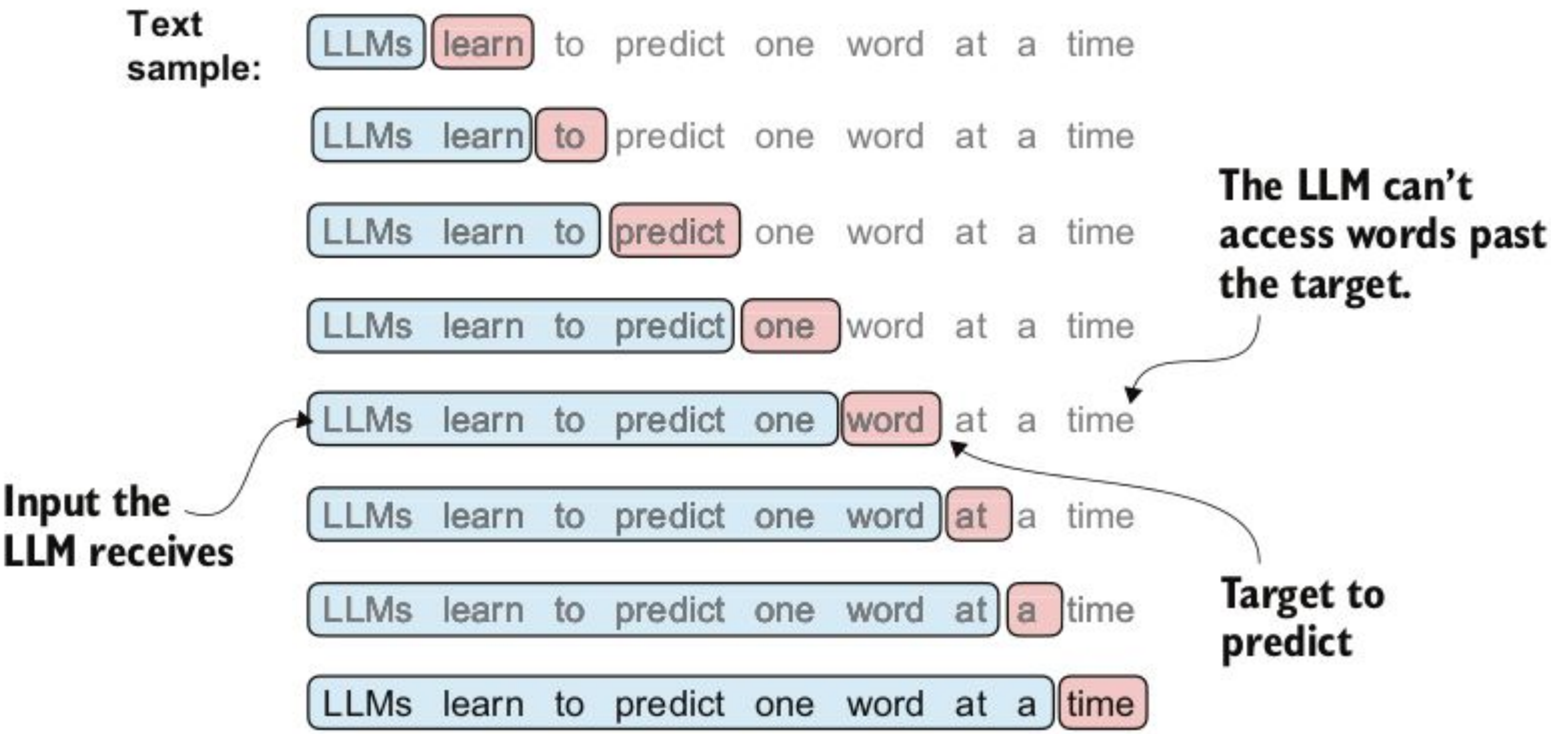

BERT
This is an example of how concise I can be
Encoder
Preprocessing steps
Input text
This is an __ of how concise I __ be
Fills in the missing words to generate the original sentence
Receives inputs where words are randomly masked during training
GPT
This is an example of how concise I can be
Decoder
Preprocessing steps
Input text
This is an example of how concise I can
Learns to generate one word at a time
Receives incomplete texts

What does "auto-regressive" mean?

It means that for generating a single new token we feed the model with the input + all tokens generated so far.


Creates the next word based on the input text
Iteration 1
Iteration 2
Iteration 3
“This is”
“This is an”
“This is an example”
Output layers
Output layers
Output layers
Decoder
Decoder
Decoder
Preprocessing steps
Preprocessing steps
Preprocessing steps
Input text
Input text
Input text
“This”
“This is”
“This is an”
The output of the previous round serves as input to the next round.

Sliding windows



Models' signatures

Input: x of shape (context_length), y of shape (context_length)

Output: model(x,y) = (logits, loss) where

- logits has shape (context_length, vocab_size)
- loss has shape (context_length)

For each window, make the prediction and compute the loss

Models' signatures with batching

Input: X of shape (batch_size, context_length), Y of shape (batch_size, context_length)

Output: model(X,Y) = (logits, loss) where

- logits has shape (batch_size, context_length, vocab_size)
- loss has shape (batch_size, context_length)

Pre-training

Training a language model

Input: a sentence (as a sequence of tokens)

Output: predict the next token

Training loop:

- Feed the model a piece of text, using the next token as target
- Apply gradient descent to increase the probability to generate the target

What is cross entropy loss?

Cross entropy measures the difference between probability distributions: it quantifies the dissimilarity between the predicted probability distribution and the true probability distribution.

In language modelling we do not have the true distribution of words, it is approximated from a training set:

$$H(T, q) = -\sum_{i=1}^{N} \frac{1}{N} \log_2 q(x_i)$$

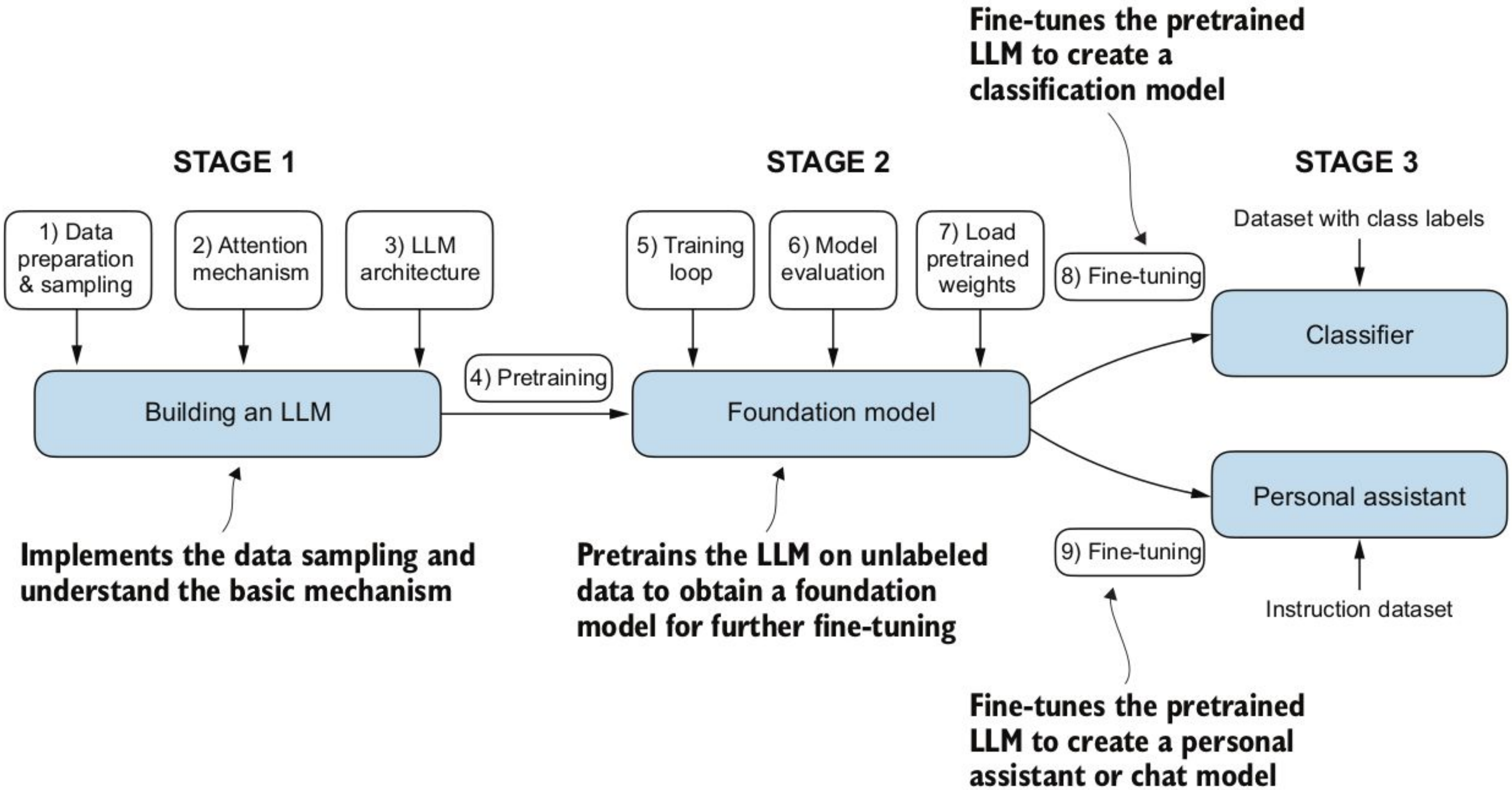
Where N is the number of tokens in the training set and q(x_i) is the probability that the model outputs x_i.

Fine-tuning

FOUNDATION MODELS

Language Models are not very useful, they randomly generate texts… But this means that they somehow capture some information from natural language! They are also called *foundation models*.

Fine-tuning is about making Language Models solve concrete tasks, like classification, question answering, named-entity recognition…



Fine-tunes the pretrained LLM to create a classification model
STAGE 1
STAGE 2
STAGE 3
1) Data preparation & sampling
2) Attention mechanism
3) LLM architecture
5) Training loop
6) Model evaluation
7) Load pretrained weights
8) Fine-tuning
Dataset with class labels
Classifier
4) Pretraining
Building an LLM
Foundation model
Personal assistant
9) Fine-tuning
Instruction dataset
Implements the data sampling and understand the basic mechanism
Pretrains the LLM on unlabeled data to obtain a foundation model for further fine-tuning
Fine-tunes the pretrained LLM to create a personal assistant or chat model

Training is expensive

We often cannot afford updating the ***whole*** model!

Most of us will not train foundation models… Rather fine-tune existing ones.

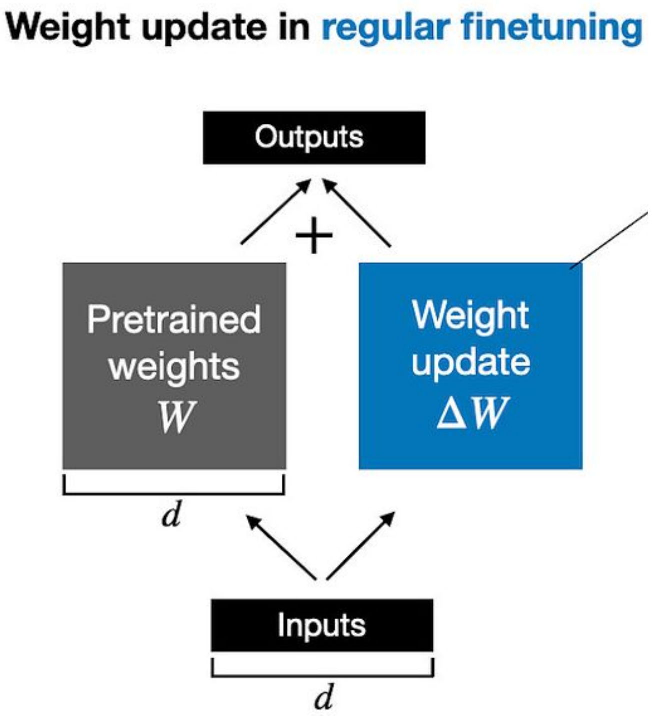
Low-rank Adaptation (LoRA)

Two key ideas:

(1) We only store the changes, not a new model
(2) We only update a small number of parameters

Idea: storing weight updates

Say we consider a linear layer with matrix W. We keep the matrix W fixed and store ΔW

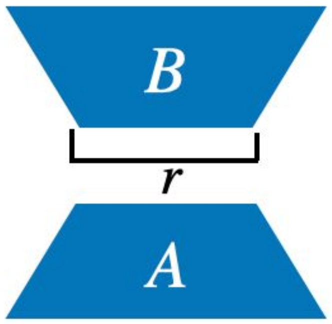


Rank approximations

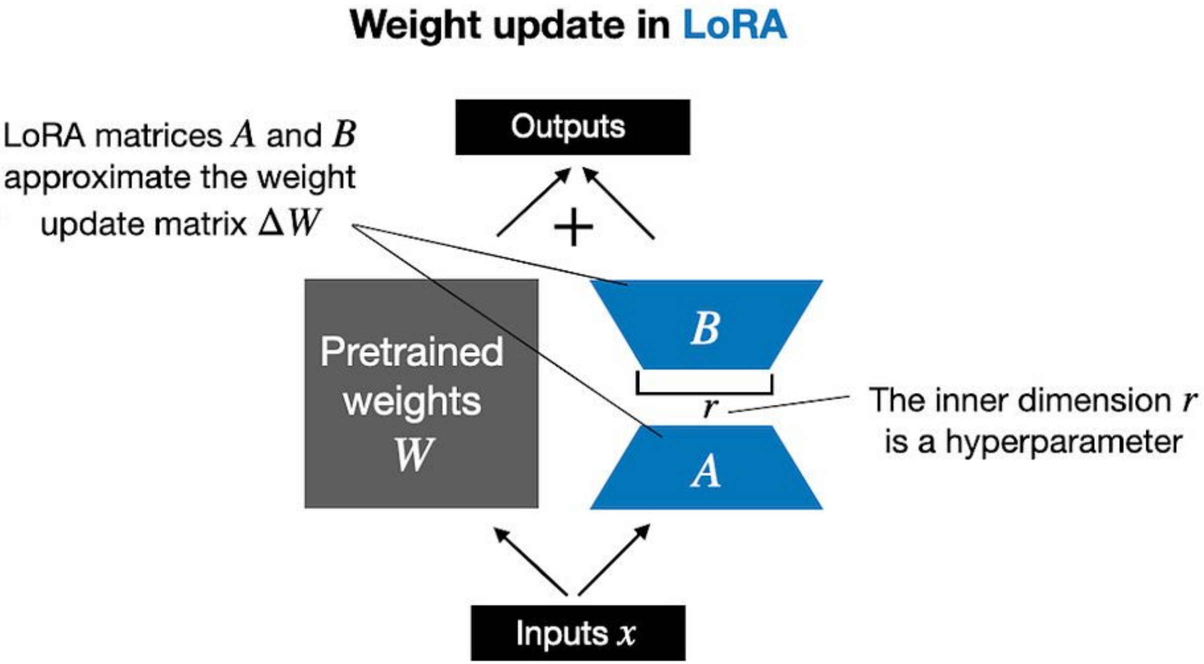
A matrix W of dimension dxd contains dxd parameters. It can be ***rank-r approximated*** by two matrices AxB with:

- A of dimension dxr
- B of dimension rxd

Instead of dxd parameters we now have 2xdxr parameters.



Weight update



```python
def replace_linear_with_lora(model, rank, alpha):
    for name, module in model.named_children():
        if isinstance(module, torch.nn.Linear):
            # Replace the Linear layer with LinearWithLoRA
            setattr(model, name, LinearWithLoRA(module, rank, alpha))
        else:
            # Recursively apply the same function to child modules
            replace_linear_with_lora(module, rank, alpha)
```

- We then freeze the original model parameter and use the `replace_linear_with_lora` to replace the said `Linear` layers using the code below
- This will replace the `Linear` layers in the LLM with `LinearWithLoRA` layers

```python
total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters before: {total_params:,}")

for param in model.parameters():
    param.requires_grad = False

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable parameters after: {total_params:,}")
```

```
Total trainable parameters before: 124,441,346
Total trainable parameters after: 0
```

```python
replace_linear_with_lora(model, rank=16, alpha=16)

total_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f"Total trainable LoRA parameters: {total_params:,}")
```

```
Total trainable LoRA parameters: 2,666,528
```

Part 2:
How to deploy a Large Language Model?

- Models
- Tokenizers
- Datasets
- Fine-tuning
- Inference

MODELS

Three options

(1) **Inference-as-a-service**: through an API
(2) **On the cloud**: as managed service or custom deployment
(3) **Locally**: possible for small enough models (Ollama)

Hugging Face enables all three options!

HTTPS://HUGGINGFACE.CO/

It is a primarily a GitHub for models, but also develops a lot of useful packages and resources!

Architectures

- **CPUs:** While generally slower for LLMs, they are more accessible and cost-effective for smaller models or less demanding tasks.
- **GPUs:** The most common choice for LLMs, offering significant performance improvements due to their parallel processing capabilities.
- **TPUs:** Google's specialized hardware designed for machine learning, providing even faster performance than GPUs for certain models.
- **Distributed Systems:** Multiple processors (CPUs, GPUs, or TPUs) working together to handle large models or high inference demands. Use accelerate: https://huggingface.co/docs/accelerate/index
- **Edge Devices:** Smaller, less powerful devices like smartphones and IoT devices can run optimized LLMs for specific tasks.

How to get GPU resources *for free*?

Anyone: Google colab, Kaggle, Codesphere, Sagemaker…

Locally: LaBRI has some (small-ish) GPU servers

Academics:

- grid5000 https://www.grid5000.fr/w/Grid5000:Home
- Jean-Zay https://www.edari.fr/

How much memory for a 3B model?

The memory required to hold a 3B parameter LLM in memory depends heavily on the **data type** used to store the model weights:

Full Precision (FP32):

- Each parameter requires 32 bits (4 bytes)
- Total memory: 3 billion parameters * 4 bytes/parameter = 12 GB

Heavily Quantized (INT4):

- Each parameter requires 4 bits (0.5 bytes)
- Total memory: 3 billion parameters * 0.5 bytes/parameter = 1.5 GB

For comfortable performance and headroom, **8 GB or more** is recommended.

Pipeline and tasks

Tasks

1. Text Classification
   - Sentiment analysis: Determining the emotional tone of a text (positive, negative, neutral).
   - Topic classification: Categorizing text into predefined topics.
   - Spam detection: Identifying unsolicited or unwanted messages.
   - Natural language inference: Determining the relationship between two sentences (entailment, contradiction, neutral).
2. Token Classification
   - Named entity recognition (NER): Identifying and classifying named entities in text (people, organizations, locations, etc.).
   - Part-of-speech (POS) tagging: Assigning grammatical tags to words (noun, verb, adjective, etc.).
3. Question Answering
   - Extractive question answering: Finding the answer to a question within a given text.
   - Multiple choice question answering: Selecting the best answer from a set of options.

More tasks

4. Text Generation

 - Text summarization: Generating a concise summary of a longer text.
 - Translation: Translating text from one language to another.
 - Dialogue generation: Creating conversational responses in a chatbot.
 - Code generation: Generating code in various programming languages.

5. Text2Text Generation

 - Paraphrasing: Rewriting a text while preserving its meaning.
 - Summarization: Generating a concise summary of a longer text.
 - Translation: Translating text from one language to another.

6. Fill-Mask

 - Masked language modeling: Predicting missing words in a text.

7. Feature Extraction

 - Generating embeddings: Creating numerical representations of text for use in other machine learning tasks.

Pipeline: concise but little control

```python
from transformers import pipeline

# Create a sentiment analysis pipeline
classifier = pipeline(task="sentiment-analysis",
                      model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")

# Run inference
result = classifier("This course is f***ing great!")

# Print the result
print(result)
```

```
[{'label': 'POSITIVE', 'score': 0.9998470544815063}]
```

You can choose the "device" (CPU, GPU)

```python
from transformers import pipeline

# Create a sentiment analysis pipeline
classifier = pipeline(task="sentiment-analysis",
                      model="distilbert/distilbert-base-uncased-finetuned-sst-2-english",
                      device="cuda")

# Run inference
result = classifier("This course is f***ing great!")

# Print the result
print(result)
```

The same model can be used for different tasks!

!! IMPORTANT !! Set `torch_dtype="auto"` to load the weights in the data type defined in a model's `config.json` file to automatically load the most memory-optimal data type.

```
from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(model="distilbert/distilbert-base-uncased",
                                                           torch_dtype="auto")
```

```
from transformers import AutoModelForTokenClassification

model = AutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased",
                                                         num_labels = 5,
                                                        torch_dtype="auto")
```

WTF?

```python
from transformers import AutoModelForTokenClassification

model = AutoModelForTokenClassification.from_pretrained("distilbert/distilbert-base-uncased",
                                                        num_labels = 5,
                                                       torch_dtype="auto")
```

config.json: 100% 483/483 [00:00<00:00, 9.73kB/s]

model.safetensors: 100% 268M/268M [00:05<00:00, 51.5MB/s]

```
Some weights of DistilBertForTokenClassification were not initialized from the model checkpoint at distilbert/dist
ilbert-base-uncased and are newly initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.
```

Hugging Face: “Don’t worry, this is completely normal! The pretrained head of the BERT model is discarded, and replaced with a randomly initialized classification head. You will fine-tune this new model head on your sequence classification task, transferring the knowledge of the pretrained model to it.”

Serverless inference API: slow but free

```
from huggingface_hub import InferenceClient

client = InferenceClient(
    "cardiffnlp/twitter-roberta-base-sentiment-latest",
    token="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",
)

client.text_classification("Today is a great day")
```

```
[TextClassificationOutputElement(label='positive', score=0.9836677312850952),
 TextClassificationOutputElement(label='neutral', score=0.011358872987329960),
 TextClassificationOutputElement(label='negative', score=0.004973393864929676)]
```

Pipeline is good to get started

But soon you feel limited.

Let’s see how to get a bit more control!

Tokenizers

Only dealing with text here...

The tokenizer is for dealing with texts. For other formats:

- Speech and audio, use a Feature extractor to extract sequential features from audio waveforms and convert them into tensors.
- Image inputs use a ImageProcessor to convert images into tensors.
- Multimodal inputs, use a Processor to combine a tokenizer and a feature extractor or image processor.

```python
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("google-bert/bert-base-uncased")
tokenizer
```

```
BertTokenizerFast(name_or_path='google-bert/bert-base-uncased', vocab_size=30522, model_max_length=512, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pad_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True),  added_tokens_decoder={
	0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
	100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
	101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
	102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
	103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
}
```

```python
sequence = "In a hole in the ground there lived a hobbit."
print(tokenizer(sequence))
```

```
{'input_ids': [101, 1999, 1037, 4920, 1999, 1996, 2598, 2045, 2973, 1037, 7570, 10322, 4183, 1012, 102], 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

The nasty business of data preprocessing

You want to create batches (a lot more efficient!). This means that you may need to:

- Pad: add a special token [PAD] to make sure all inputs have the same size
- Truncate: if some inputs are larger than the context length of your model, you need to break them up into more inputs (but it’s not that obvious: better introduce some overlapping!)

Good news: Tokenizer does that for you!

```python
batch_sentences = [
    "But what about second breakfast?",
    "Don't think he knows about second breakfast, Pip.",
    "What about elevensies?",
]
encoded_input = tokenizer(batch_sentences, padding=True, truncation=True)
print(encoded_input)
```

```
{'input_ids': [[101, 2021, 2054, 2055, 2117, 6350, 1029, 102, 0, 0, 0, 0, 0, 0], [101, 2123, 1005, 1056, 2228, 200
2, 4282, 2055, 2117, 6350, 1010, 28315, 1012, 102], [101, 2054, 2055, 5408, 14625, 1029, 102, 0, 0, 0, 0, 0, 0,
0]], 'token_type_ids': [[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]], 'attention_mask': [[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, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0]]}
```

Datasets

```python
from datasets import load_dataset

dataset = load_dataset("yelp_review_full")
dataset["train"][100]
```

```
{'label': 0,
 'text': 'My expectations for McDonalds are t rarely high. But for one to still fail so spectacularly...that takes
something special!\\nThe cashier took my friends\'s order, then promptly ignored me. I had to force myself in fron
t of a cashier who opened his register to wait on the person BEHIND me. I waited over five minutes for a gigantic
order that included precisely one kid\'s meal. After watching two people who ordered after me be handed their foo
d, I asked where mine was. The manager started yelling at the cashiers for \\"serving off their orders\\" when the
y didn\'t have their food. But neither cashier was anywhere near those controls, and the manager was the one servi
ng food to customers and clearing the boards.\\nThe manager was rude when giving me my order. She didn\'t make sur
e that I had everything ON MY RECEIPT, and never even had the decency to apologize that I felt I was getting poor
service.\\nI\'ve eaten at various McDonalds restaurants for over 30 years. I\'ve worked at more than one location.
I expect bad days, bad moods, and the occasional mistake. But I have yet to have a decent experience at this stor
e. It will remain a place I avoid unless someone in my party needs to avoid illness from low blood sugar. Perhaps
I should go back to the racially biased service of Steak n Shake instead!'}
```

Fine-tuning

TrainingArguments

```
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="checkpoints",
    learning_rate=2e-5,
    eval_strategy="epoch",
    save_strategy="epoch",
    logging_strategy="epoch",
    per_device_train_batch_size=32,
    per_device_eval_batch_size=16,
    num_train_epochs=10,
    weight_decay=0.01,
    report_to="none",
)
```

Trainer

```
from transformers import Trainer

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=small_train_dataset,
    eval_dataset=small_eval_dataset,
    compute_metrics=compute_metrics,
)
```

```
trainer.train()
```

LORA

Load models, two options

Option 1: load a PEFT adapter

```
from transformers import AutoModelForCausalLM, AutoTokenizer

peft_model_id = "ybelkada/opt-350m-lora"
model = AutoModelForCausalLM.from_pretrained(peft_model_id)
```

Option 2: Load the model and its adapter

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_id = "facebook/opt-350m"
peft_model_id = "ybelkada/opt-350m-lora"

model = AutoModelForCausalLM.from_pretrained(model_id)
model.load_adapter(peft_model_id)
```

```python
from transformers import AutoModelForSeq2SeqLM

model = AutoModelForSeq2SeqLM.from_pretrained("bigscience/mt0-small")
```

```python
from peft import LoraConfig

peft_config = LoraConfig(
    lora_alpha=16,
    lora_dropout=0.1,
    r=64,
    bias="none",
    task_type="CAUSAL_LM",
)
```

```python
from peft import get_peft_model

model = get_peft_model(model, peft_config)
model.print_trainable_parameters()
```

```
trainable params: 2,752,512 || all params: 302,929,280 || trainable%: 0.9086
```

The rest is as before: TrainingArguments and Trainer

Part 3: Advanced topics

- Retrieval-augmented Generation (RAG)
- Agents

Retrieval-augmented Generation (RAG)

Agents