

Large Language Models

Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux



LaBRI

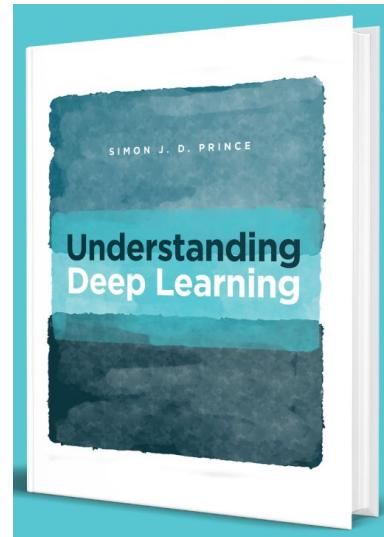
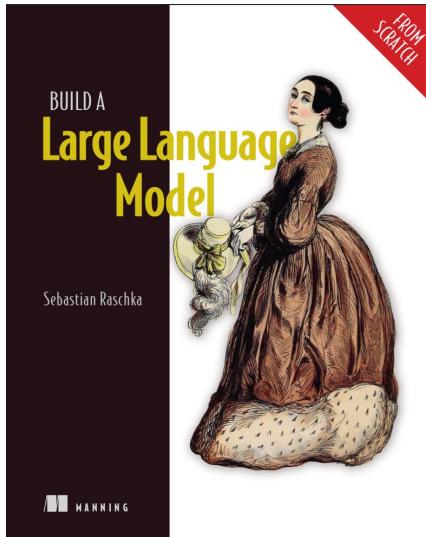


GOAL OF THIS COURSE

- Understand LLMs from a mathematical and practical point of view
 - Being able to **program** from scratch an LLM
 - Understand how LLMs can generate **code** and **proofs**
-

SOME REFERENCES

- [Build a Large Language Model](#) by Sebastian Raschka
- [minGPT / nanoGPT](#) (and videos) by Andrej Karpathy
- [Understanding Deep Learning](#) by Simon Price



Most illustrations in these slides are from the “Build a Large Language Model” book, copyright Sebastian Raschka 2024

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

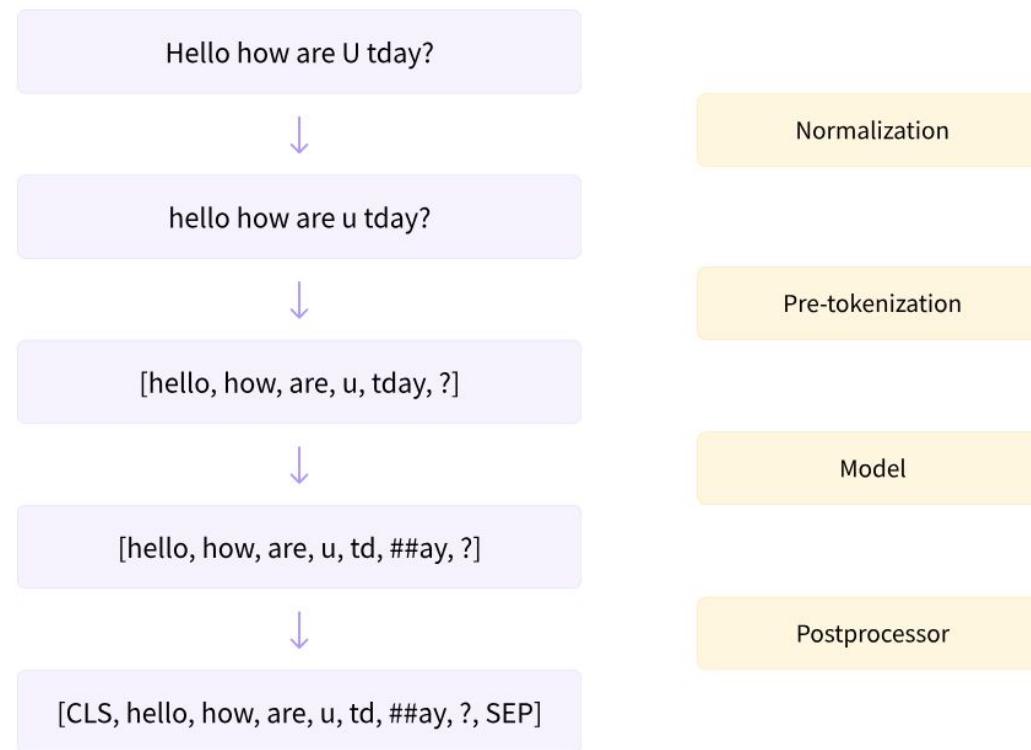
`context_length`: number of tokens considered for prediction

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')
```

TOKENIZATION IS IMPORTANT, WE'LL TALK ABOUT IT LATER!

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

GPT-2 has 50,257 tokens

EMBEDDINGS AND THE MULTI-LAYER PERCEPTRON

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

Réjean Ducharme

Pascal Vincent

Christian Jauvin

Département d’Informatique et Recherche Opérationnelle

Centre de Recherche Mathématiques

Université de Montréal, Montréal, Québec, Canada

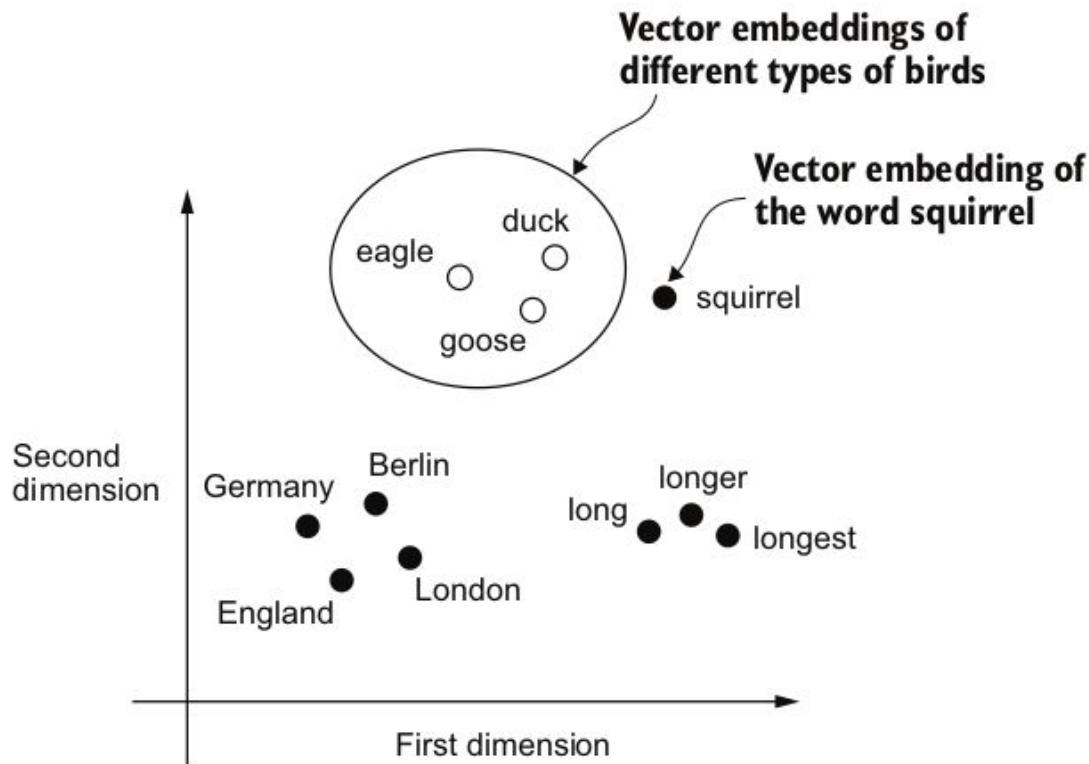
BENGIOY@IRO.UMONTREAL.CA

DUCHARME@IRO.UMONTREAL.CA

VINCENTP@IRO.UMONTREAL.CA

JAUVINC@IRO.UMONTREAL.CA

KEY IDEA: EMBEDDINGS



NN.EMBEDDING

```
import torch
import torch.nn as nn

n_token = 3
n_embed = 4

embedding = torch.nn.Embedding(n_token, n_embed)
print("Weights of the embedding:\n", embedding.weight)
print("Result of embedding token number 1:\n", embedding(torch.tensor([1])))
```

Weights of the embedding:

Parameter containing:

```
tensor([[-0.9252,  0.8805, -0.0214,  0.9724],
       [ 0.1136,  0.2035,  1.1415,  0.0875],
       [ 0.4177,  0.6348,  0.6271,  0.1938]], requires_grad=True)
```

Result of embedding token number 1:

```
tensor([[0.1136, 0.2035, 1.1415, 0.0875]], grad_fn=<EmbeddingBackward0>)
```

Advanced question: what is the difference between nn.embedding and nn.linear?

WHAT IS THE DIFFERENCE BETWEEN NN.EMBEDDING AND NN.LINEAR?

Both `nn.Embedding` and `nn.Linear` are modules in PyTorch that deal with transforming inputs, but they serve different purposes and operate differently:

`nn.Embedding`

- **Purpose:** This module is used to represent categorical data, such as words in a vocabulary. It creates a lookup table where each unique category (e.g., word) is assigned a unique vector (embedding).
- **Operation:** It works by looking up the embedding vector corresponding to the given input index. It's essentially a dictionary that maps indices to vectors.

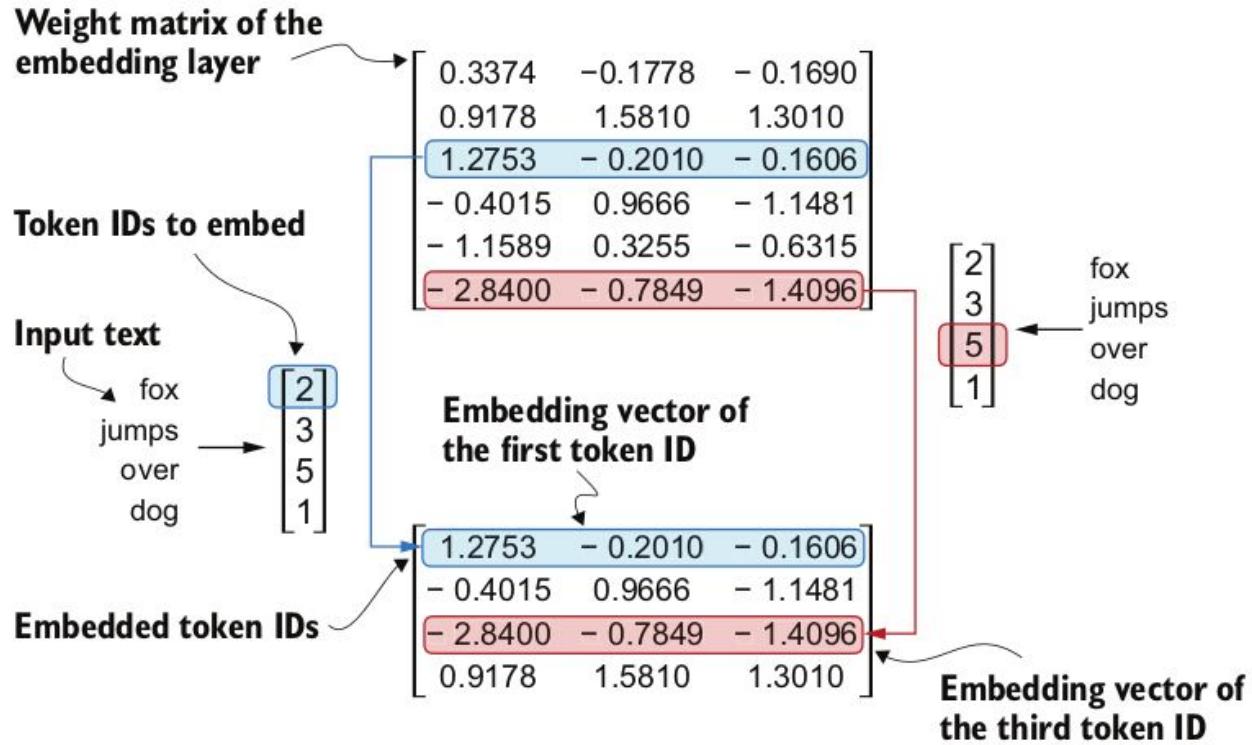
`nn.Linear`

- **Purpose:** This module performs a linear transformation on the input data. It applies a weight matrix and a bias vector to the input.
- **Operation:** It calculates the dot product of the input with the weight matrix and adds the bias vector. This is a fundamental operation in many neural networks.

In Summary:

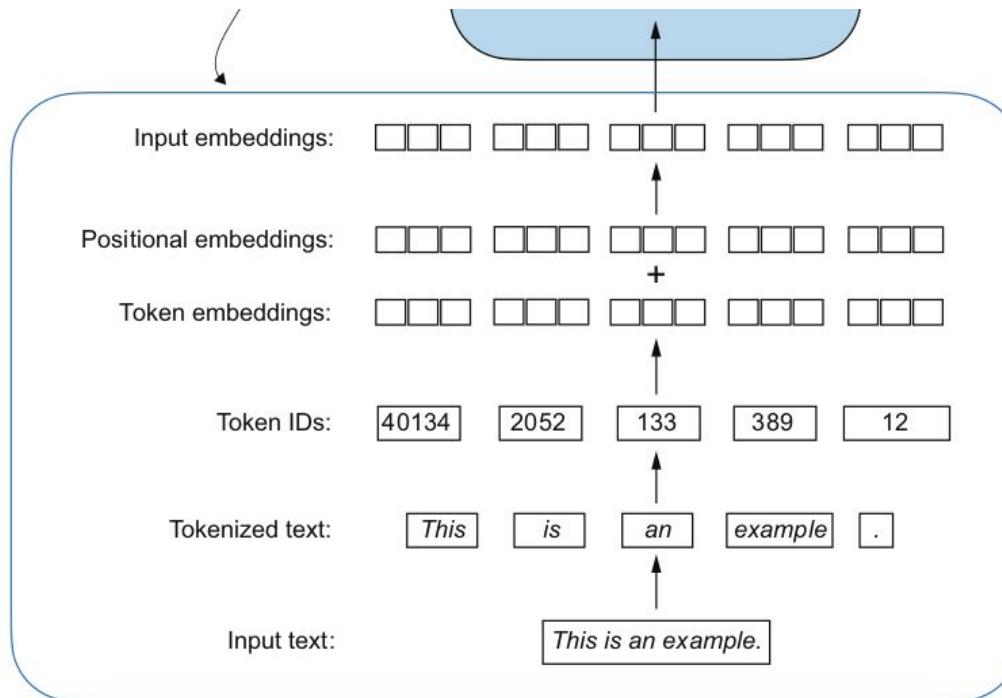
- Use `nn.Embedding` for representing categorical data as dense vectors.
- Use `nn.Linear` for performing linear transformations in neural networks.

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.

(We will discuss later *positional embeddings*.)



STATISTICS

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

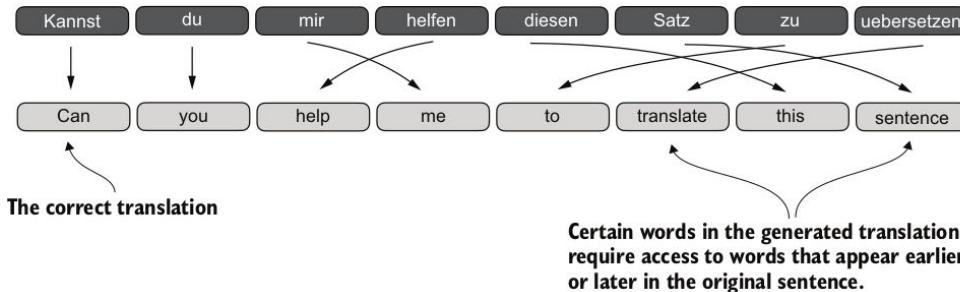
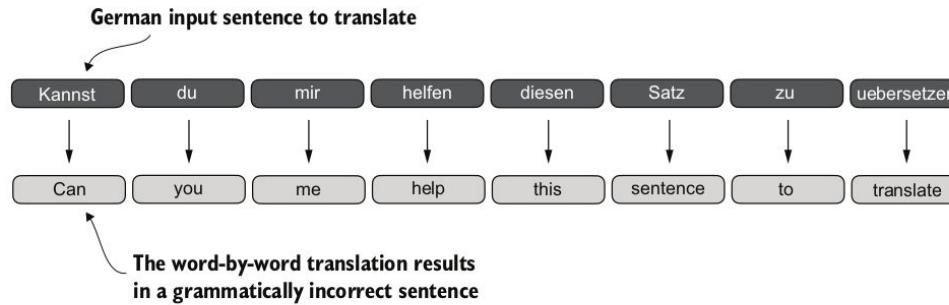
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

MULTI-LAYER PERCEPTRON (MLP)

```
class MLP(nn.Module):
    def __init__(self, context_length, n_embed, n_hidden):
        super().__init__()
        self.token_embedding_table = nn.Embedding(n_token, n_embed)
        self.net = nn.Sequential(
            nn.Linear(context_length * n_embed, n_hidden),
            nn.Tanh(),
            nn.Linear(n_hidden, n_token)
    )
```

TWO ISSUES WITH MLPs

- We cannot have long contexts
- Struggle with long dependencies



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

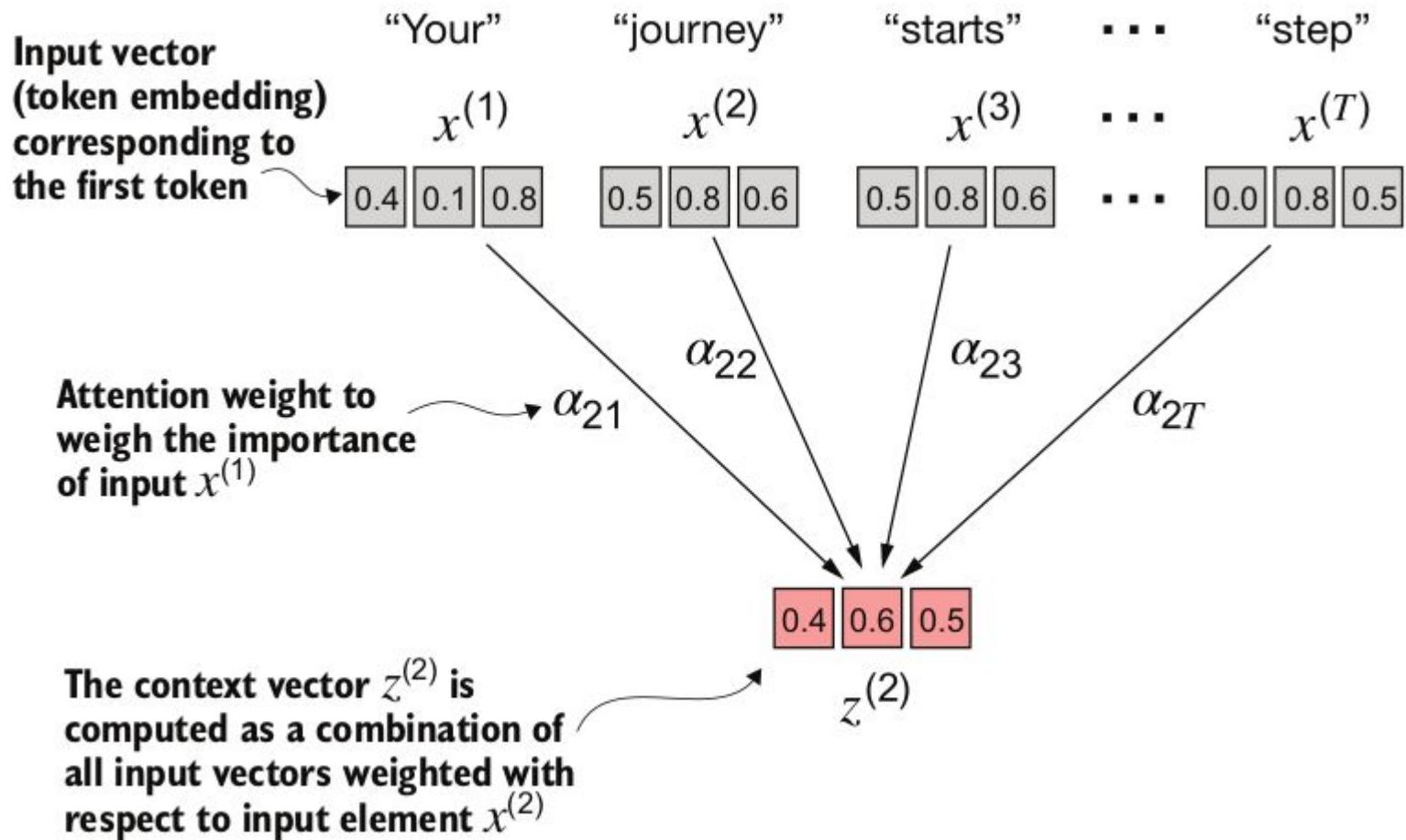
Input: an embedding vector $x(i)$ for each token i

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

Intuition: $z(i)$ gathers contextual information

COMPUTING CONTEXT VECTORS

This is very easy assuming we have computed **attention weights**: $\text{alpha}(i,j)$ describes the importance of token j for token i .



JUST A MATRIX MULTIPLICATION...

```
context_length = 3
embed_dim = 2

x = torch.randn(context_length, embed_dim)
attention_weights = torch.randn(context_length, context_length) # We'll discuss later how to compute them

context_vectors = attention_weights @ x
```

COMPUTING ATTENTION SCORES AND WEIGHTS

Now we focus on the core computation: attention scores and weights.

We first compute **attention scores**, and then normalise them into **attention weights**.

KEYS, QUERIES, AND VALUES

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

Output: for each token i :

- A query vector $q(i)$, describing the information token i is interested in,
- A key vector $k(i)$, whose goal is to match the relevant queries for token i ,
- A value vector $v(i)$, describing the information contained by token i .

INFORMATION-RETRIEVAL INTUITION

Think of a database, it holds (**keys**, **values**), and it can be accessed through **queries**.

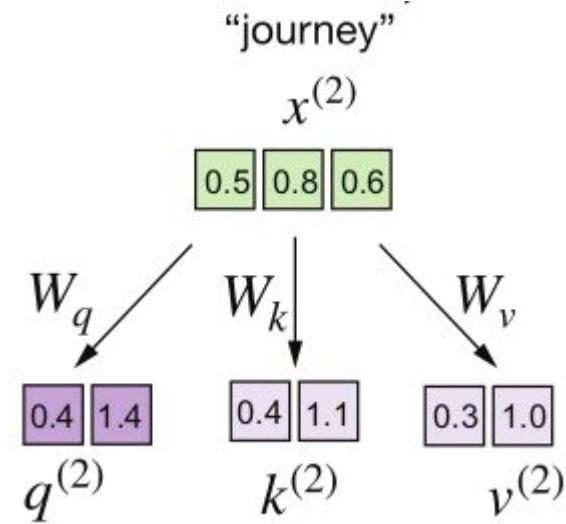
Here, to match a query with a key we simply do a dot-product.

So: the attention score $\alpha(i, j)$ is defined as the dot-product between $q(i)$ and $k(j)$

KEYS, QUERIES, AND VALUES ARE COMPUTED BY MATRIX MULTIPLICATIONS

We introduce three matrices with trainable parameters:

- W_q for query,
- W_k for key,
- W_v for value.



FROM ATTENTION SCORES TO ATTENTION WEIGHTS

Attention scores are computed by a single matrix multiplication:

query @ key.T

Now, how do we normalise scores into weights?

SOFTMAX IS VECTOR NORMALISATION

```
context_length = 5

attention_scores = torch.randn(context_length)
print("The attention scores: \n", attention_scores)
scores_exped = attention_scores.exp()
print("After exponentiation: \n", scores_exped)
probs = scores_exped / scores_exped.sum()
print("After normalisation: \n", probs)
print("\nThe two steps above are called softmax: \n", torch.softmax(attention_scores, -1))
```

The attention scores:

```
tensor([ 1.4529,  0.3491, -0.8928,  0.2072, -0.3993])
```

After exponentiation:

```
tensor([4.2757, 1.4177, 0.4095, 1.2302, 0.6708])
```

After normalisation:

```
tensor([0.5342, 0.1771, 0.0512, 0.1537, 0.0838])
```

The two steps above are called softmax:

```
tensor([0.5342, 0.1771, 0.0512, 0.1537, 0.0838])
```

WE HAVE TO BE CAREFUL WITH SOFTMAX

It is a classical story in Deep Learning: values should be kept in a reasonable range to avoid vanishing or exploding gradients.

A second reason is softmax sensitivity to large numbers:

```
torch.softmax(torch.tensor([0.1, -0.2, -0.3, 0.2, 0.5]), dim=-1)
```

```
tensor([0.1997, 0.1479, 0.1338, 0.2207, 0.2979])
```

```
torch.softmax(torch.tensor([0.1, -0.2, -0.3, 0.2, 0.5])*10, dim=-1)
```

```
tensor([1.7128e-02, 8.5274e-04, 3.1371e-04, 4.6558e-02, 9.3515e-01])
```

SCALED SELF-ATTENTION

Assume u, v are vectors of dimension d :

$$u, v \sim N(0, 1)$$

What is the distribution of $u \cdotdot v$?

Answer: $\text{Exp}[u \cdotdot v] = 0$ but $\text{Var}(u \cdotdot v) = d$

But: $\text{Var}(u \cdotdot v / \sqrt{d}) = 1$

SELF-ATTENTION HEAD

```
x = torch.randn(context_length, input_dim)

key = nn.Linear(input_dim, head_dim, bias=False)
query = nn.Linear(input_dim, head_dim, bias=False)
value = nn.Linear(input_dim, output_dim, bias=False)

k = key(x)
q = query(x)
v = value(x)

attention_scores = q @ k.T
attention_weights = torch.softmax(attention_scores * head_dim**-0.5, dim=-1)
context_vectors = attention_weights @ v
```

AS A NN MODULE

```
class Head(nn.Module):
    def __init__(self, context_length, head_input_dim, head_size, head_output_dim):
        super().__init__()
        self.key = nn.Linear(head_input_dim, head_size, bias=False)
        self.query = nn.Linear(head_input_dim, head_size, bias=False)
        self.value = nn.Linear(head_input_dim, head_output_dim, bias=False)

    def forward(self, x):
        B, T, C = x.shape
        # if training: B = batch_size, else B = 1
        # T = context_length
        # I = head_input_dim
        # H = head_size
        # O = head_output_dim

        k = self.key(x)    # (B, T, H)
        q = self.query(x) # (B, T, H)
        v = self.value(x) # (B, T, O)
        attention_scores = q @ k.transpose(1,2) # (B, T, H) @ (B, H, T) -> (B, T, T)
        attention_weights = torch.softmax(attention_scores * self.head_size**-0.5, dim=-1) # (B, T, T)
        context_vectors = attention_weights @ v # (B, T, T) @ (B, T, O) -> (B, T, O)
        return context_vectors
```

THE POWER OF PYTORCH BROADCASTING SEMANTICS

Did you notice that multiplying a tensor (B, T, H) with another one (B, H, T) yields a tensor (B, T, T) ?

This is called broadcasting semantics:

<https://pytorch.org/docs/stable/notes/broadcasting.html>

COMPLEXITY

C = context_length

I = input_dim

H = head_dim

O = output_dim

- key(x): $(C \times I) \times (I \times H) \rightarrow C \times H$
- query(x): $(C \times I) \times (I \times H) \rightarrow C \times H$
- value(x): $(C \times I) \times (I \times O) \rightarrow C \times O$
- attention_scores: $(C \times H) \times (H \times C) \rightarrow C \times C$
- context_vectors: $(C \times C) \times (C \times O) \rightarrow C \times O$

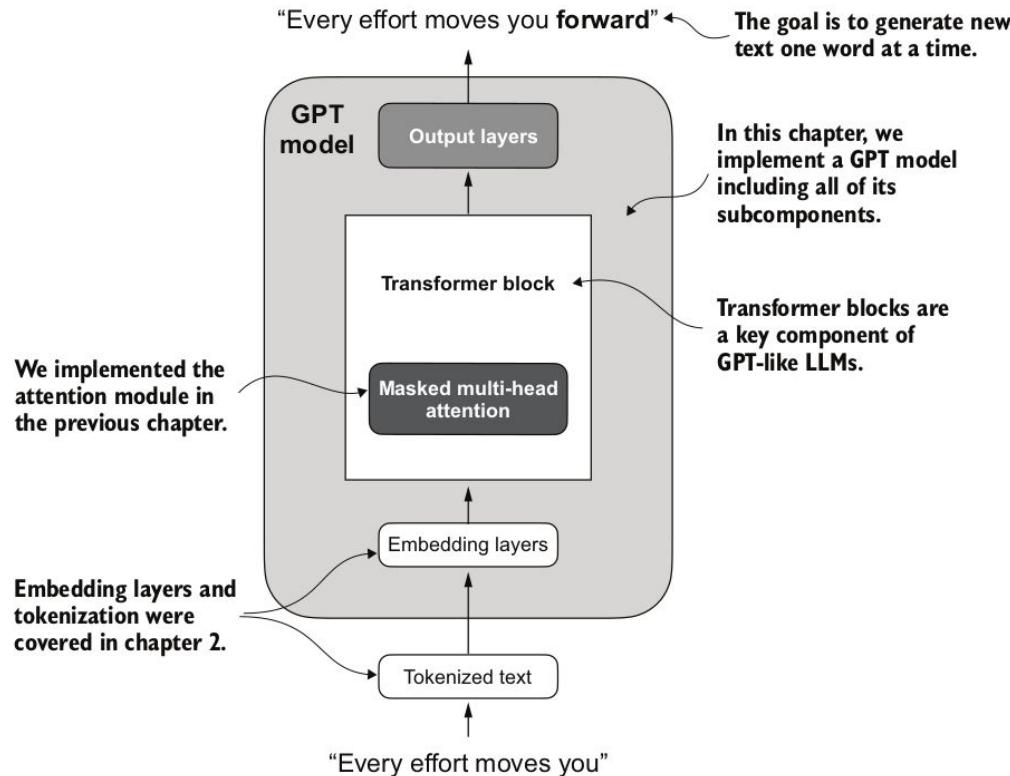
The memory footprint is quadratic in context length!

IMPORTANT

The matrices for computing keys, queries, and values include **trainable** parameters, so the attention mechanism **learns** where to put attention in a **data-driven way**.

BUT: the three matrices are the same for all indices! In other words, the attention mechanism is not aware of positions (neither absolute nor relative).

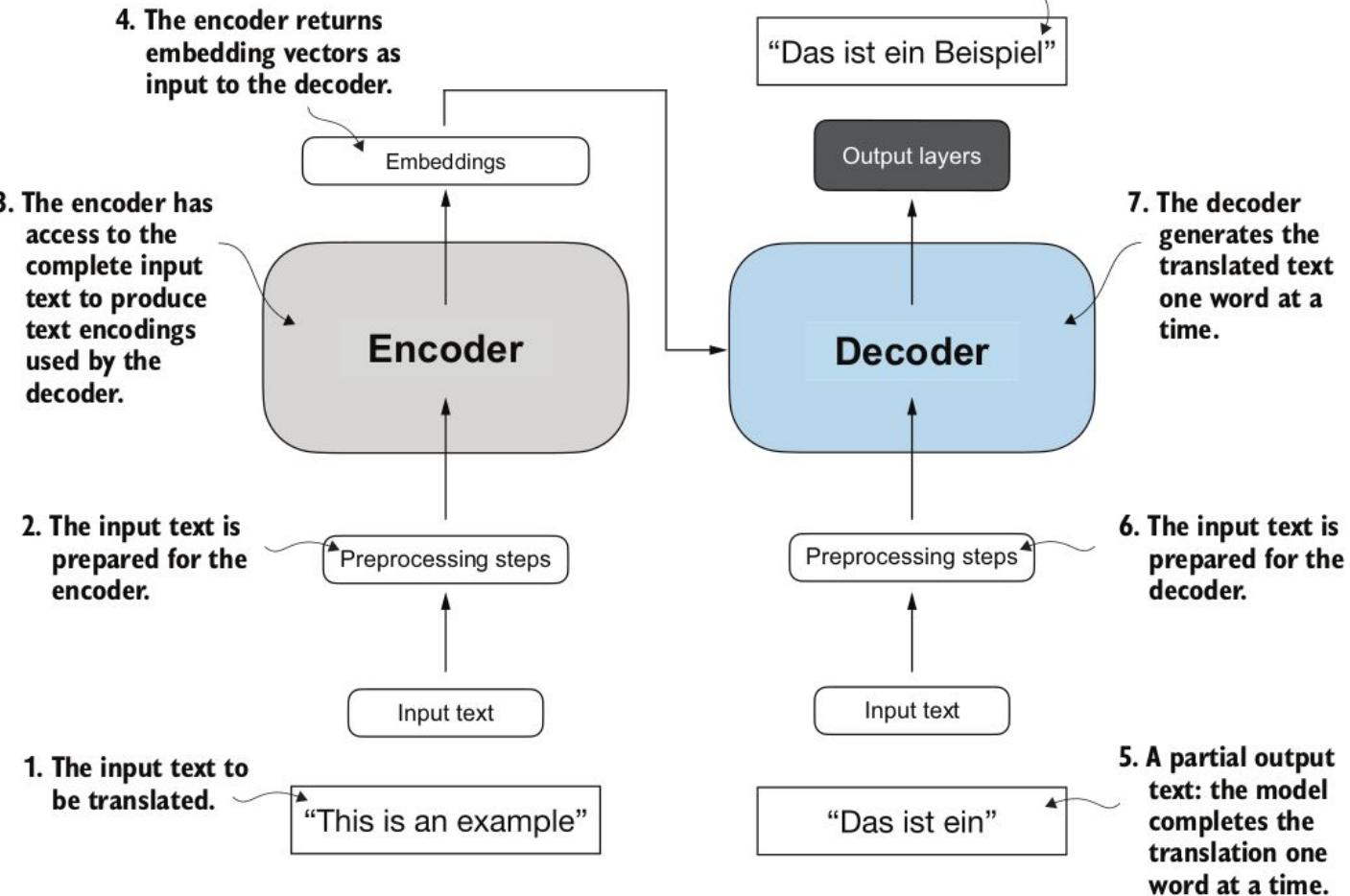
ATTENTION HEADS AS KEY COMPONENTS IN A TRANSFORMER



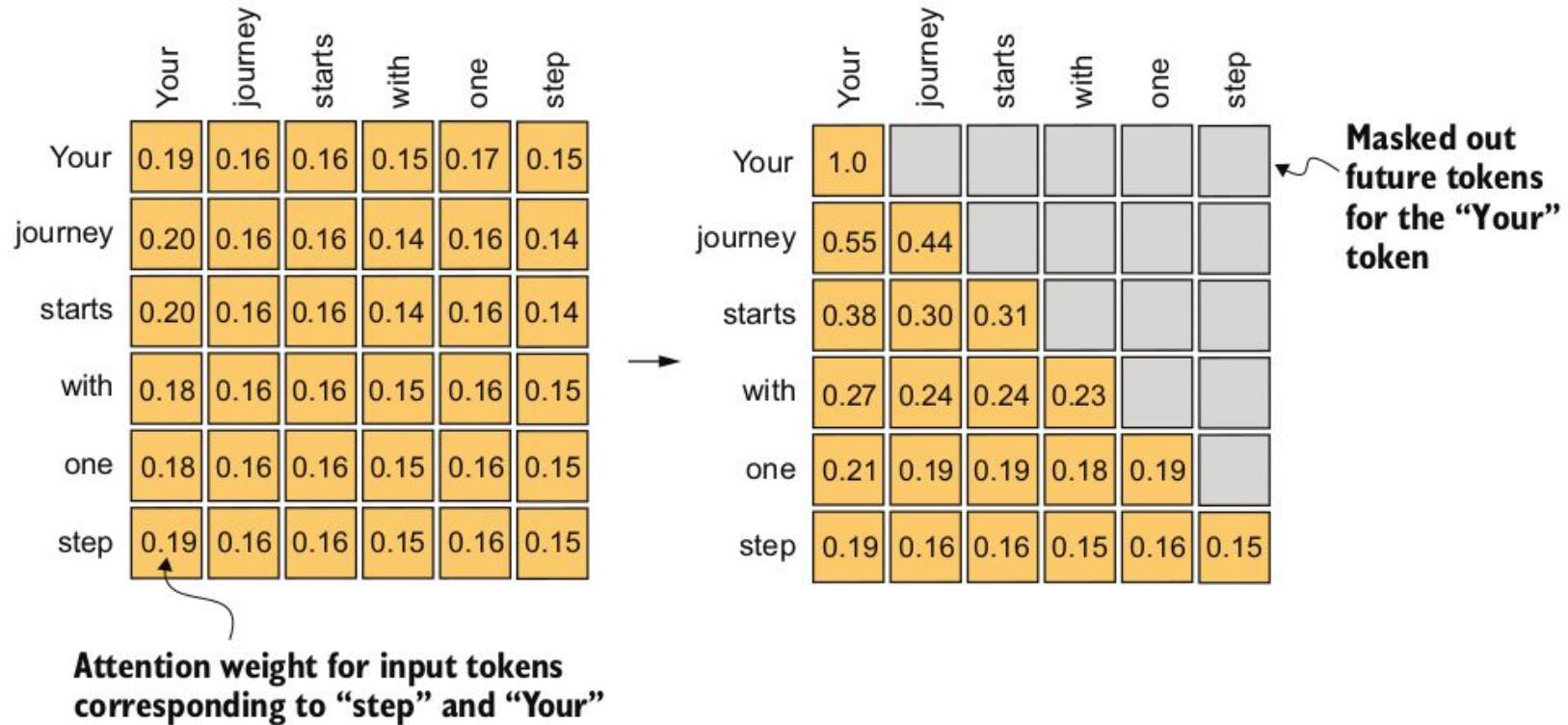
ENCODER / DECODER



8. The complete output (translation)



DECODERS USE CAUSAL ATTENTION



IMPLEMENTATION OF THE MASK

```
x = torch.randn(context_length, input_dim)

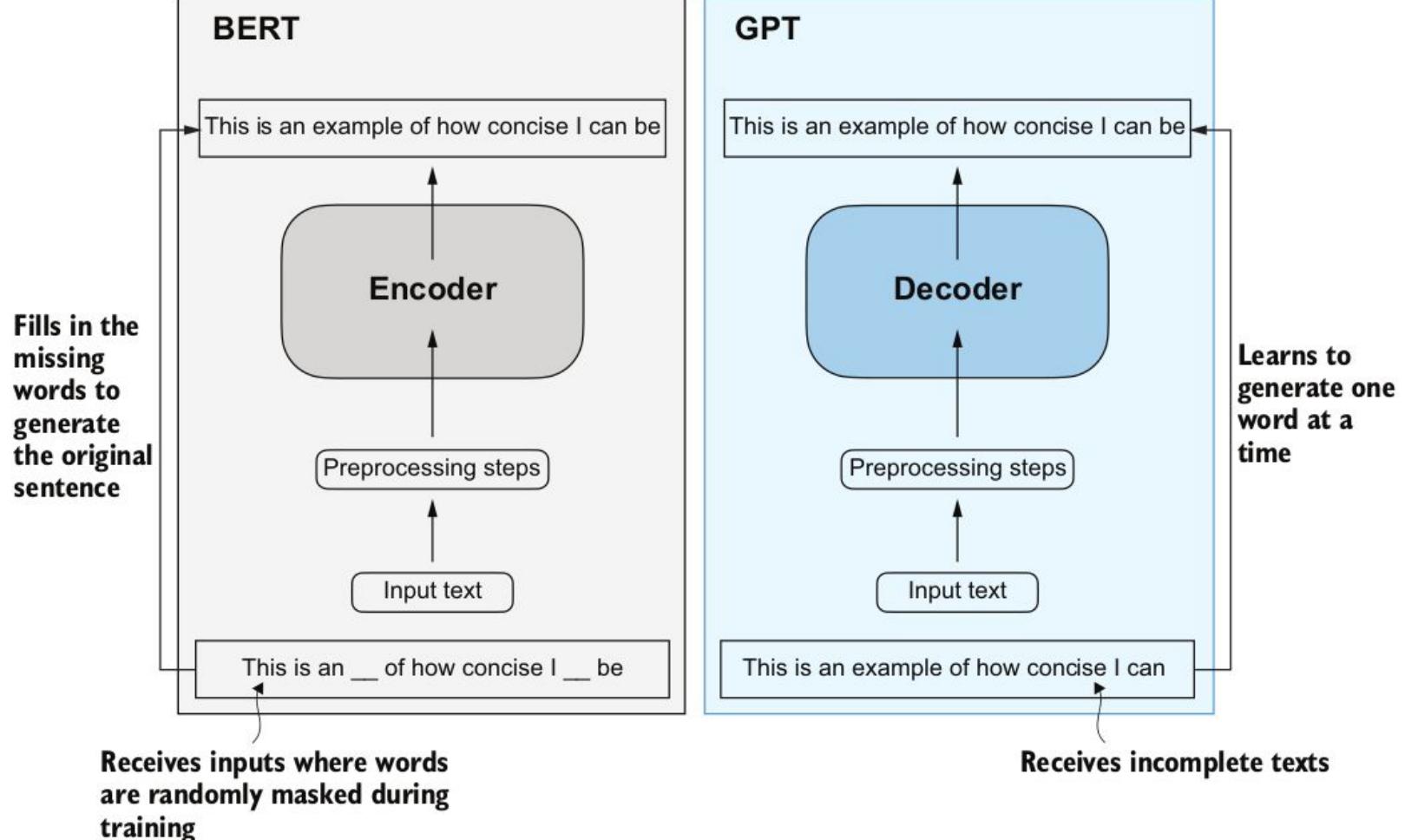
key = nn.Linear(input_dim, head_dim, bias=False)
query = nn.Linear(input_dim, head_dim, bias=False)
value = nn.Linear(input_dim, output_dim, bias=False)

k = key(x)
q = query(x)
v = value(x)

attention_scores = q @ k.T

mask = torch.triu(torch.ones(context_length, context_length), diagonal=1)
masked_attention_scores = attention_scores.masked_fill(mask.bool(), -torch.inf)

attention_weights = torch.softmax(masked_attention_scores * head_dim**-0.5, dim=-1)
context_vectors = attention_weights @ v
```

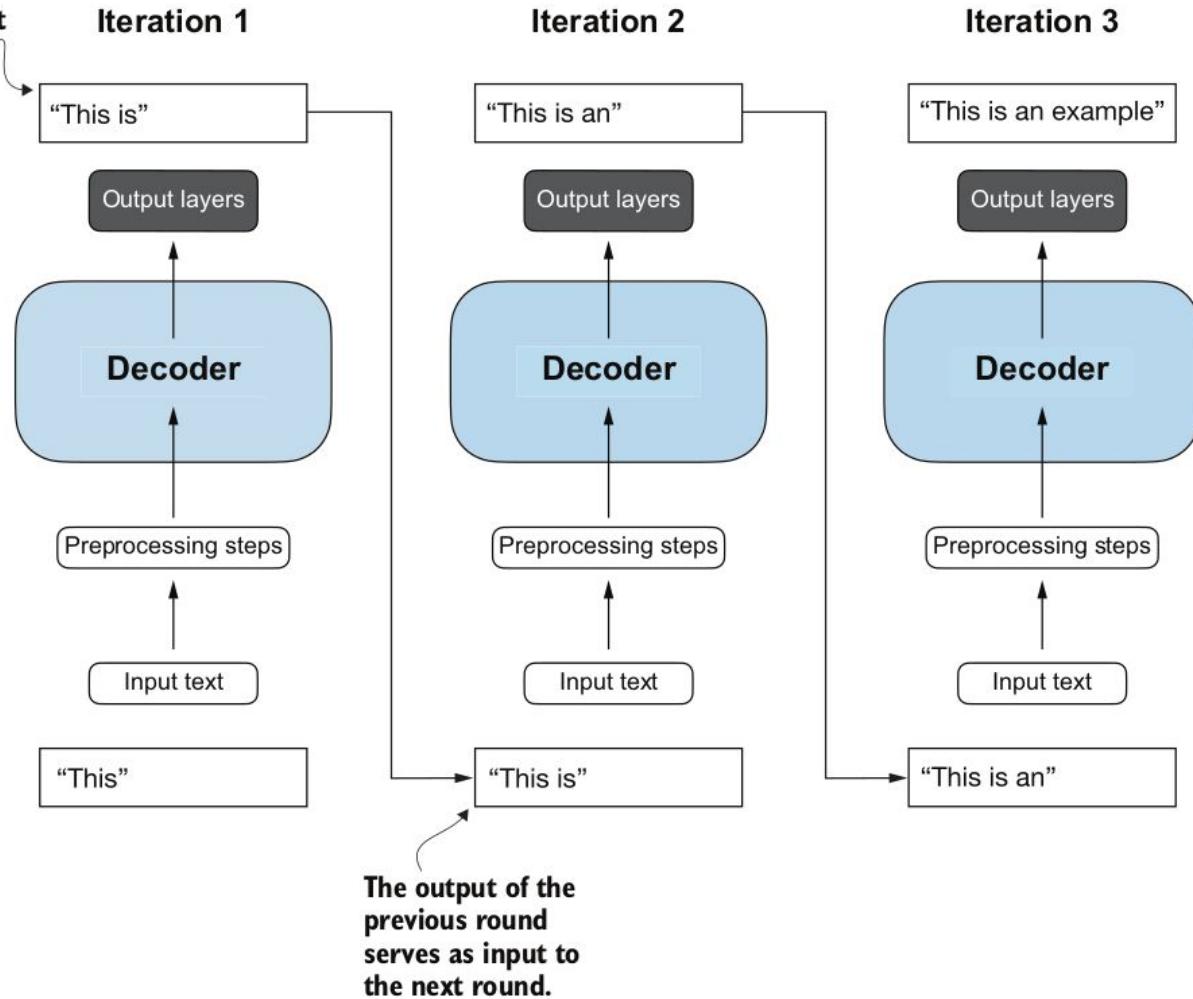


AUTOREGRESSIVE MODELS

AUTOREGRESSIVE

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



SLIDING WINDOWS



SLIDING WINDOWS

Text sample:

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

LLMs learn to predict one word at a time

Input the LLM receives

The LLM can't access words past the target.

Target to predict

IMPORTANT

The Transformer consists of a number of “blocks” and “layers”, each with the same signature:

Input: a sequence of vectors, one for each token

Output: a sequence of vectors, one for each token

WHAT ARE THE BENEFITS OF THE SLIDING WINDOWS?

Fix $c = \text{context_length}$

A single data point (meaning, a sequence of $c+1$ tokens) becomes c data points, for free:

- A single tensor stores all c data points
- Running the model once on the whole sequence yields predictions for all c data points

BATCHING



MODELS' SIGNATURES

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

Output: $\text{model}(x, y) = (\text{logits}, \text{loss})$ 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)

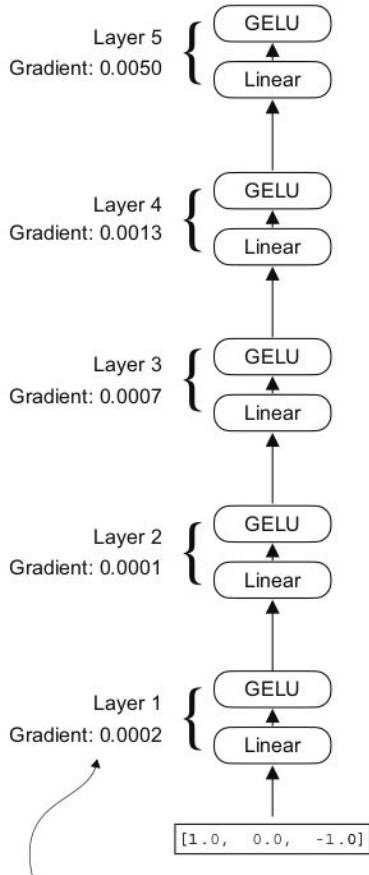
SHORTCUT CONNECTIONS



SHORTCUT CONNECTIONS

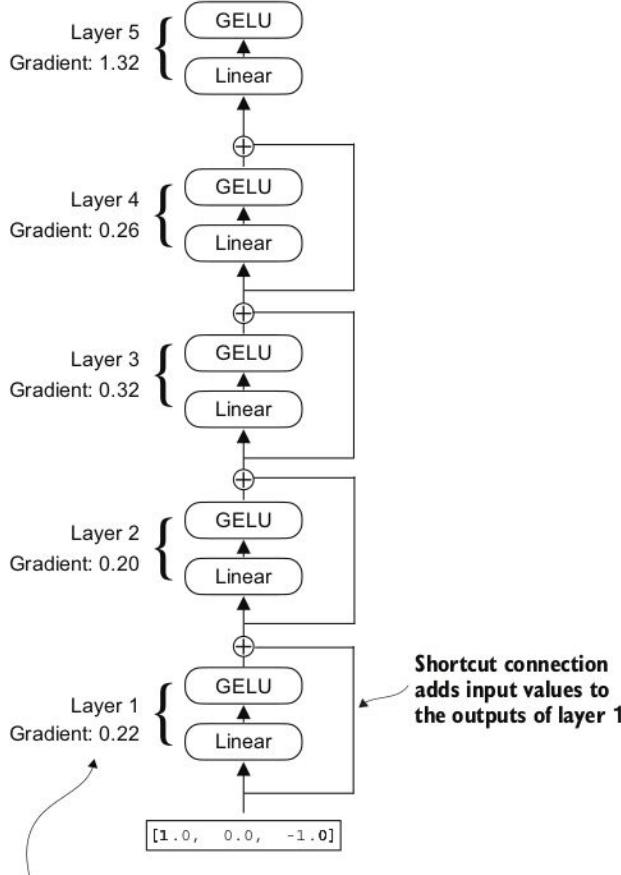
Shortcut connections (also called residual connections / skip connections) provide a pathway for the gradient to flow more easily during backpropagation, mitigating the vanishing gradient problem and enabling the training of much deeper networks.

Deep neural network



In very deep networks, the gradient values in early layers become vanishingly small

Deep neural network with shortcut connections



Shortcut connection adds input values to the outputs of layer 1

The shortcut connections help with maintaining relatively large gradient values even in early layers

DROPOUT



DROPOUT

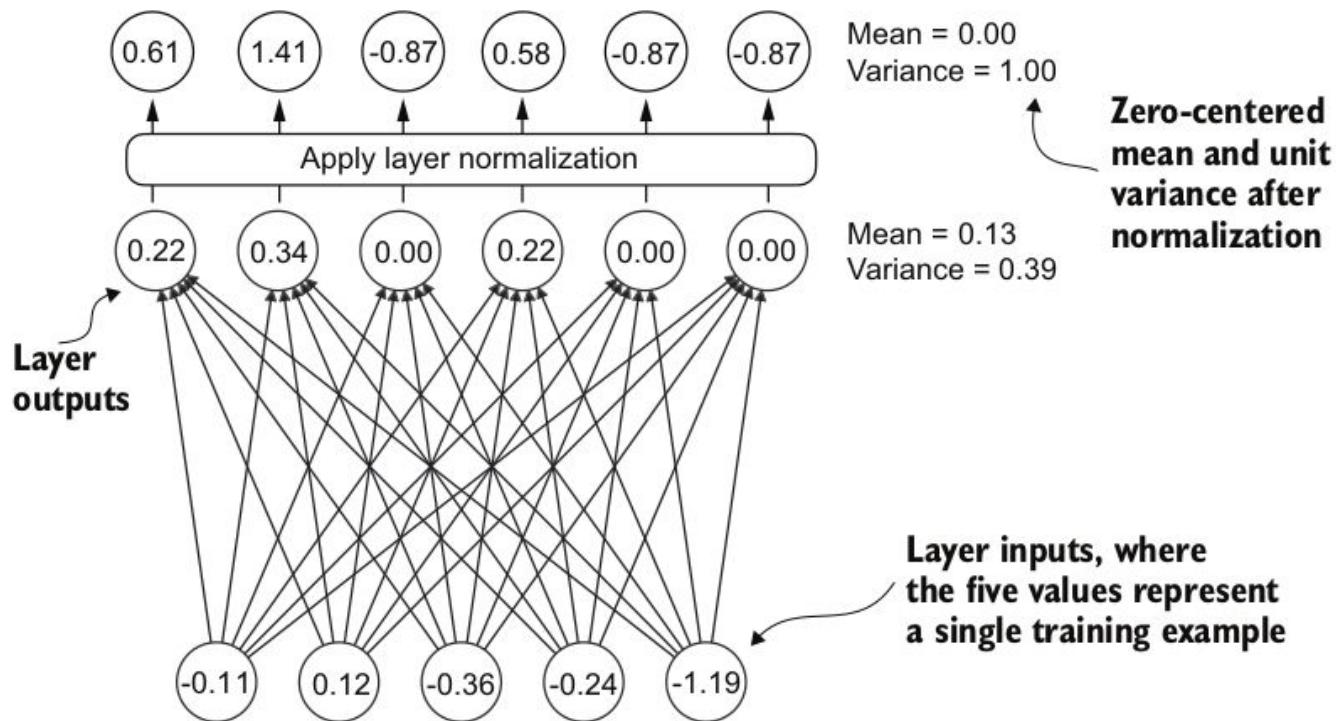
Dropout is a regularization technique used in neural networks to prevent overfitting. It works by randomly dropping out (setting to zero) a certain proportion of neurons in a layer during each training step.

- **Prevents Overfitting:** By randomly dropping out neurons, dropout prevents the network from learning complex co-adaptations that are specific to the training data. This helps the model generalize better to unseen data.
- **Ensemble Effect:** Dropout can be seen as training an ensemble of multiple smaller networks. Each training step effectively samples a different subnetwork. At test time, the average of these subnetworks is used, which improves the overall performance.
- **Reduces Co-adaptation:** Dropout forces neurons to learn more robust features that are not dependent on the presence of specific other neurons. This leads to better feature representations.

LAYER NORMALIZATION



LAYER NORMALIZATION



POSITIONAL EMBEDDINGS

IMPORTANT

The matrices for computing keys, queries, and values include **trainable** parameters, so the attention mechanism **learns** where to put attention in a **data-driven way**.

BUT: the three matrices are the same for all indices! In other words, the attention mechanism is not aware of positions (neither absolute nor relative).

POSITIONAL EMBEDDINGS

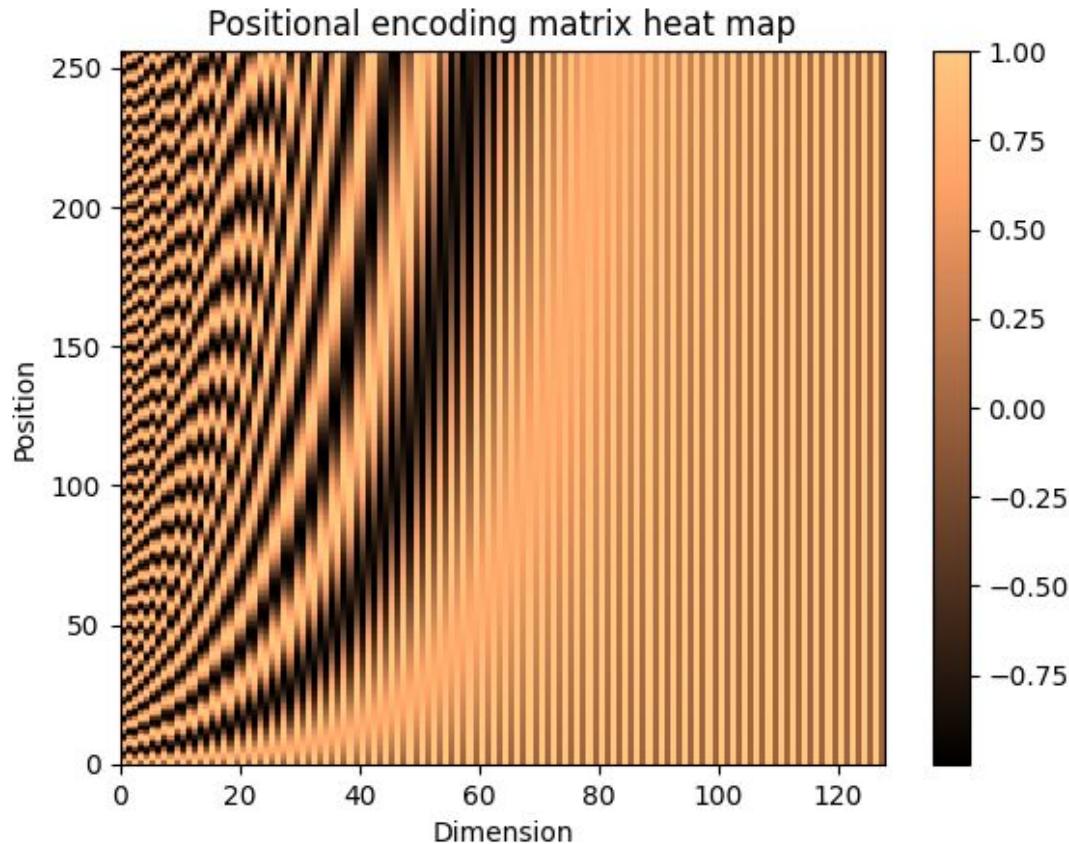
Positional embeddings are added to bring information about position. Two goals:

- The information can be decoded by the model
- The information contains relevant absolute and relative positions

SIMPLEST VERSION: LEARNED POSITIONAL EMBEDDINGS

```
tok_emb = self.token_embedding_table(idx) # (B, T, I)
pos_emb = self.position_embedding_table(torch.arange(T)) # (T, I)
x = tok_emb + pos_emb # (B, T, I)
```

THE ORIGINAL POSITIONAL EMBEDDING



THE FORMULA

Fix $c = \text{context_length}$ and $d = \text{input_dim}$

The positional embedding is a vector $p : (c, d)$

- $p(\text{pos}, 2t) = \sin(\text{pos} / 10\text{e}300^{2t/d})$
- $p(\text{pos}, 2t+1) = \cos(\text{pos} / 10\text{e}300^{2t/d})$

THE MORE RECENT ROPE (ROTARY POSITIONAL EMBEDDINGS)

An important difference:

- The original Transformers adds positional embedding to the token embedding, not later
- RoPE adds positional information in each attention head

RoPE rotates embeddings vectors by an angle which depends on the position (encoding this way both absolute and relative positions).

THE FORMULA

Fix $c = \text{context_length}$ and $d = \text{input_dim}$. Let $x : (c, d)$ the embedding vector.

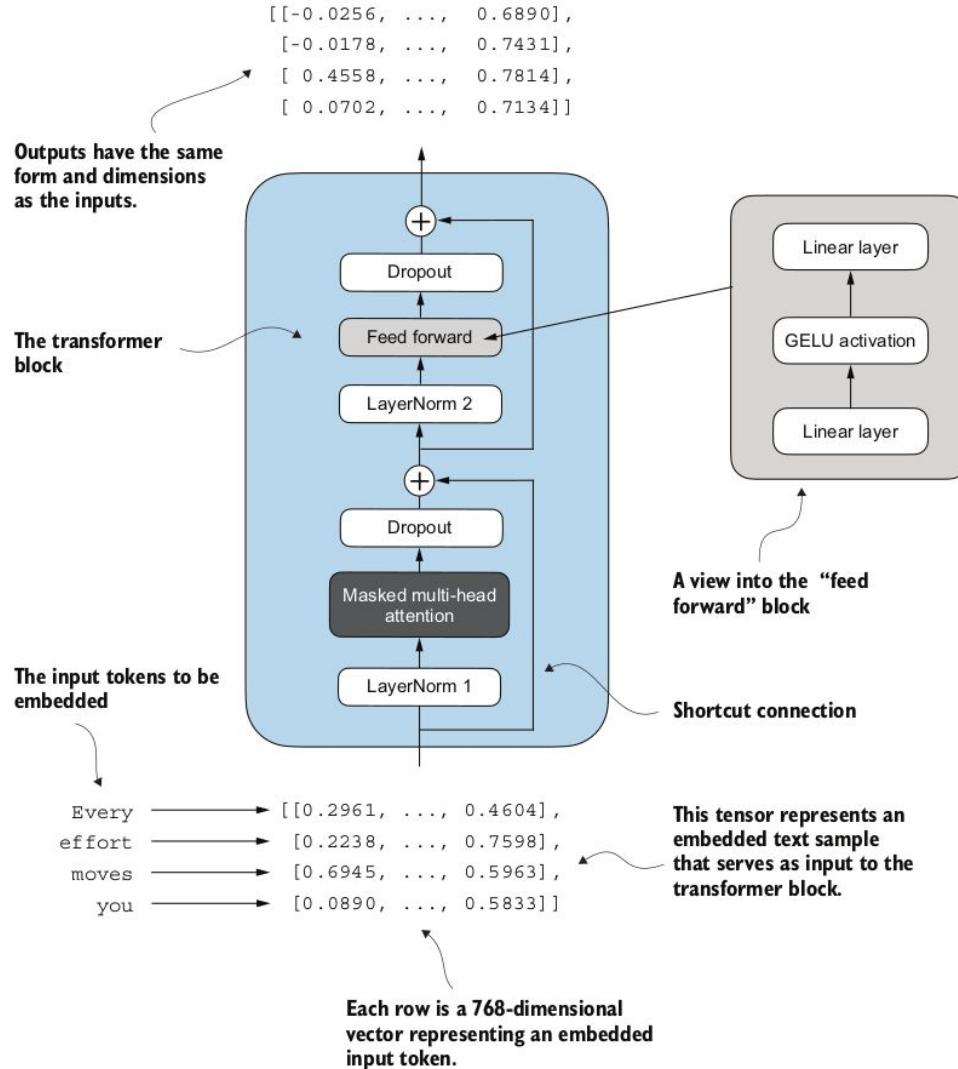
We group dimensions by pairs, and for pair $(i, i+d/2)$ we apply a rotation of angle $\text{pos} * \theta_i$ where $\theta_i = 10_000^{-2(i-1)/d}$:

$$x(\text{pos}, i) = x(\text{pos}, i) \cos(\text{pos} * \theta_i) - x(\text{pos}, i+d/2) \sin(\text{pos} * \theta_i)$$

$$x(\text{pos}, i+d/2) = x(\text{pos}, i+d/2) \cos(\text{pos} * \theta_i) + x(\text{pos}, i) \sin(\text{pos} * \theta_i)$$

TRANSFORMER ARCHITECTURE



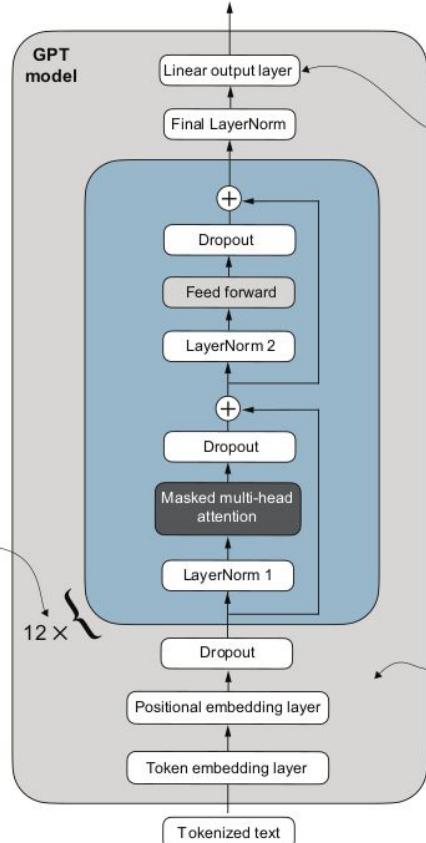


A $4 \times 50,257$ -dimensional tensor

```
[ [-0.0055, ..., -0.4747],  
 [ 0.2663, ..., -0.4224],  
 [ 1.1146, ..., 0.0276],  
 [-0.8239, ..., -0.3993] ]
```

The goal is for these embeddings to be converted back into text such that the last row represents the word the model is supposed to generate (here, the word "forward").

The transformer block is repeated 12 times.



The last linear layer embeds each token vector into a $50,257$ -dimensional embedding, where $50,257$ is the size of the vocabulary.

The GPT code implementation includes a token embedding and positional embedding layer (see chapter 2).

PRETRAINING



BOILERPLATE TRAINING CODE

```
@torch.no_grad()
def estimate_loss(model):
    out = {}
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    return out
```

```
def train(model):
    # create a PyTorch optimizer
    optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

    for iter in range(n_iterations):
        # every once in a while evaluate the loss on train and validation sets
        if iter % eval_interval == 0 or iter == n_iterations - 1:
            losses = estimate_loss(model, eval_iters)
            print(f"step {iter}: train loss {losses['train']:.4f}, validation loss {losses['val']:.4f}")

        X,Y = get_batch("train")
        _, loss = model(X, Y)
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
```

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 .

CROSS ENTROPY LOSS

```
vocab_size = 5

logits = torch.randn(vocab_size)
print("The logits: \n", logits)
probs = torch.softmax(logits, 0)
print("After softmax: \n", probs)
logprobs = -probs.log()
print("The -log probabilities: \n", logprobs)

y = torch.randint(vocab_size, (), dtype=torch.int64)
print("\nLet us consider a target y: ", y.item())

loss = F.cross_entropy(logits, y)
print("The cross entropy loss between logits and y is: ", loss.item())
```

The logits:
tensor([0.0465, 0.2514, -0.6639, -0.5434, -0.0025])

After softmax:
tensor([0.2367, 0.2905, 0.1163, 0.1312, 0.2253])
The -log probabilities:
tensor([1.4411, 1.2362, 2.1516, 2.0310, 1.4901])

Let us consider a target y: 0
The cross entropy loss between logits and y is: 1.4411031007766724

WHY IS CROSS ENTROPY LOSS INTERESTING?

- **Maximum likelihood estimation:** Minimizing cross-entropy is equivalent to maximizing the likelihood of the observed data.
- **Encourages accurate probabilities:** It encourages the model to produce probabilities that closely match the true distribution, not just predict the correct class.
- **Smooth and differentiable:** Cross-entropy loss is a smooth and differentiable function, which is crucial for gradient-based optimization algorithms like gradient descent.
- **Avoids saturation:** Unlike some other loss functions (e.g., mean squared error with sigmoid), cross-entropy with softmax reduces the problem of saturating gradients.

TOKENIZATION

- Basics of encoding
 - Pre-tokenization
 - Byte-Pair Encoding (BPE)
 - WordPiece
-

CREDITS

Images and contents from Chapter 6 of Hugging Face's course on NLP:

<https://huggingface.co/learn/nlp-course/chapter6>

BASICS

A **bit** = 0 or 1

A **byte** = typically an octet, meaning 8 bits

Character encodings:

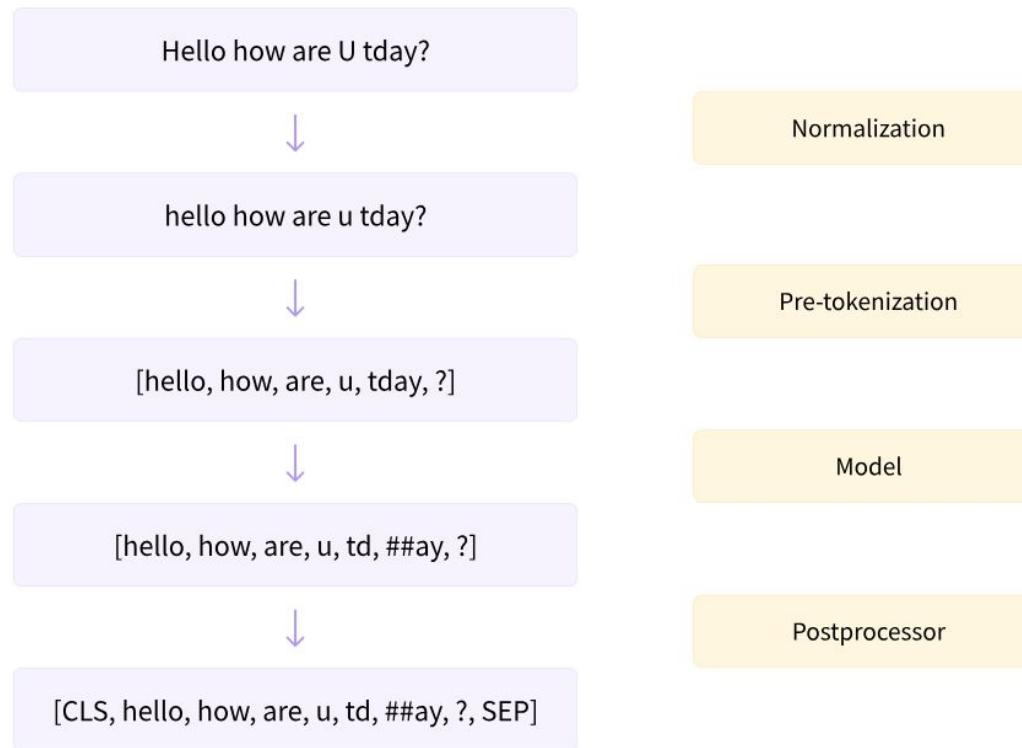
- ASCII (code unit: 7 bits)
- Unicode: UTF-8, UTF-16, UTF-32 (code unit: 8,16,32 bits)

98% of WWW is UTF-8. Technically UTF is variable-length (so infinite...)

ATTENTION

We are only considering “subword tokenization algorithms”
but there are other tokenization algorithms...

THE FULL TOKENIZATION PIPELINE



NORMALIZATION

The normalization step involves some general cleanup, such as removing needless whitespace, lowercasing, and/or removing accents.

```
from transformers import AutoTokenizer  
  
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")  
print(type(tokenizer.backend_tokenizer))  
  
<class 'tokenizers.Tokenizer'>  
  
print(tokenizer.backend_tokenizer.normalize_str("Héllò hôw are ü?"))  
hello how are u?
```

PRE-TOKENIZATION

Breaks a text into words (keeping the offsets):

```
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are you?")  
[('Hello', (0, 5)),  
 (',', (5, 6)),  
 ('how', (7, 10)),  
 ('are', (11, 14)),  
 ('you', (16, 19)),  
 ('?', (19, 20))]
```

PRE-TOKENIZATION

Again there are many variants...

SentencePiece is a simple pre-tokenization algorithm:

- Treats everything as Unicode characters
- Replaces spaces with “_”

TOKENIZATION ALGORITHMS

Two components:

- The *training* algorithm: preprocessing on a training set, to determine what will be the tokens
- The *tokenization* algorithm: at run time, transforming text inputs into sequences of tokens

BYTE-PAIR ENCODING

Developed by OpenAI for GPT-2

Pre-tokenization adds “Ġ” before each word except the first:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("gpt2")
tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str("Hello, how are you?")

[('Hello', (0, 5)),
 (',', (5, 6)),
 ('Ġhow', (6, 10)),
 ('Ġare', (10, 14)),
 ('Ġ', (14, 15)),
 ('Ġyou', (15, 19)),
 ('?', (19, 20))]
```

BPE IN ONE SLIDE

The goal is to learn merge rules, of the form:

(“Amer”, “ica”) → “America”

Training: starting from characters, we create rules by merging the most frequent pairs, until we reach the budget number of tokens

Processing: to process an input text we apply rules greedily

EXAMPLE CORPUS

```
corpus = [  
    "This is the Hugging Face Course.",  
    "This chapter is about tokenization.",  
    "This section shows several tokenizer algorithms.",  
    "Hopefully, you will be able to understand how they are trained and generate tokens.",  
]
```

BPE TRAINING ALGORITHM, STEP 0: COMPUTE FREQUENCIES

```
from collections import defaultdict

word_freqs = defaultdict(int)

for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

print(word_freqs)

defaultdict(<class 'int'>, {'This': 3, 'is': 2, 'the': 1, 'Hugging': 1, 'Face': 1, 'Course': 1, '.': 4, 'chapter': 1, 'about': 1, 'tokenization': 1, 'section': 1, 'shows': 1, 'several': 1, 'tokenizer': 1, 'algorithm': 1, 'Hopefully': 1, ',': 1, 'you': 1, 'will': 1, 'be': 1, 'able': 1, 'to': 1, 'understand': 1, 'how': 1, 'they': 1, 'are': 1, 'trained': 1, 'and': 1, 'generate': 1, 'tokens': 1})
```

BPE TRAINING ALGORITHM, STEP 1: COLLECT CHARACTERS

```
alphabet = []

for word in word_freqs.keys():
    for letter in word:
        if letter not in alphabet:
            alphabet.append(letter)
alphabet.sort()

print(alphabet)
```

```
[',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'r',
's', 't', 'u', 'v', 'w', 'y', 'z', 'G']
```

```
vocab = ["<|endoftext|>"] + alphabet.copy()
```

“<|endoftext|>” is a special token

BPE TRAINING ALGORITHM, STEP 2: COMPUTE PAIR FREQUENCIES

```
splits = {word: [c for c in word] for word in word_freqs.keys()}
```

```
def compute_pair_freqs(splits):
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            pair_freqs[pair] += freq
    return pair_freqs
```

```
pair_freqs = compute_pair_freqs(splits)

for i, key in enumerate(pair_freqs.keys()):
    print(f'{key}: {pair_freqs[key]}')
    if i >= 5:
        break
```

```
('T', 'h'): 3
('h', 'i'): 3
('i', 's'): 5
('G', 'i'): 2
('G', 't'): 7
('t', 'h'): 3
```

```
best_pair = ""
max_freq = None

for pair, freq in pair_freqs.items():
    if max_freq is None or max_freq < freq:
        best_pair = pair
        max_freq = freq

print(best_pair, max_freq)

('G', 't') 7
```

BPE TRAINING ALGORITHM, STEP 3: ADD A MERGE RULE

```
merges = {("t", "t"): "Gt"}  
vocab.append("Gt")
```

```
def merge_pair(a, b, splits):  
    for word in word_freqs:  
        split = splits[word]  
        if len(split) == 1:  
            continue  
  
        i = 0  
        while i < len(split) - 1:  
            if split[i] == a and split[i + 1] == b:  
                split = split[:i] + [a + b] + split[i + 2 :]  
            else:  
                i += 1  
        splits[word] = split  
    return splits
```

```
splits = merge_pair("t", "t", splits)  
print(splits["Gtrained"])
```

```
['Gt', 'r', 'a', 'i', 'n', 'e', 'd']
```

BPE TRAINING ALGORITHM: THE LOOP

```
vocab_size = 50

while len(vocab) < vocab_size:
    pair_freqs = compute_pair_freqs(splits)
    best_pair = ""
    max_freq = None
    for pair, freq in pair_freqs.items():
        if max_freq is None or max_freq < freq:
            best_pair = pair
            max_freq = freq
    splits = merge_pair(*best_pair, splits)
    merges[best_pair] = best_pair[0] + best_pair[1]
    vocab.append(best_pair[0] + best_pair[1])
```

```
print(merges)
```

```
{('t', 't'): 'tt', ('i', 's'): 'is', ('e', 'r'): 'er', ('a', 'a'): 'aa', ('o', 'o'): 'oo', ('e', 'n'): 'en',
('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('to', 'k'): 'tok', ('tok', 'en'): 'token',
('n', 'd'): 'nd', ('is', 'is'): 'isis', ('t', 'h'): 'th', ('th', 'e'): 'the', ('i', 'n'): 'in', ('a', 'b'): 'ab',
('tok', 'en'): 'tokeni'}
```

```
print(vocab)
```

```
[ '<|endoftext|>', ',', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n',
'o', 'p', 'r', 's', 't', 'u', 'v', 'w', 'y', 'z', 'G', 'Gt', 'is', 'er', 'Ga', 'Gto', 'en', 'Th', 'This', 'ou', 'se',
'Gtok', 'Gtoken', 'nd', 'Gis', 'Gth', 'Gthe', 'in', 'Gab', 'Gtokeni' ]
```

BPE TOKENIZATION ALGORITHM

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    splits = [[l for l in word] for word in pre_tokenized_text]
    for pair, merge in merges.items():
        for idx, split in enumerate(splits):
            i = 0
            while i < len(split) - 1:
                if split[i] == pair[0] and split[i + 1] == pair[1]:
                    split = split[:i] + [merge] + split[i + 2 :]
                else:
                    i += 1
            splits[idx] = split

    return sum(splits, [])
```

```
tokenize("This is not a token.")
```

```
['This', 'Ġis', 'Ġ', 'n', 'o', 't', 'Ġa', 'Ġtoken', '.']
```

BPE TOKENIZATION ALGORITHM CAN FAIL?

What happens if there's an unknown character? This code would fail...

In actual (byte-level) implementations, it cannot happen.

IN PRACTICE

Tiktoken implements BPE:

<https://github.com/openai/tiktoken>

WORDPIECE

Developed by Google for BERT (but never open sourced!)

The pre-tokenizer feels a lot more civilized:

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
tokenizer.backend_tokenizer.pre_tokenize_str("Hello, how are you?")

[('Hello', (0, 5)),
 (',', (5, 6)),
 ('how', (7, 10)),
 ('are', (11, 14)),
 ('you', (15, 18)),
 ('?', (18, 19))]
```

WORDPIECE IN ONE SLIDE

The goal is to learn merge rules, of the form:
("Amer", "ica") -> "America"

Training: starting from characters, we create tokens by merging pairs with highest score, until we reach the budget number of tokens

Processing: to process an input text we look for the longest token and continue recursively (not using rules!)

WORDPIECE TRAINING ALGORITHM, STEP 0: COMPUTE CHARACTERS

```
from collections import defaultdict

word_freqs = defaultdict(int)
for text in corpus:
    words_with_offsets = tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str(text)
    new_words = [word for word, offset in words_with_offsets]
    for word in new_words:
        word_freqs[word] += 1

word_freqs

defaultdict(int,
            {'This': 3,
             'is': 2,
             'the': 1,
             'Hugging': 1,
             'Face': 1,
             'Course': 1,
             '.': 4,
             'chapter': 1,
             'about': 1,
             'tokenization': 1,
             'section': 1,
             'shows': 1,
             'several': 1,
             'tokenizer': 1,
             'algorithms': 1,
             'Hopefully': 1,
```

WORDPIECE TRAINING ALGORITHM, STEP 1: COMPUTE FREQUENCIES

```
alphabet = []
for word in word_freqs.keys():
    if word[0] not in alphabet:
        alphabet.append(word[0])
    for letter in word[1:]:
        if f"##'{letter}'" not in alphabet:
            alphabet.append(f"##'{letter}'")

alphabet.sort()
alphabet

print(alphabet)

['##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p', '##r',
'##s', '##t', '##u', '##v', '##w', '##y', '##z', '.', ',', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's',
't', 'u', 'w', 'y']

vocab = "[PAD]", "[UNK]", "[CLS]", "[SEP]", "[MASK]" + alphabet.copy()

splits = {
    word: [c if i == 0 else f"##'{c}" for i, c in enumerate(word)]
    for word in word_freqs.keys()
}
splits

{'This': ['T', '##h', '##i', '##s'],
 'is': ['i', '##s'],
 'the': ['t', '##h', '##e'],
 'Hugging': ['H', '##u', '##g', '##g', '##i', '##n', '##g'],
 'Face': ['F', '##a', '##c', '##e'],
 'Course': ['C', '##o', '##u', '##r', '##s', '##e'],
```

WORDPIECE TRAINING ALGORITHM, STEP 2: COMPUTE SCORES

WordPiece computes a score for each pair, using the following formula:

$$\text{freq_of_pair} / (\text{freq_of_first_element} \times \text{freq_of_second_element})$$

The algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary:

- It won't necessarily merge ("un", "#able") even if that pair occurs very frequently in the vocabulary, because the two pairs "un" and "#able" will likely each appear in a lot of other words and have a high frequency.
- In contrast, a pair like ("hu", "#gging") will probably be merged faster (assuming the word "hugging" appears often in the vocabulary) since "hu" and "#gging" are likely to be less frequent individually.

WORDPIECE TRAINING ALGORITHM, STEP 2: COMPUTE SCORES

```
def compute_pair_scores(splits):
    letter_freqs = defaultdict(int)
    pair_freqs = defaultdict(int)
    for word, freq in word_freqs.items():
        split = splits[word]
        if len(split) == 1:
            letter_freqs[split[0]] += freq
            continue
        for i in range(len(split) - 1):
            pair = (split[i], split[i + 1])
            letter_freqs[split[i]] += freq
            pair_freqs[pair] += freq
        letter_freqs[split[-1]] += freq

    scores = {
        pair: freq / (letter_freqs[pair[0]] * letter_freqs[pair[1]])
        for pair, freq in pair_freqs.items()
    }
    return scores
```

```
best_pair = ""
max_score = None
for pair, score in pair_scores.items():
    if max_score is None or max_score < score:
        best_pair = pair
        max_score = score

print(best_pair, max_score)
```

```
('a', '#b') 0.2
```

```
vocab.append("ab")
```

```
def merge_pair(a, b, splits):
    for word in word_freqs:
        split = splits[word]
        if len(split) == 1:
            continue
        i = 0
        while i < len(split) - 1:
            if split[i] == a and split[i + 1] == b:
                merge = a + b[2:] if b.startswith("##") else a + b
                split = split[:i] + [merge] + split[i + 2 :]
            else:
                i += 1
        splits[word] = split
    return splits
```

```
splits = merge_pair("a", "#b", splits)
splits["about"]
```

```
['ab', '#o', '#u', '#t']
```

WORDPIECE TRAINING ALGORITHM: THE LOOP

```
vocab_size = 70
while len(vocab) < vocab_size:
    scores = compute_pair_scores(splits)
    best_pair, max_score = "", None
    for pair, score in scores.items():
        if max_score is None or max_score < score:
            best_pair = pair
            max_score = score
    splits = merge_pair(*best_pair, splits)
    new_token =
        best_pair[0] + best_pair[1][2:]
        if best_pair[1].startswith("##")
        else best_pair[0] + best_pair[1]
    )
    vocab.append(new_token)
```

```
print(vocab)
```

```
[ '[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]', '##a', '##b', '##c', '##d', '##e', '##f', '##g', '##h', '##i', '##k', '##l', '##m', '##n', '##o', '##p', '##r', '##s', '##t', '##u', '##v', '##w', '##y', '##z', '.', '.', '.', 'C', 'F', 'H', 'T', 'a', 'b', 'c', 'g', 'h', 'i', 's', 't', 'u', 'w', 'y', 'ab', '##fu', 'Fa', 'Fac', '##ct', '##ful', '##full', '##fully', 'Th', 'ch', '##hm', 'cha', 'chap', 'chapt', '##thm', 'Hu', 'Hug', 'Hugg', 'sh', 'th', 'is', '##ths', '##za', '##zat', '##ut' ]
```

WORDPIECE TOKENIZATION ALGORITHM

```
def encode_word(word):
    tokens = []
    while len(word) > 0:
        i = len(word)
        while i > 0 and word[:i] not in vocab:
            i -= 1
        if i == 0:
            return ["[UNK]"]
        tokens.append(word[:i])
        word = word[i:]
        if len(word) > 0:
            word = f"##{word}"
    return tokens
```

```
print(encode_word("Hugging"))
print(encode_word("HOgging"))
```

```
['Hugg', '##i', '##n', '##g']
['[UNK]']
```

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_tokenize_str(text)
    pre_tokenized_text = [word for word, offset in pre_tokenize_result]
    encoded_words = [encode_word(word) for word in pre_tokenized_text]
    return sum(encoded_words, [])
```

SUMMARY FOR THE TWO ALGORITHMS

Model	BPE	WordPiece
Training	Starts from a small vocabulary and learns rules to merge tokens	Starts from a small vocabulary and learns rules to merge tokens
Training step	Merges the tokens corresponding to the most common pair	Merges the tokens corresponding to the pair with the best score based on the frequency of the pair, privileging pairs where each individual token is less frequent
Learns	Merge rules and a vocabulary	Just a vocabulary
Encoding	Splits a word into characters and applies the merges learned during training	Finds the longest subword starting from the beginning that is in the vocabulary, then does the same for the rest of the word

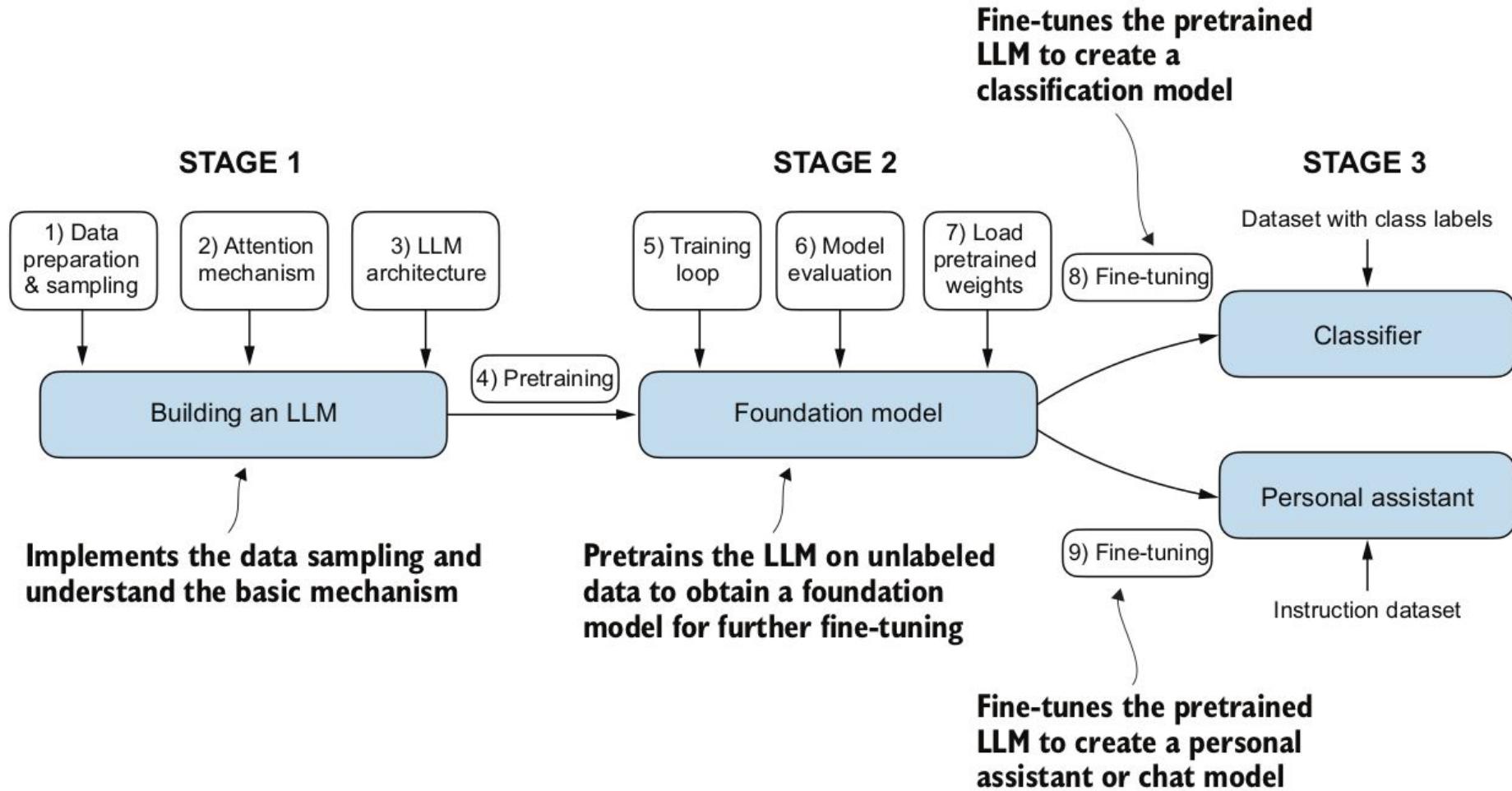
FINE-TUNING

- General overview
 - LoRA
-

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, name entity recognition...



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)

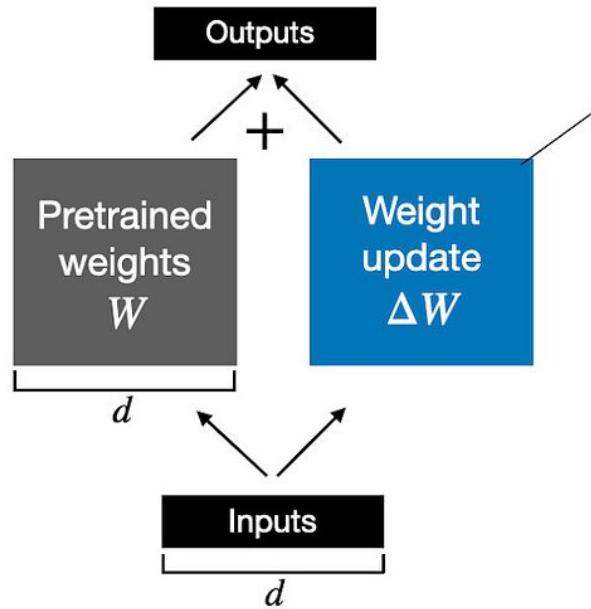
Two key ideas:

- (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

Weight update in regular finetuning

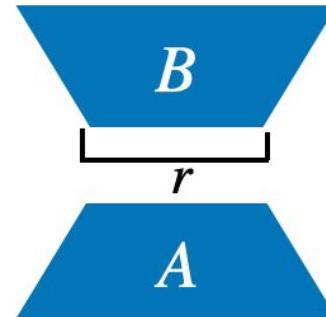


RANK APPROXIMATIONS

A matrix W of dimension $d \times d$ contains $d \times d$ parameters. It can be ***rank- r approximated*** by two matrices $A \times B$ with:

- A of dimension $d \times r$
- B of dimension $r \times d$

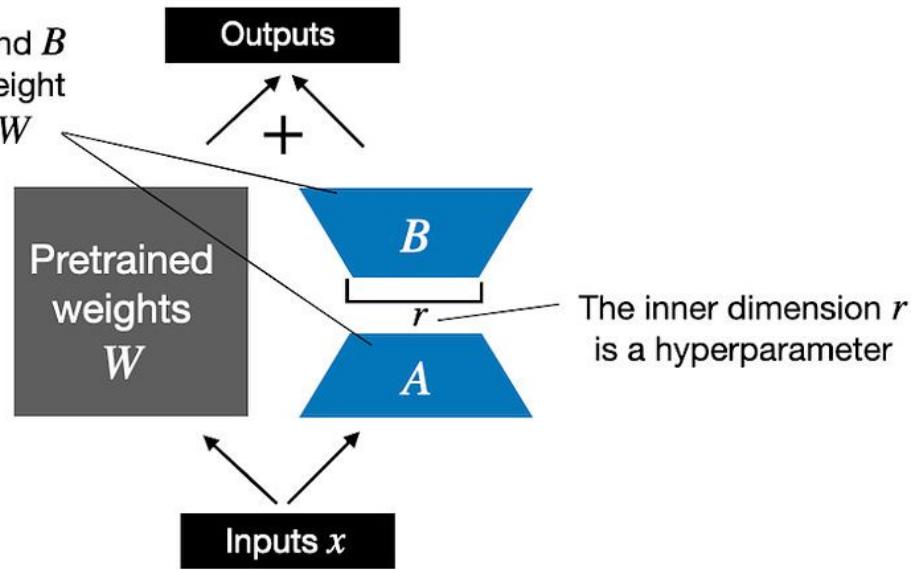
Instead of $d \times d$ parameters we now have $2 \times d \times r$ parameters.



WEIGHT UPDATE

Weight update in LoRA

LoRA matrices A and B
approximate the weight
update matrix ΔW



LORA LAYER

```
import math

class LoRALayer(torch.nn.Module):
    def __init__(self, in_dim, out_dim, rank, alpha):
        super().__init__()
        std_dev = 1 / torch.sqrt(torch.tensor(rank).float())
        self.A = nn.Parameter(torch.randn(in_dim, rank) * std_dev)
        self.B = nn.Parameter(torch.zeros(rank, out_dim))
        self.alpha = alpha

    def forward(self, x):
        x = self.alpha * (x @ self.A @ self.B)
        return x
```

ADDING THE LORA LAYER

```
class LinearWithLoRA(torch.nn.Module):
    def __init__(self, linear, rank, alpha):
        super().__init__()
        self.linear = linear
        self.lora = LoRALayer(
            linear.in_features, linear.out_features, rank, alpha
        )

    def forward(self, x):
        return self.linear(x) + self.lora(x)
```

```
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

```
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
```

```
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
```

```
print(model)

GPTModel(
  (tok_emb): Embedding(50257, 768)
  (pos_emb): Embedding(1024, 768)
  (drop_emb): Dropout(p=0.0, inplace=False)
  (trf_blocks): Sequential(
    (0): TransformerBlock(
      (att): MultiHeadAttention(
        (W_query): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_key): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
        (W_value): LinearWithLoRA(
          (linear): Linear(in_features=768, out_features=768, bias=True)
          (lora): LoRALayer()
        )
      )
    )
  )
)
```

SOME HUGGING FACE PROPAGANDA

- Models
 - Pipeline and tasks
 - 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/](https://huggingface.co/)



It is 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...

Academics:

- grid5000 <https://www.grid5000.fr/>
- 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

This is only for holding the model in memory! You should aim at 4x this for inference and fine-tuning.

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

```
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)

```
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.

WTF?

```
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/distilbert-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="TO BE FILLED HERE",
)
client.text_classification("Today is a great day")

[TextClassificationOutputElement(label='positive', score=0.9836677312850952),
 TextClassificationOutputElement(label='neutral', score=0.01135887298732996),
 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.

```
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),
}

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 simple: better introduce some overlapping!)

Good news: Tokenizer does that for you!

```
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]]}
```

DATASETS

```
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 front 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 food, I asked where mine was. The manager started yelling at the cashiers for \\\"serving off their orders\\\" when they didn\'t have their food. But neither cashier was anywhere near those controls, and the manager was the one serving food to customers and clearing the boards.\\nThe manager was rude when giving me my order. She didn\'t make sure 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 store. 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()
```

PARAMETER-EFFICIENT FINE-TUNING (PEFT)

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)
```

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

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

```
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

INFERENCE

PAGED ATTENTION

Key idea: when generating many tokens with the same prompt, many values can be cached! The keys and queries are augmented with new values rather than recomputed.

Doing it efficiently in a GPU-friendly way is non-trivial...

VLLM

Faster inference: <https://github.com/vllm-project/vllm>



RETRIEVAL-AUGMENTED GENERATION (RAG)



THE IDEA

Allow the LLM to access an external source of knowledge, later referred to as corpus.

Main application: navigate long documents (manuals, regulations,...)

Does it reduce hallucination? Yes. Is it *the* solution? **No!**

GENERIC FRAMEWORK

We have a vectorstore, which is a database (key, value) where:

- **Keys** are “embedding vectors”
- **Values** are “chunks”

An *embedding vector* is a vector of float of fixed dimension.

A *chunk* is a piece of text extracted from the corpus.

PREPROCESSING

This is the process of populating the dataset. It involves:

- **Text cleaning and splitting:** from a document to a set of chunks
- **Embedding:** computing embeddings for each chunk

PREPROCESSING: TEXT SPLITTING

Naive: fix chunk length and split

Better:

- split on new lines
- include chunk overlap

PREPROCESSING: EMBEDDING

Pre-LLM:

- Word2vec: uses a simple RNN
- GloVe: Global Vectors for Word Representation

Post-LLM:

- BERT, for instance ModernBert (see MTEB for benchmarks)

PREPROCESSING: CONTEXTUAL EMBEDDING

A simple idea by Anthropic (Claude): instead of embedding the chunk **itself**, we:

- Ask an LLM to produce a description of the chunk (using the chunk **and** the full document)
- Embed the description + the chunk

PROCESSING

At inference time:

- **Query:** from the prompt we construct queries to the database
- **Retrieve:** we retrieve the most relevant chunks
- **Answer:** we use the added contents to formulate an answer to the prompt

PROCESSING: QUERY

Naive: turn the prompt into a query

Better:

- Ask an LLM to formulate a query from the prompt
- HyDE: ask an LLM to generate a hypothetical document, embed this document, and retrieve similar documents

PROCESSING: RETRIEVE

Naive:

- Choose a notion of similarity between embedding vectors
- Retrieve the K-nearest neighbours

Better:

- For scaling: use approximation algorithms
- For diversity: use an SVM

PROCESSING: NOTIONS OF SIMILARITY

There are many notions of similarity:

- Cosine similarity
- Dot product
- Euclidean distance

Remark: when vectors are normalized, cosine similarity coincides with dot product

PROCESSING: ANSWER

Naive: add the most relevant chunks to the prompt

Better:

- Ask an LLM, called a reranker, to filter and rearrange the most relevant chunks
- Perform a BM25 on the side, which operates on keywords, and merge the resulting most relevant chunks

FROM UNSTRUCTURED TO STRUCTURED RAG

We only discussed the case where the corpus is unstructured. If it is structured, there are more specific techniques...

IN PRACTICE

Two frameworks for building RAGs:

- [LangChain](#)
- [LlamaIndex](#)

They do everything for you, sometimes not leaving enough control...

MULTI-HEAD LATENT
ATTENTION



MIXTURE OF EXPERTS



GROUP RELATIVE POLICY OPTIMIZATION (GRPO)
