Large Language Models

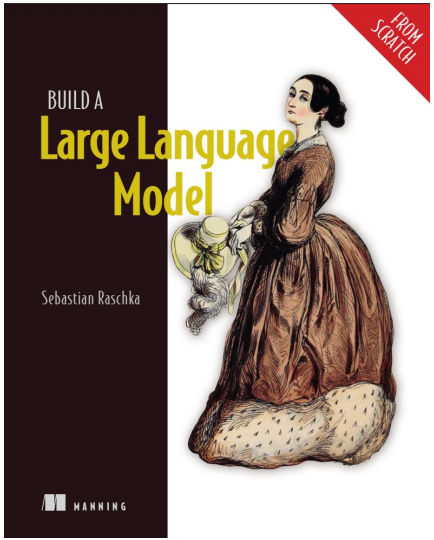
Nathanaël Fijalkow
CNRS, LaBRI, Bordeaux

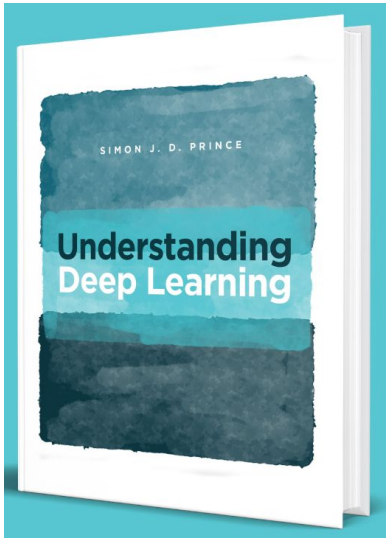
Goal of this course

- 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

Other sources are mentioned when used

Language models

What is a language model (LM)?

Input: a sentence (as a sequence of tokens)

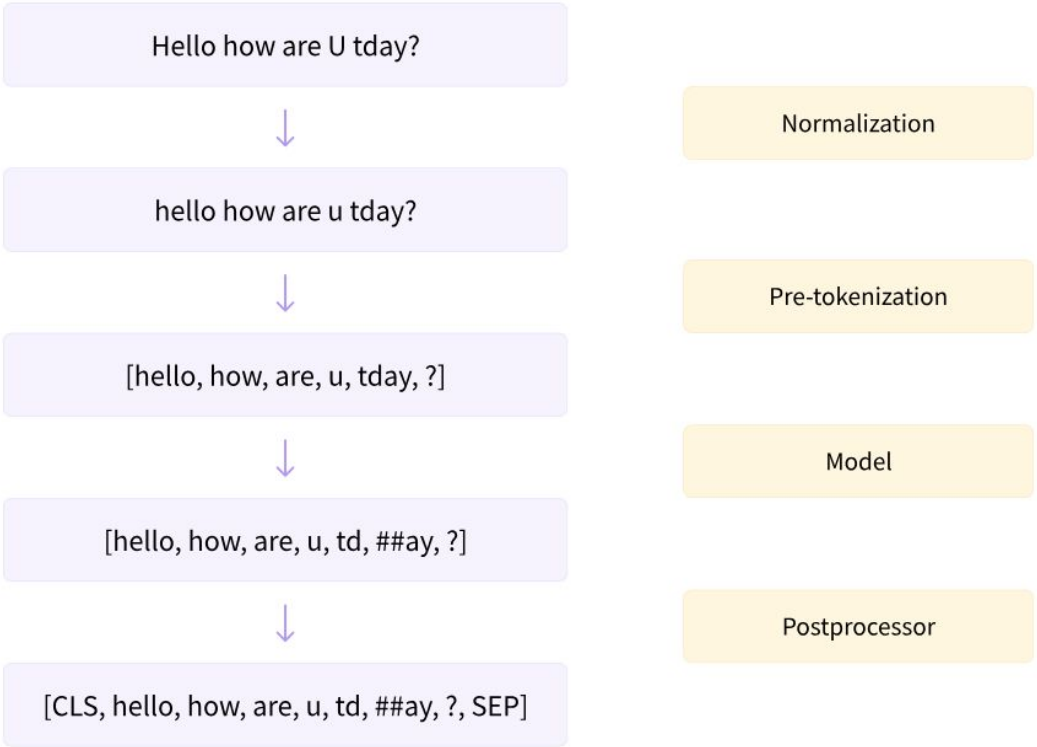
Output: predict the next token

Basic examples:

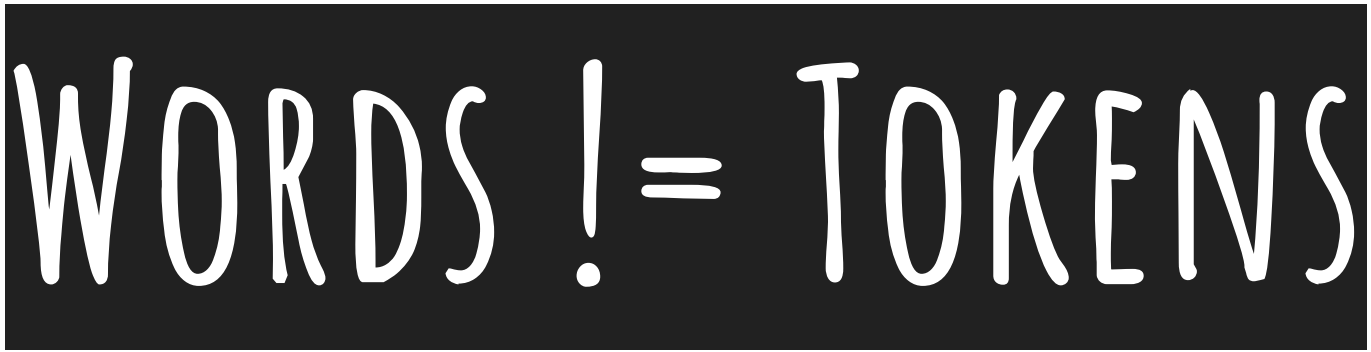
- *Markov chain* is a LM, it gives a probabilistic distribution over the next token given the last token
- Naturally extended to *n-grams*: use the (n-1) last tokens to predict the next one

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**



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

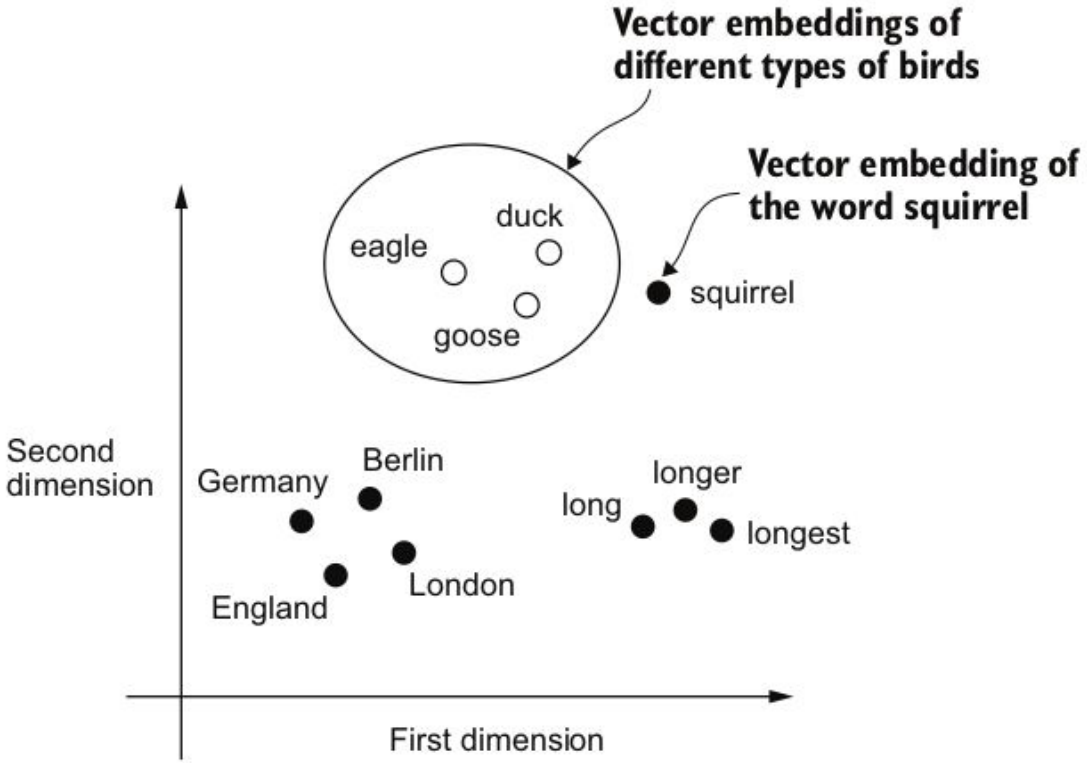
The 2003 (silent) breakthrough

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

A Neural Probabilistic Language Model

Yoshua Bengio 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



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.

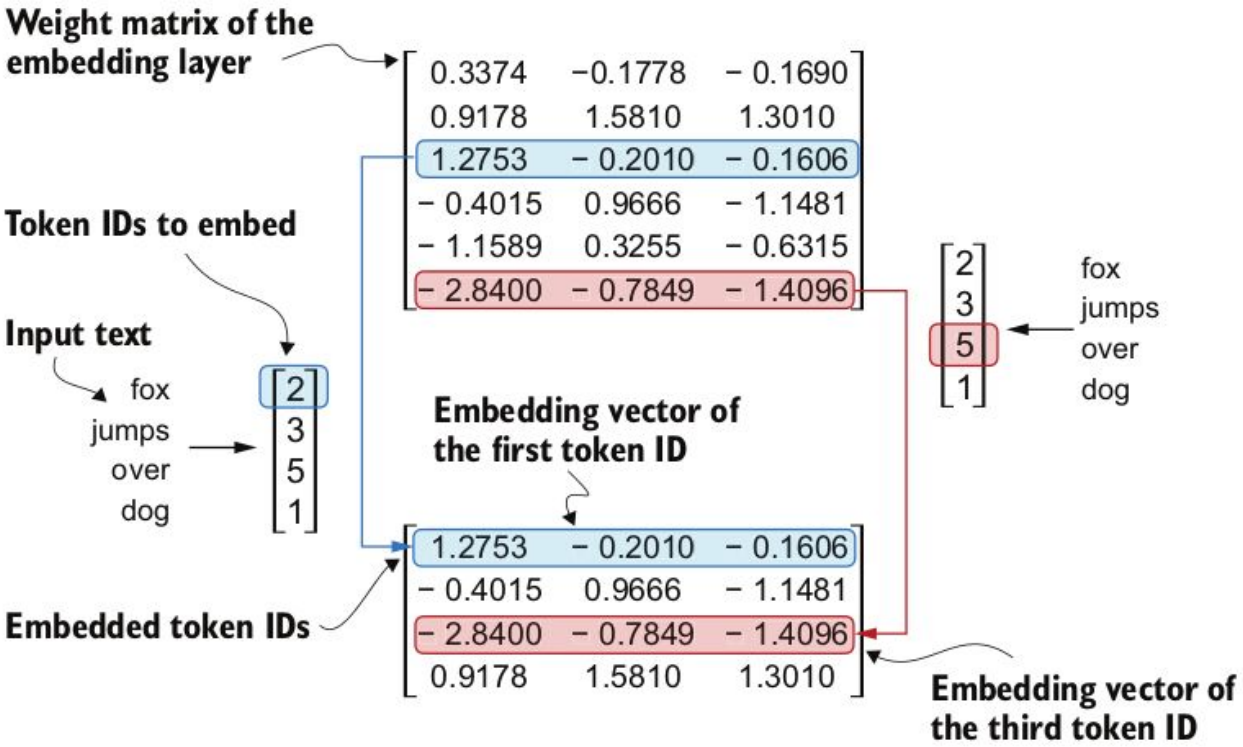
`nn.Linear`

- **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:

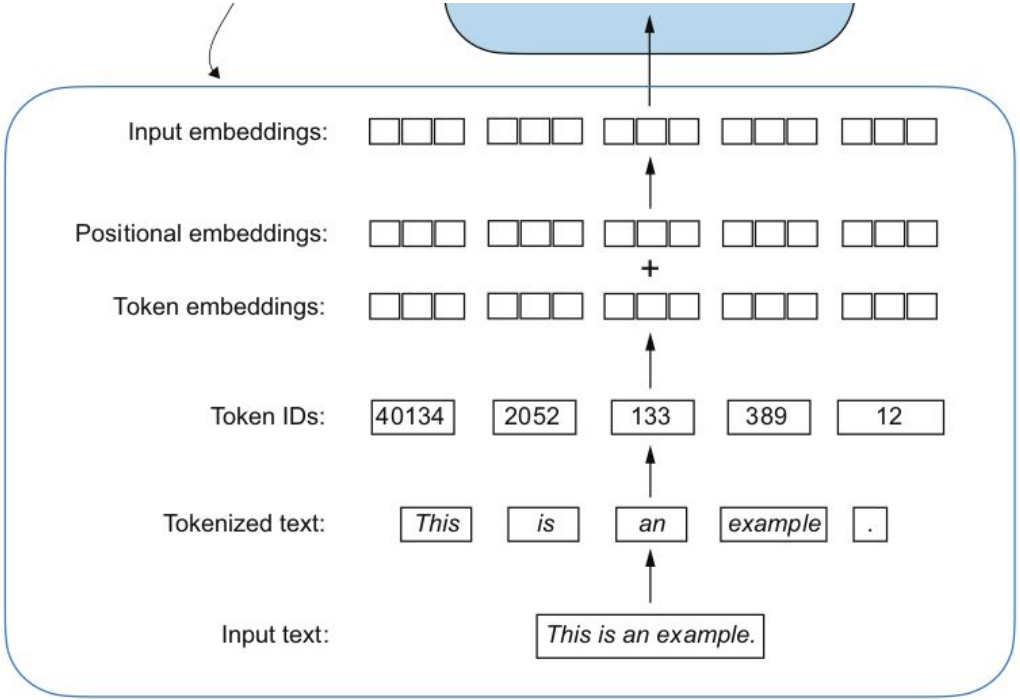
- Use `nn.Embedding` for representing categorical data as dense vectors.
- Use `nn.Linear` for performing linear transformations in neural networks.

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.

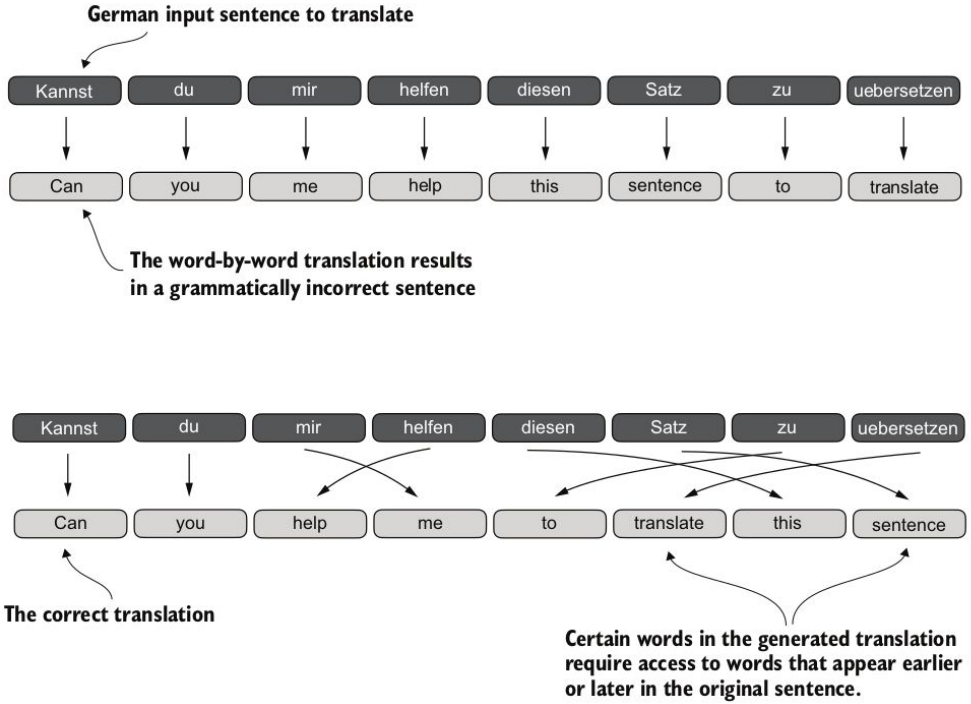
The largest GPT-3 model (175B parameters) uses an embedding size of 12,288 dimensions.

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

- Long contexts require huge amount of compute
- Struggle with long-range 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 (made parallelizable) reached similar performances as Transformers…

A self-attention head

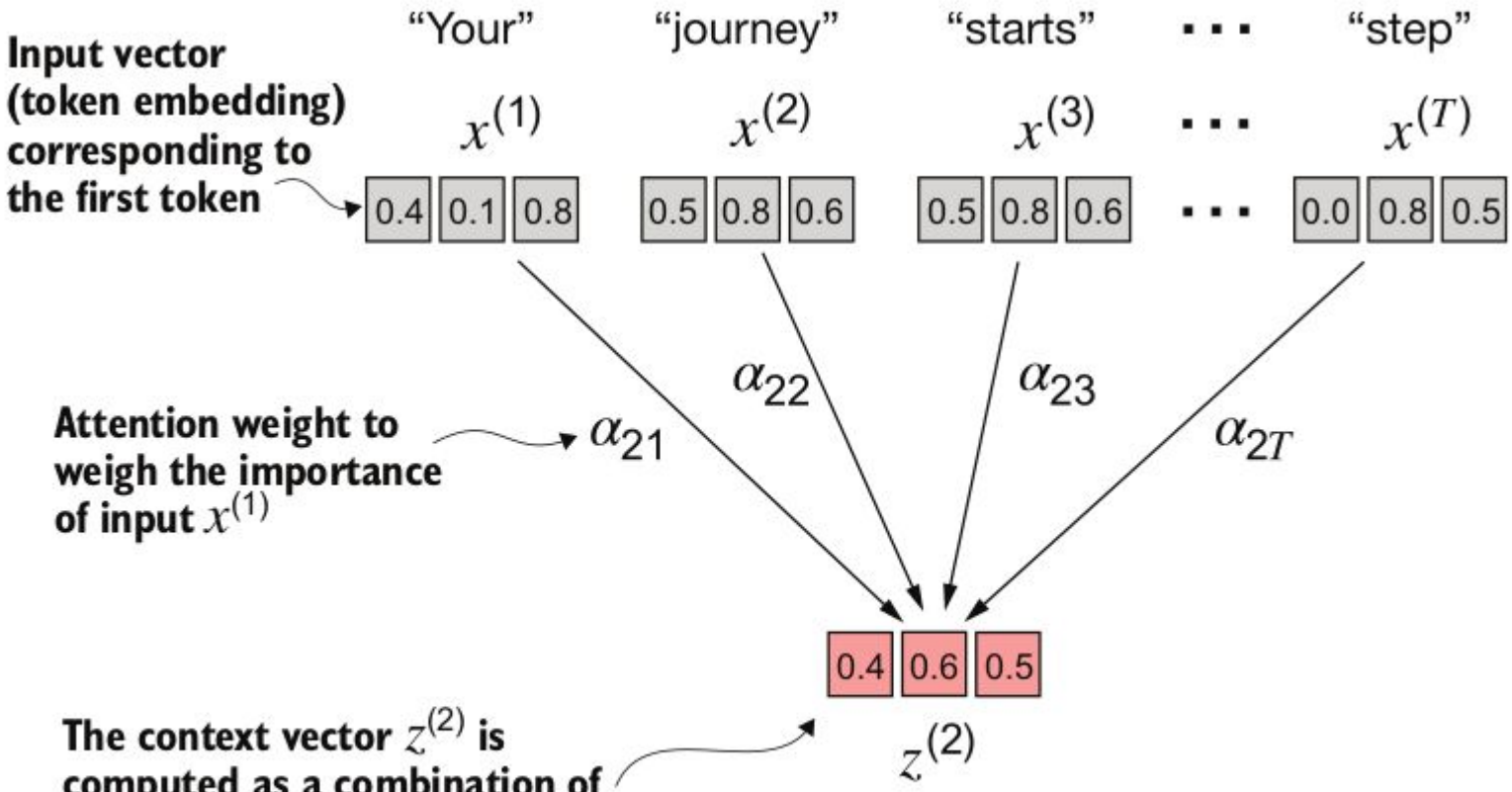
Input: an embedding vector x(i) for each token i

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

Intuition: z(i) gathers *contextual* information

Computing context vectors

Computing context vectors is very easy assuming we have computed **attention weights**: alpha(i,j) describes the importance of token j for token i.


"Your"
"journey"
"starts"
"step"
Input vector (token embedding) corresponding to the first token
$x^{(1)}$
$x^{(2)}$
$x^{(3)}$
$x^{(T)}$
0.4 0.1 0.8
0.5 0.8 0.6
0.5 0.8 0.6
0.0 0.8 0.5
α_{21}
α_{22}
α_{23}
α_{2T}
Attention weight to weigh the importance of input $x^{(1)}$
0.4 0.6 0.5
$z^{(2)}$
The context vector $z^{(2)}$ is computed as a combination of all input vectors weighted with respect to input element $x^{(2)}$

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***.

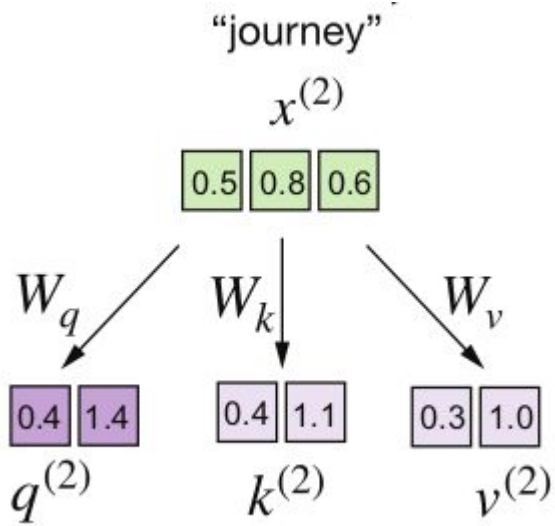
Here, keys, queries, and values are vectors. To match a query with a key we simply do a dot-product.

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:

- Wq for query,
- Wk for key,
- Wv 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, illustrated below:

```
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 $u \cdot v$?

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

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

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

```python
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 of self-attention heads

C = context_length

I = input_dim

H = head_dim

O = output_dim

- key(x): (C x I) x (I x H) -> C x H
- query(x): (C x I) x (I x H) -> C x H
- value(x): (C x I) x (I x O) -> C x O
- attention_scores: (C x H) x (H x C) -> C x C
- context_vectors: (C x C) x (C x O) -> C x O

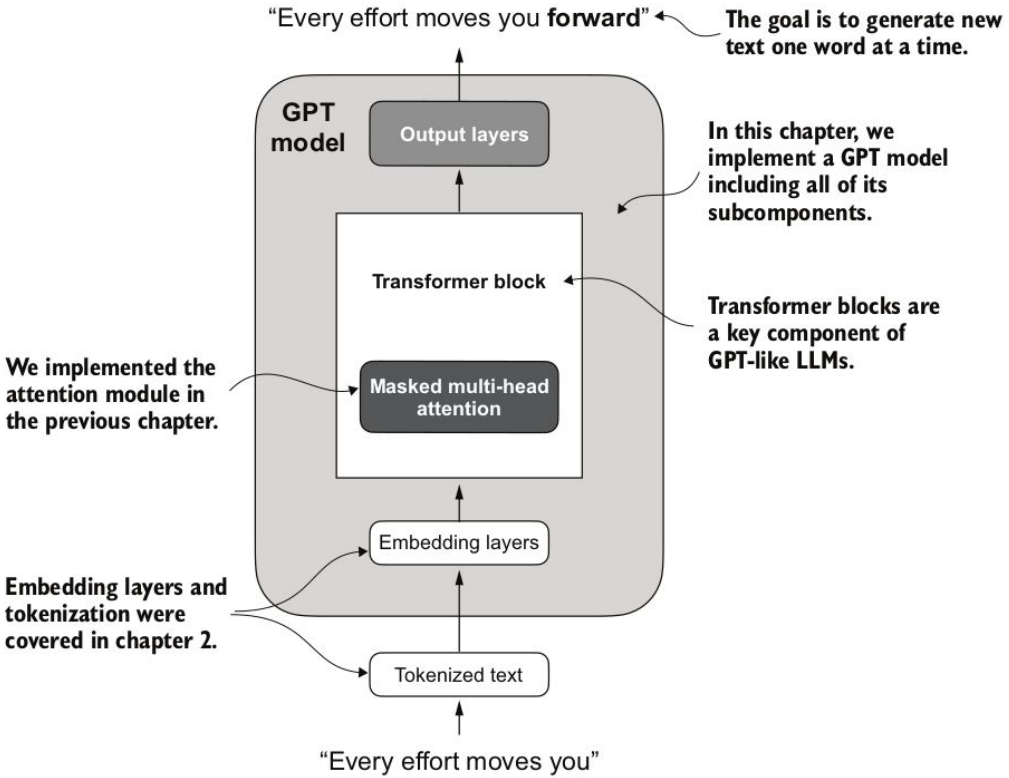
The memory footprint is quadratic in context length!

IMPORTANT

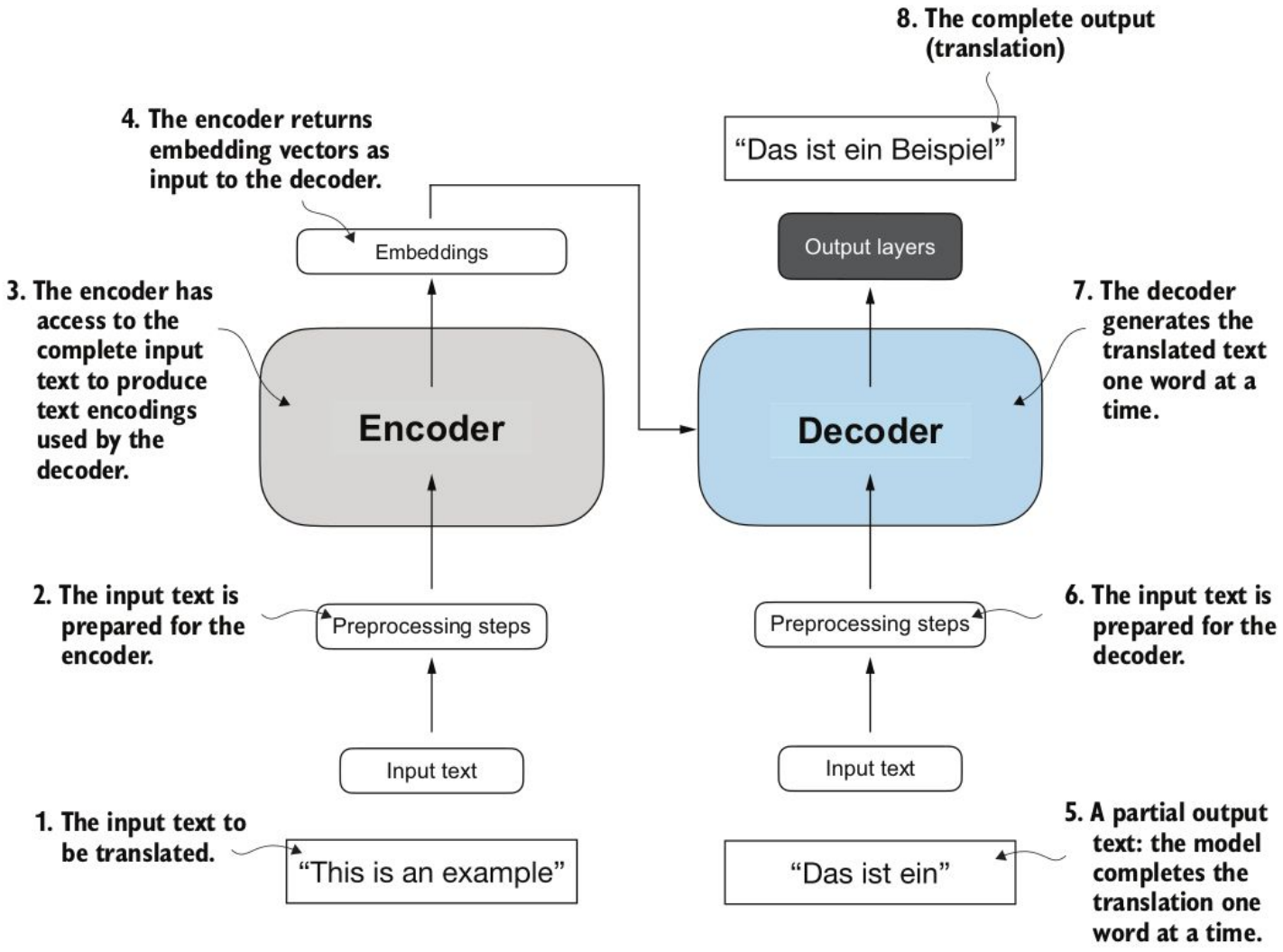
The matrices for computing keys, queries, and values include **trainable** parameters, so the attention mechanism **learns** where to put attention in a **data-driven way.**

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).

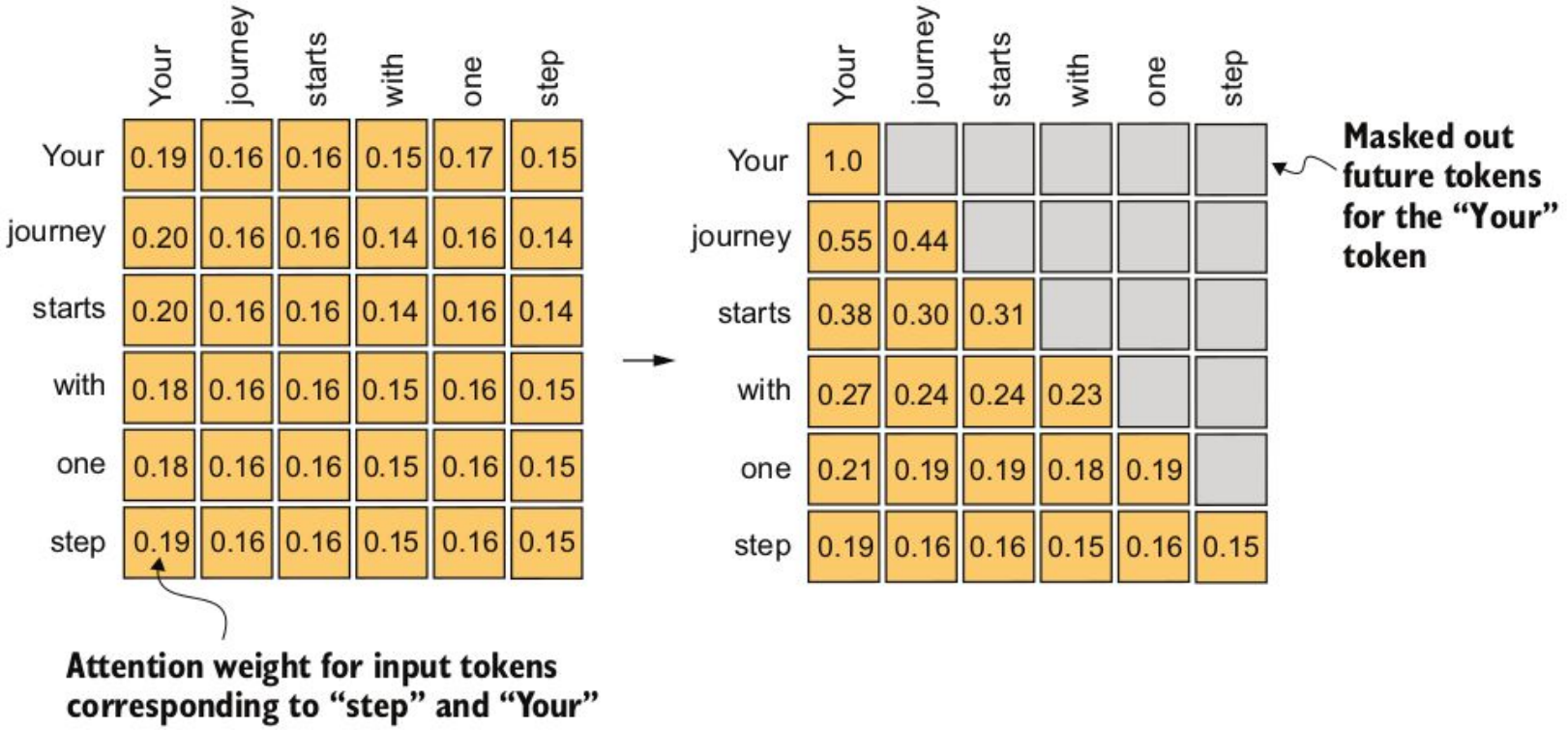
Attention heads as key components in a Transformer



Encoder / decoder

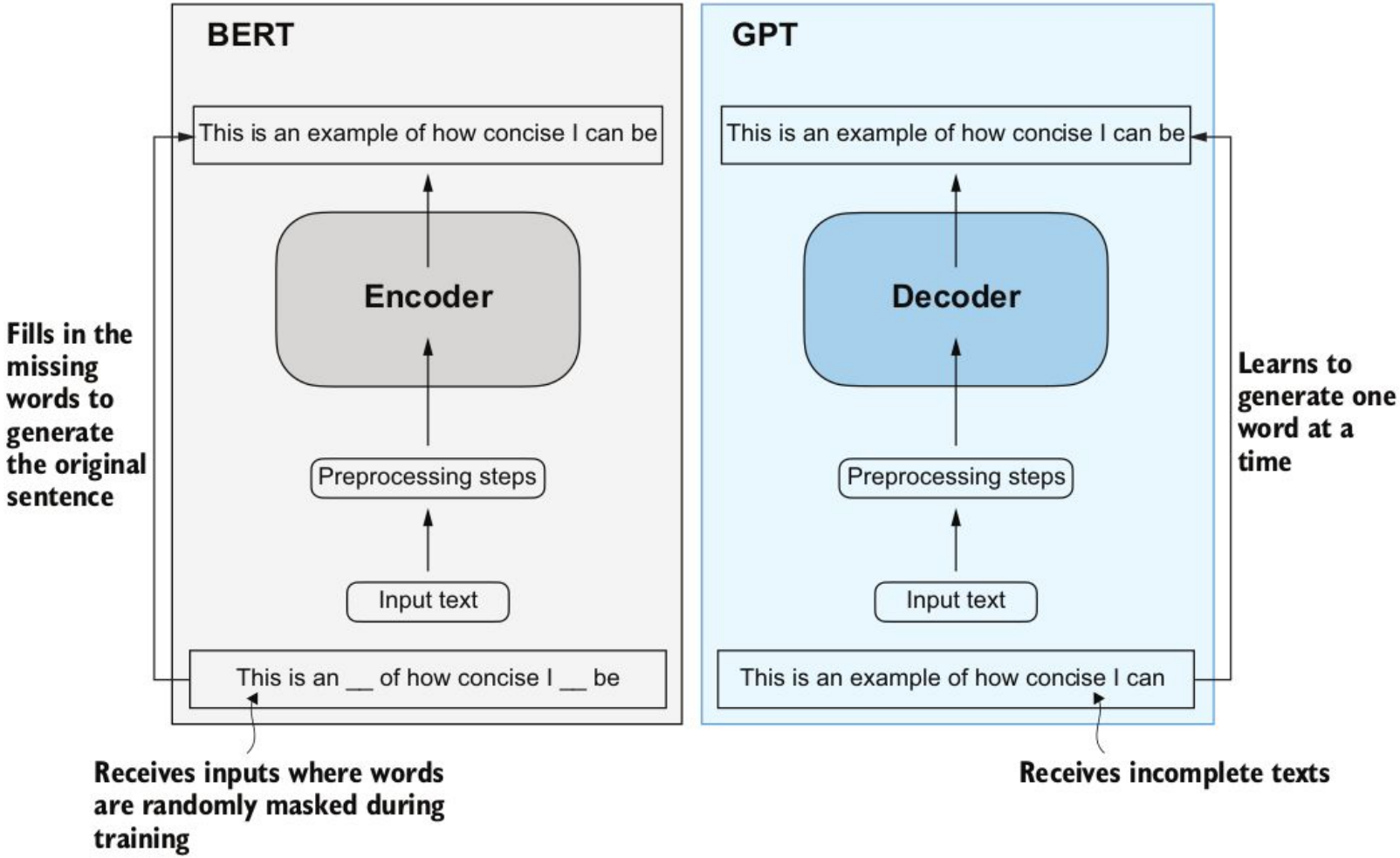

8. The complete output (translation)
"Das ist ein Beispiel"
4. The encoder returns embedding vectors as input to the decoder.
Embeddings
Output layers
3. The encoder has access to the complete input text to produce text encodings used by the decoder.
Encoder
Decoder
7. The decoder generates the translated 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



Implementation of the mask

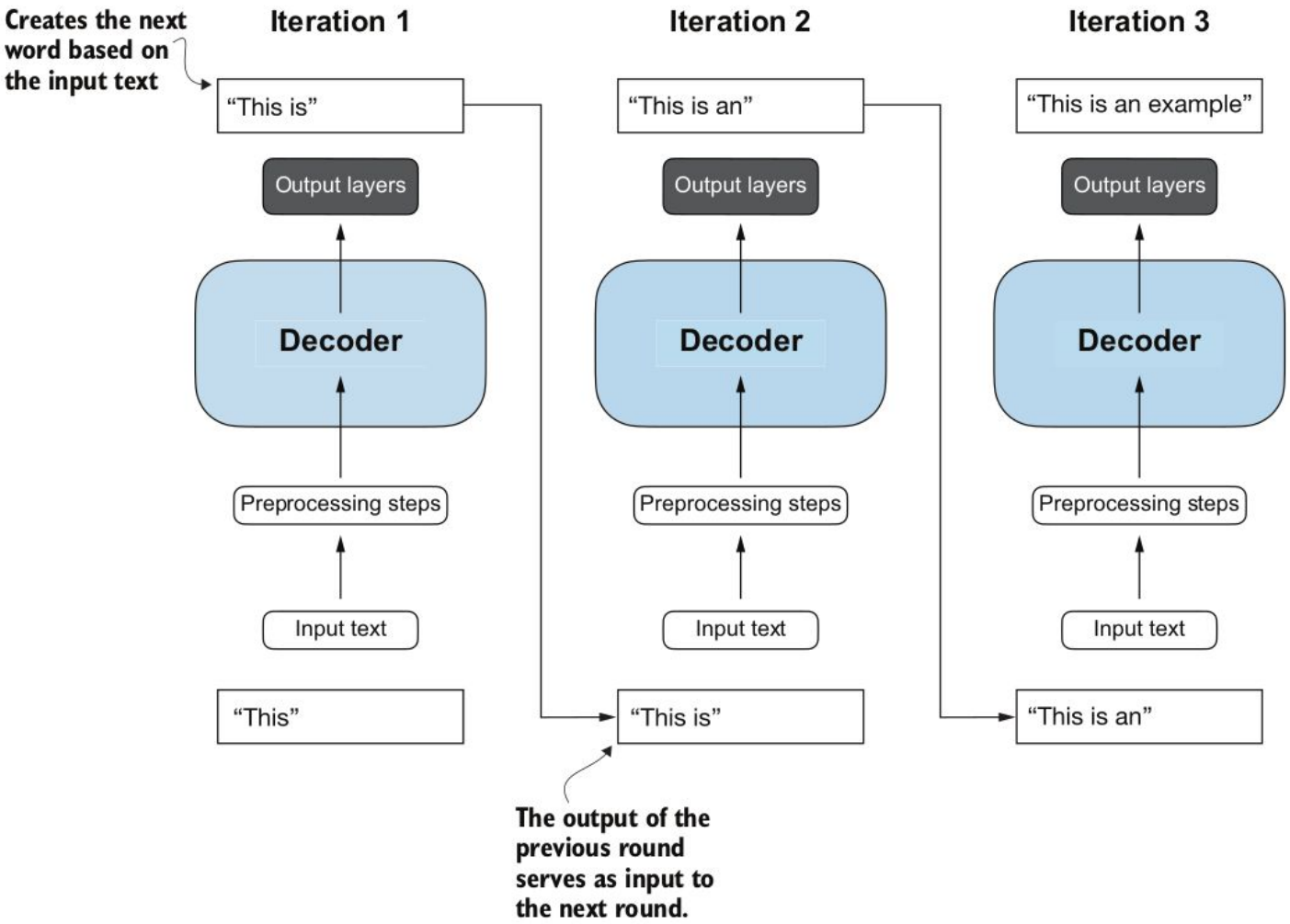
```
x = torch.randn(context_length, input_dim)

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
```


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

Autoregressive models

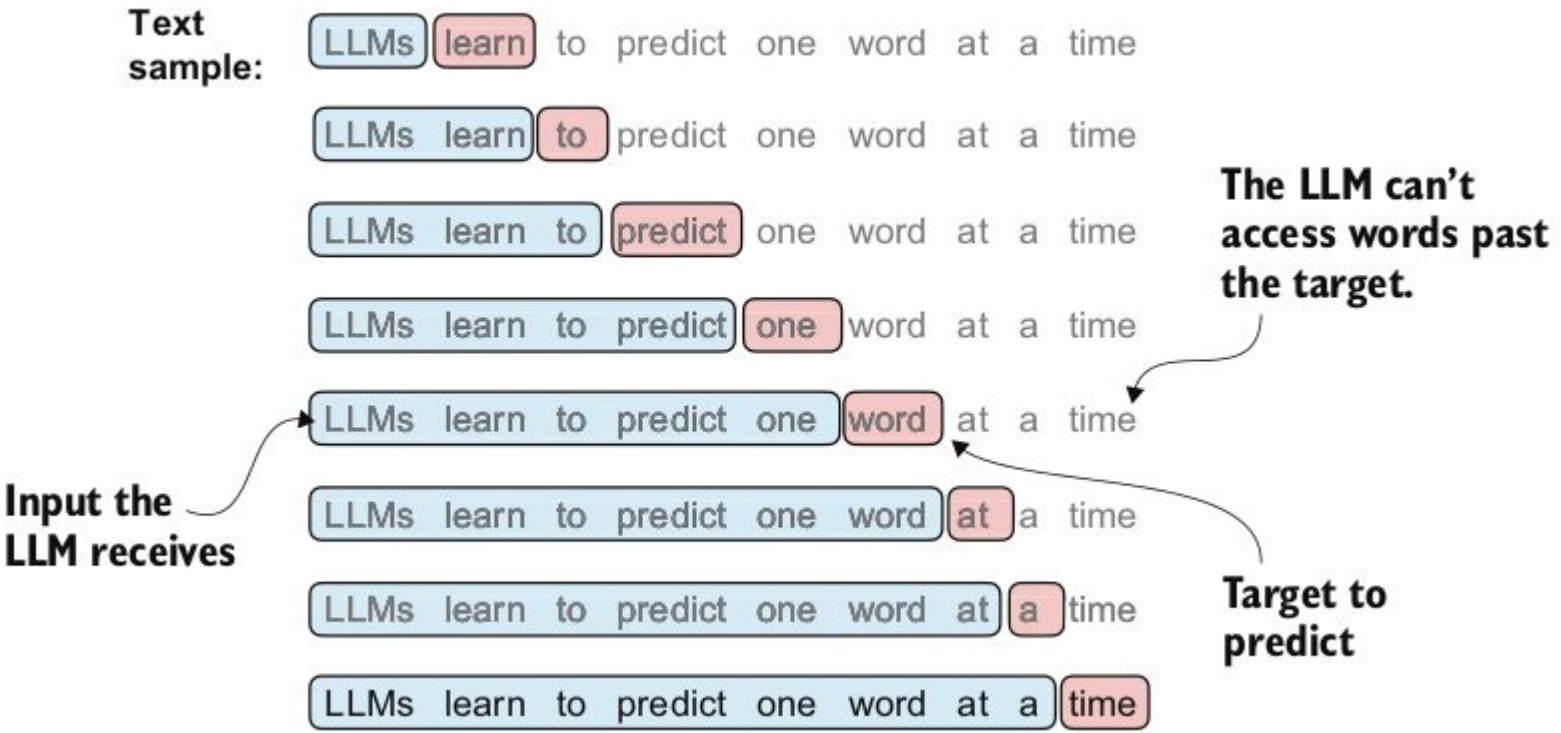
AUTOREGRESSIVE

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
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

Sliding windows



IMPORTANT

A 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 = 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 (without batching)

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)

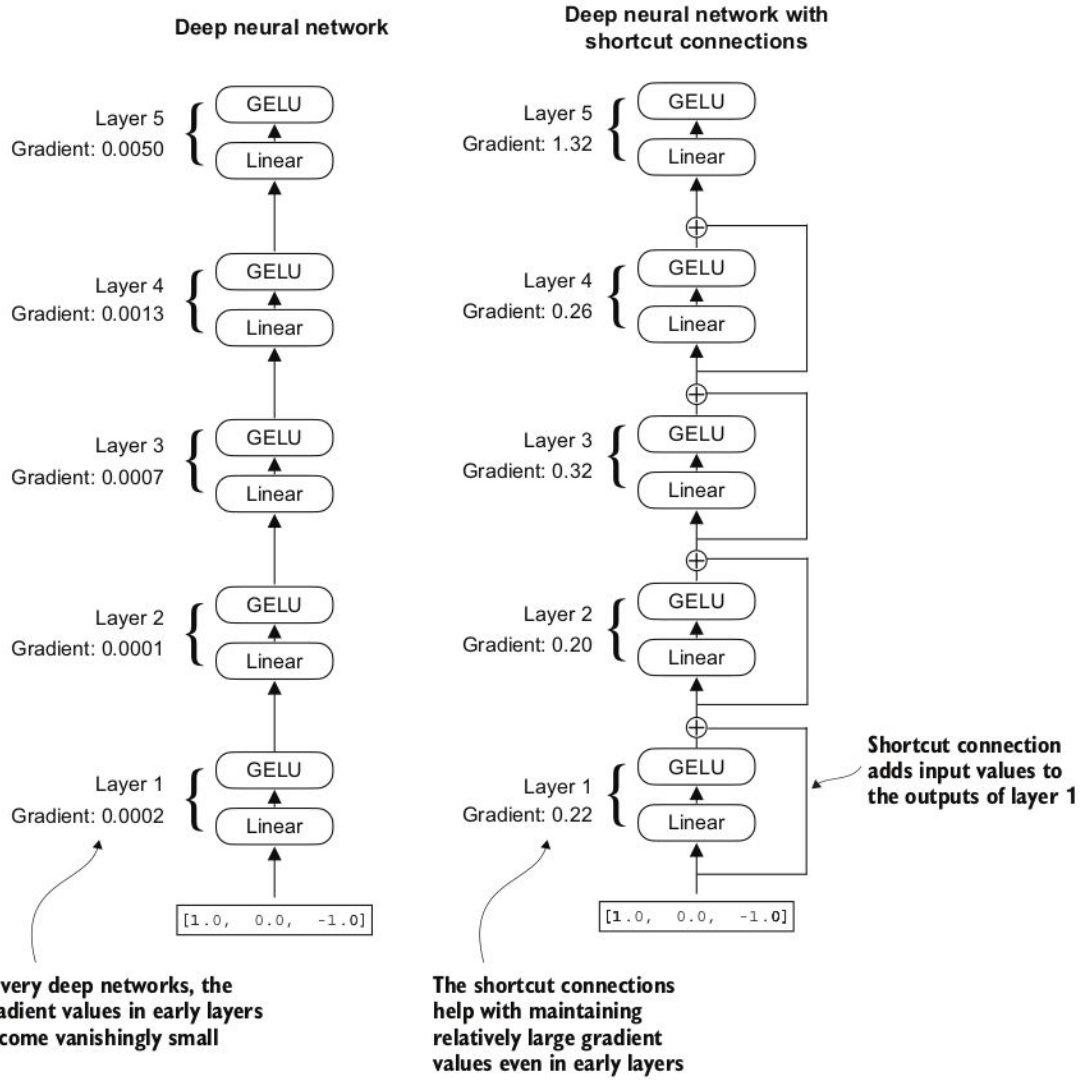
Note: this is called “batch-first”, sometimes the models are “input-first” (just a matter of definitions)

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

Concretely: each computation is added to the input (rather than replacing the input)


Deep neural network
Layer 5
Gradient: 0.0050
GELU
Linear
Layer 4
Gradient: 0.0013
GELU
Linear
Layer 3
Gradient: 0.0007
GELU
Linear
Layer 2
Gradient: 0.0001
GELU
Linear
Layer 1
Gradient: 0.0002
GELU
Linear
[1.0, 0.0, -1.0]
In very deep networks, the gradient values in early layers become vanishingly small
Deep neural network with shortcut connections
Layer 5
Gradient: 1.32
GELU
Linear
Layer 4
Gradient: 0.26
GELU
Linear
Layer 3
Gradient: 0.32
GELU
Linear
Layer 2
Gradient: 0.20
GELU
Linear
Layer 1
Gradient: 0.22
GELU
Linear
[1.0, 0.0, -1.0]
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.

DROPOUT

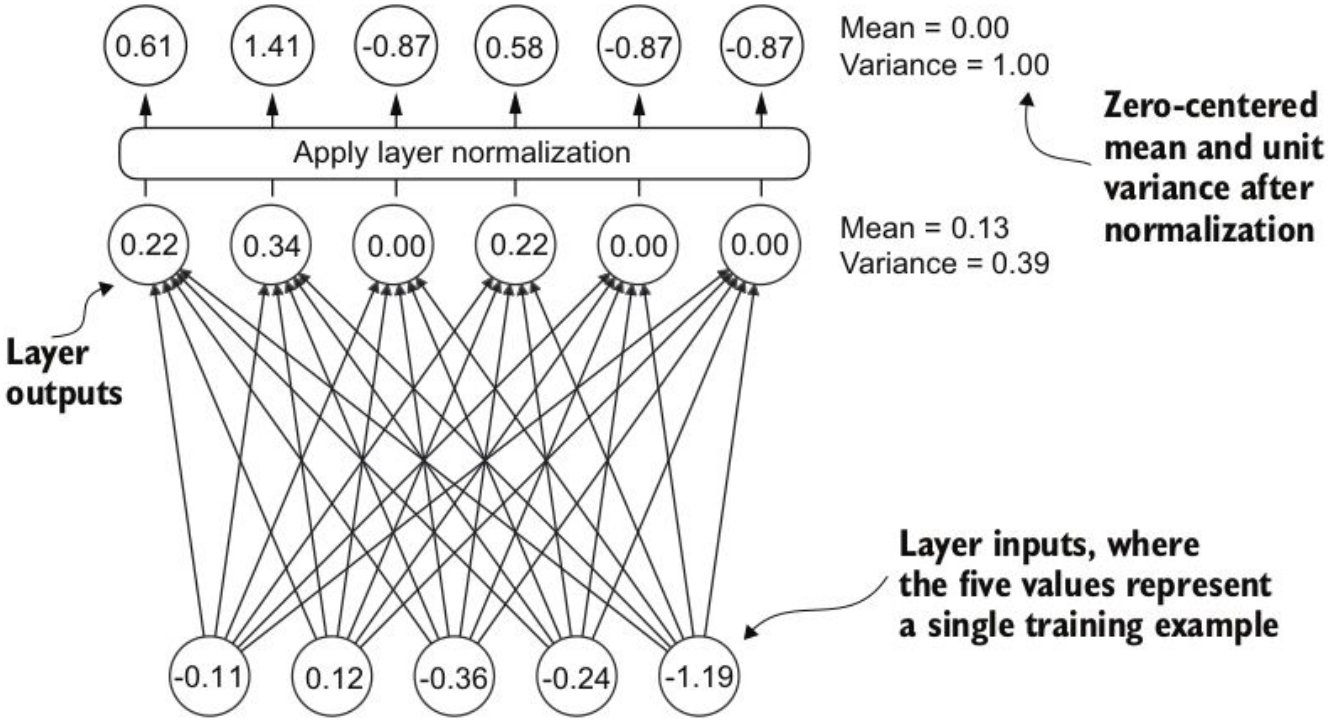
Dropout is only used during training (using *model.train*), it must be deactivated for inference, using *model.eval*

Layer normalization

Why renormalization?

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

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

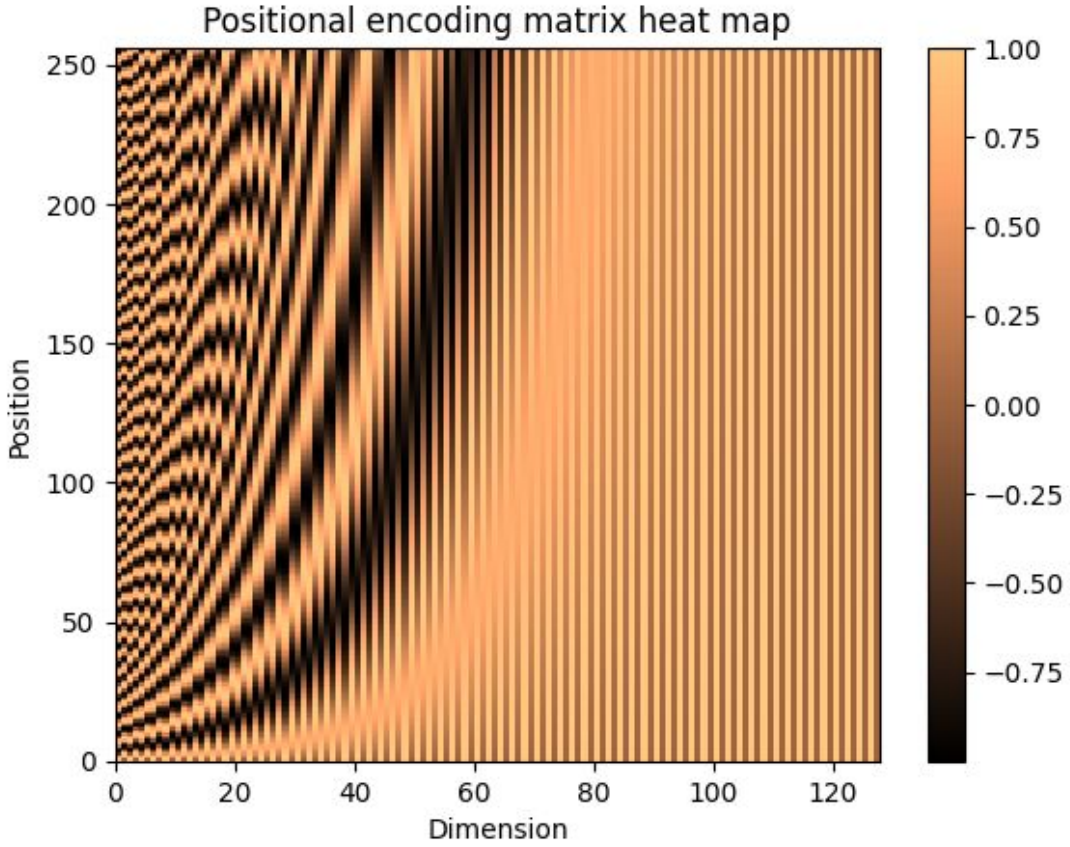
Positional embeddings are added to bring information about position of the tokens.

Simplest version: learned positional embeddings

They are simply added to the token embeddings at the beginning of the model:

```
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 = context_length and d = input_dim

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

- p(pos, 2t) = sin(pos / 10_000^{2t/d})
- p(pos, 2t+1) = cos(pos / 10_000^{2t/d})

Remarks:

- added to the token embeddings (just as learned positional embedding)
- p(c+k,d) is a linear function of p(c,d), suggesting that the model should be able to pick up relative positions

The more recent RoPE (Rotary Positional Embeddings)

An important difference:

- The original Transformer **only** adds positional embedding to the token embeddings
- RoPE adds positional information in each attention head

RoPE rotates embeddings vectors by an angle which depends on the position

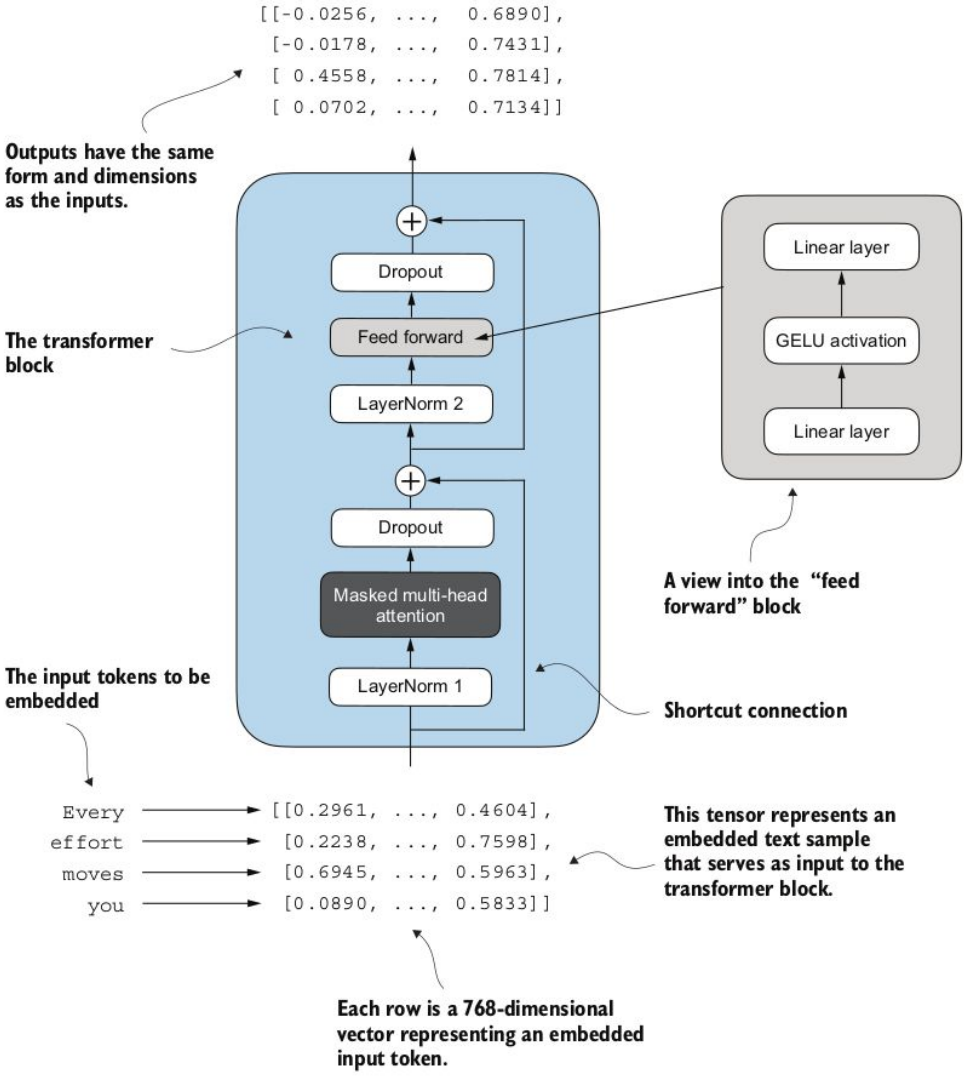
The formula

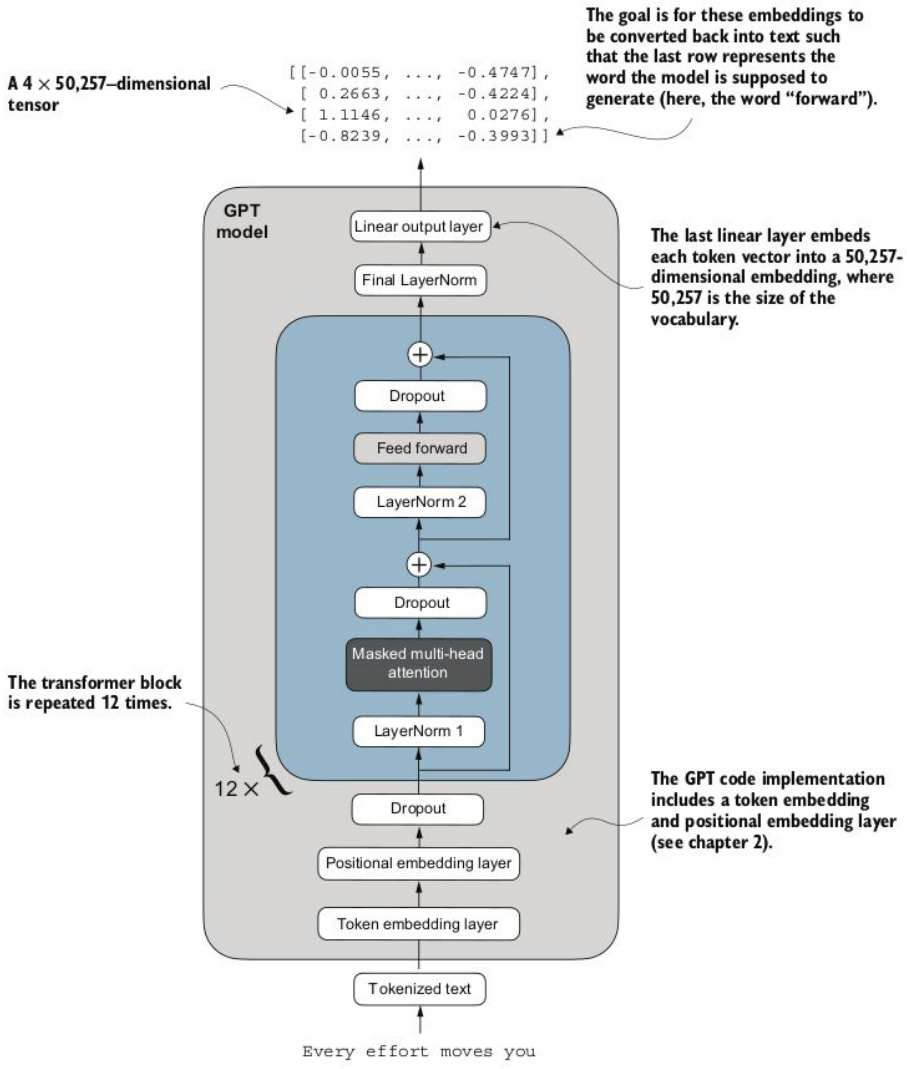
Fix c = context_length and d = input_dim. Let x : (c, d) the embedding vector.

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

$$\begin{pmatrix} \widehat{x}(pos, i) \\ \widehat{x}(pos, i+1) \end{pmatrix} = \begin{pmatrix} cos(pos \cdot \theta_i) & -sin(pos \cdot \theta_i) \\ cos(pos \cdot \theta_i) & sin(pos \cdot \theta_i) \end{pmatrix} \cdot \begin{pmatrix} x(pos, i) \\ x(pos, i+1) \end{pmatrix}$$

Transformer architecture


[[-0.0256, ..., 0.6890],
[-0.0178, ..., 0.7431],
[0.4558, ..., 0.7814],
[0.0702, ..., 0.7134]]
Outputs have the same form and dimensions as the inputs.
The transformer block
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
Linear layer
GELU activation
Linear layer
A view into the "feed forward" block
Shortcut connection
The input tokens to be embedded
Every
effort
moves
you
[[0.2961, ..., 0.4604],
[0.2238, ..., 0.7598],
[0.6945, ..., 0.5963],
[0.0890, ..., 0.5833]]
This tensor represents an embedded text sample that serves as input to the transformer block.
Each row is a 768-dimensional vector representing an embedded input token.


A 4 × 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").
GPT model
Linear output layer
Final LayerNorm
Dropout
Feed forward
LayerNorm 2
Dropout
Masked multi-head attention
LayerNorm 1
12 ×
Dropout
Positional embedding layer
Token embedding layer
Tokenized text
Every effort moves you
The last linear layer embeds each token vector into a 50,257-dimensional embedding, where 50,257 is the size of the vocabulary.
The transformer block is repeated 12 times.
The GPT code implementation includes a token embedding and positional embedding layer (see chapter 2).

Pretraining

Boilerplate training code

```python
@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
```

```python
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

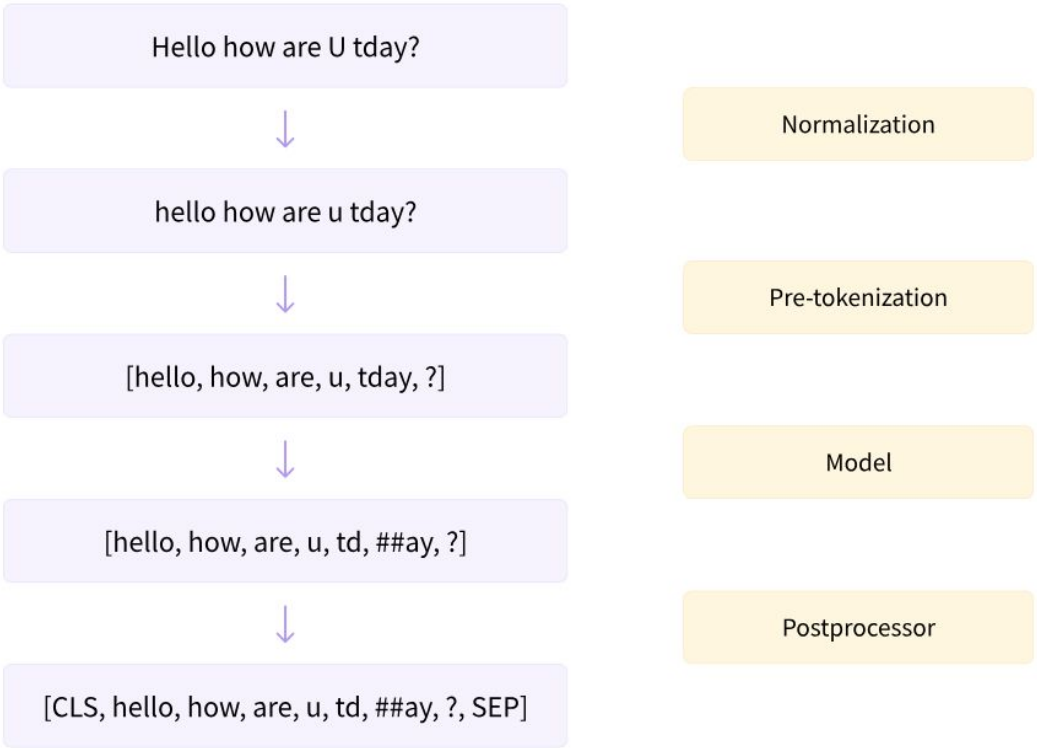
Character encodings:

- 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…)

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.normalizer.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

```python
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, 'Ġis': 2, 'Ġthe': 1, 'ĠHugging': 1, 'ĠFace': 1, 'ĠCourse': 1, '.': 4, 'Ġcha
pter': 1, 'Ġabout': 1, 'Ġtokenization': 1, 'Ġsection': 1, 'Ġshows': 1, 'Ġseveral': 1, 'Ġtokenizer': 1, 'Ġalgorithm
s': 1, 'Hopefully': 1, ',': 1, 'Ġyou': 1, 'Ġwill': 1, 'Ġbe': 1, 'Ġable': 1, 'Ġto': 1, 'Ġunderstand': 1, 'Ġhow': 1,
'Ġthey': 1, 'Ġare': 1, 'Ġtrained': 1, 'Ġand': 1, 'Ġgenerate': 1, 'Ġtokens': 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', 'Ġ']
```

```
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
('Ġ', 'i'): 2
('Ġ', '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)
```

```
('Ġ', 't') 7
```

BPE training algorithm, step 3: Add a merge rule

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

```
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("Ġ", "t", splits)
print(splits["Ġtrained"])
```

```
['Ġt', '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)
```

```
{('Ġ', 't'): 'Ġt', ('i', 's'): 'is', ('e', 'r'): 'er', ('Ġ', 'a'): 'Ġa', ('Ġt', 'o'): 'Ġto', ('e', 'n'): 'en',
('T', 'h'): 'Th', ('Th', 'is'): 'This', ('o', 'u'): 'ou', ('s', 'e'): 'se', ('Ġto', 'k'): 'Ġtok', ('Ġtok', 'en'):
'Ġtoken', ('n', 'd'): 'nd', ('Ġ', 'is'): 'Ġis', ('Ġt', 'h'): 'Ġth', ('Ġth', 'e'): 'Ġthe', ('i', 'n'): 'in', ('Ġa',
'b'): 'Ġab', ('Ġtoken', 'i'): 'Ġtokeni'}
```

```
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', 'Ġ', 'Ġt', 'is', 'er', 'Ġa', 'Ġto', 'en', 'Th', 'This', 'ou', 's
e', 'Ġtok', 'Ġtoken', 'nd', 'Ġis', 'Ġth', 'Ġthe', 'in', 'Ġab', 'Ġtokeni']
```

BPE Tokenization algorithm

```
def tokenize(text):
    pre_tokenize_result = tokenizer._tokenizer.pre_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_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

```python
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', '##fu
ll', '##fully', 'Th', 'ch', '##hm', 'cha', 'chap', 'chapt', '##thm', 'Hu', 'Hug', 'Hugg', 'sh', 'th', 'is', '##thm
s', '##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_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

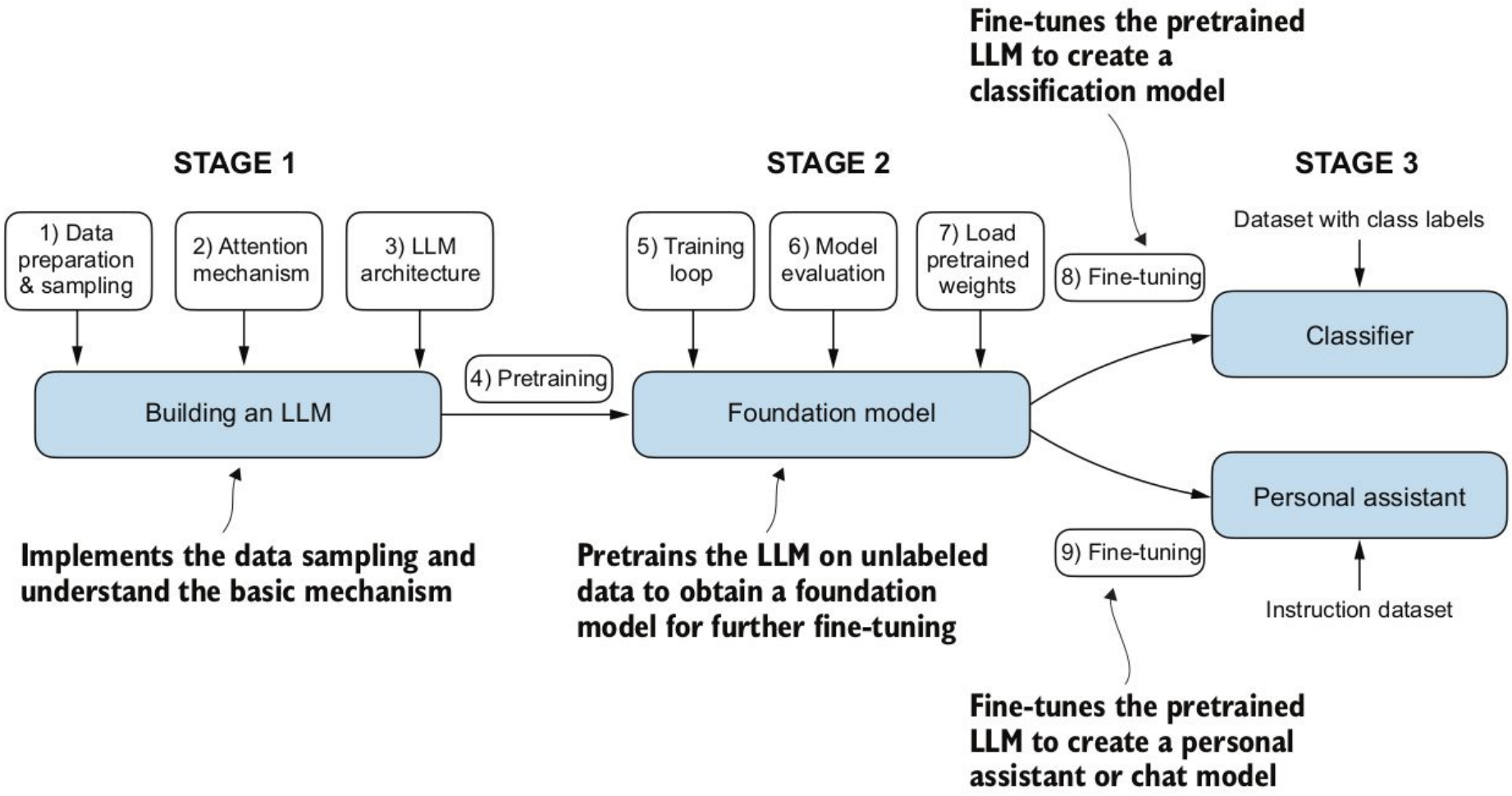
Fine-tuning

- 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…


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
Building an LLM
4) Pretraining
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.

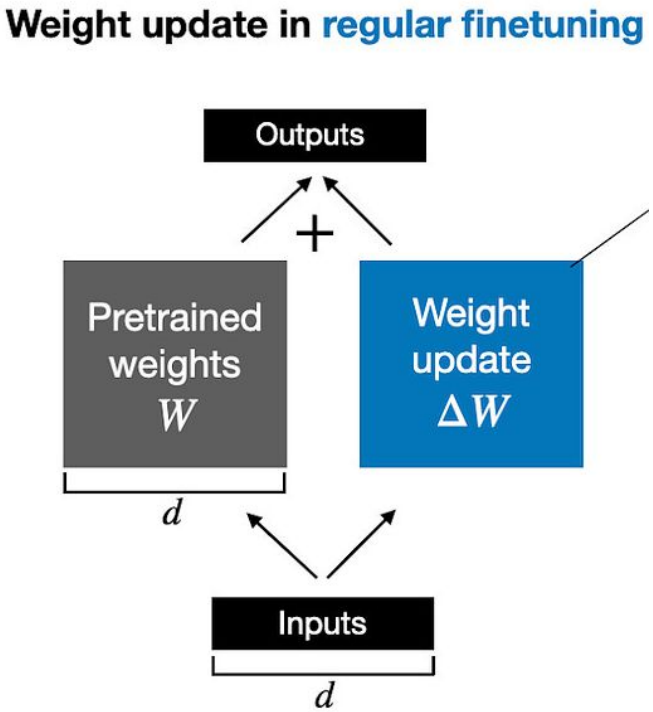
Low-rank Adaptation (LoRA)

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

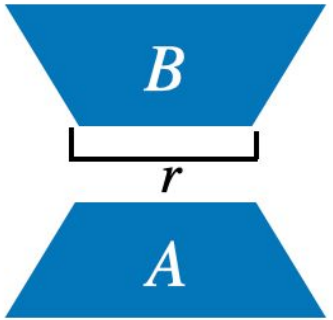


Rank approximations

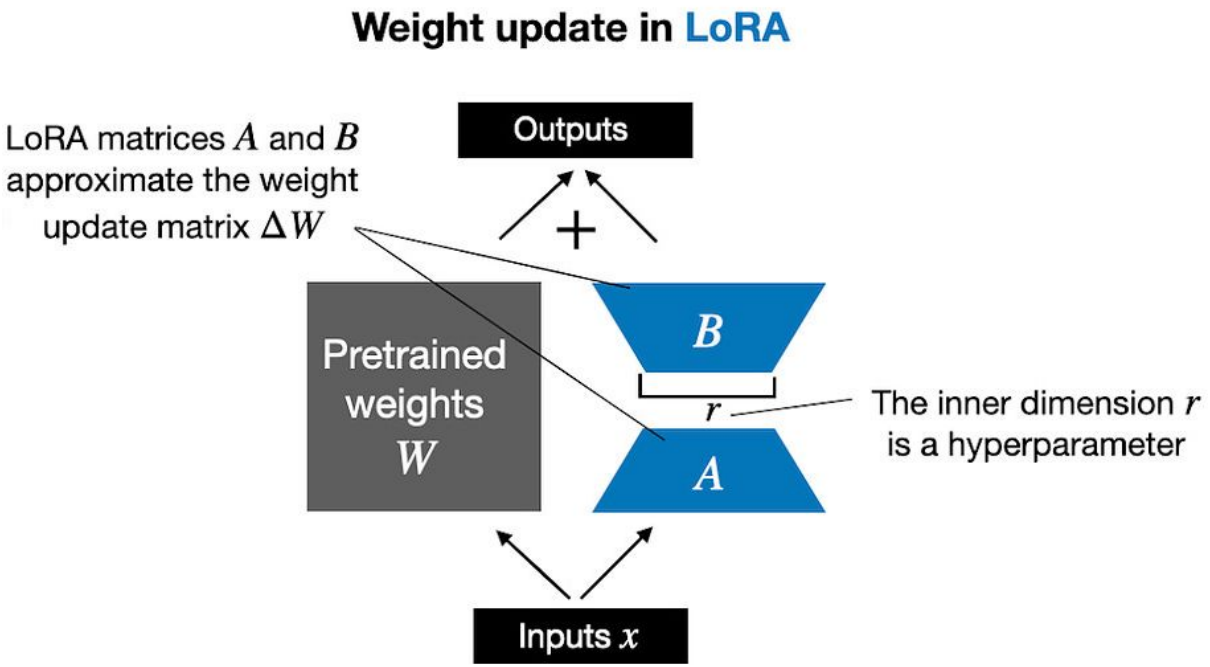
A matrix W of dimension dxd contains dxd parameters. It can be ***rank-r approximated*** by two matrices AxB with:

- A of dimension dxr
- B of dimension rxd

Instead of dxd parameters we now have 2xdxr parameters.



Weight update



LoRA Layer

```python
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)
```

```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
```

```
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/

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…

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

```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.

```python
from transformers import AutoModelForSequenceClassification

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

```python
from transformers import AutoModelForTokenClassification

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

WTF?

```
from transformers import AutoModelForTokenClassification

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


```
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

```python
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.

```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=Tr
ue, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_token': '[SEP]', 'pa
d_token': '[PAD]', 'cls_token': '[CLS]', 'mask_token': '[MASK]'}, clean_up_tokenization_spaces=True),  added_token
s_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_ty
pe_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, 0, 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()
```

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)
```

```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

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…

The DeepSeek special

References

DeepSeek papers:

- MoE: https://arxiv.org/abs/2401.06066 (early 2024)
- GRPO: https://arxiv.org/abs/2402.03300 (early 2024)
- MLA: https://arxiv.org/abs/2405.04434 (mid 2024)
- MTP: https://arxiv.org/abs/2412.19437 (end of 2024)
- DeepSeek-R1: https://arxiv.org/abs/2501.12948 (early 2025)

Implementations:

- Open-R1: https://github.com/huggingface/open-r1
- verl: https://github.com/volcengine/verl
- trl: https://github.com/huggingface/trl

Two directions

(1) Architecture:

- Mixture of Experts (MoE)
- Multi-head Latent Attention (MLA)
- Multi-Token Prediction (MTP)

(2) Group relative policy optimization (GRPO)

Mixture of experts

The general idea of mixture of experts (MoE)

The model is composed of:

- A set of **experts**, which are independent submodels
- A **router** (also called gating network)

The input is fed to the router, which determines a weight for each expert. The input is fed to the K experts with highest weights. Their outputs is aggregated using these weights.

Intuitively, each expert specialises, and the router is able to predict which expert computes relevant information.

Pros and cons

+ **Specialization**: Experts can specialize in different aspects of the problem, leading to better performance than a single, general-purpose model.
+ **Scalability**: MoE can scale to handle very complex problems by adding more experts.
+ **Efficiency**: For a given input, only a subset of the experts needs to be activated, which can improve efficiency compared to a model where all parameters are used for all inputs.
+ **Improved Capacity**: MoEs can have a much larger total capacity (number of parameters) than a single model, without a proportional increase in computational cost per example.
- **Complexity**: Training MoE models can be more complex than training standard models, as the gating network and the experts need to be trained jointly.
- **Data Sparsity**: If the experts are too specialized, they might not receive enough training data, leading to poor performance. This is related to the routing decision.
- **Routing Challenge**: The gating network needs to learn to route inputs effectively. Poor routing can lead to suboptimal performance.

MoE for transformers

- MoE layers replace MLP layers
- Each expert is an MLP
- The router is also an MLP with a softmax

Each token of the input is fed to the router, which determines a weight for each expert. The token is fed to one or two experts with highest weights. Their outputs is aggregated using these weights.

Deepseek's MoE: fine-grained expert segmentation

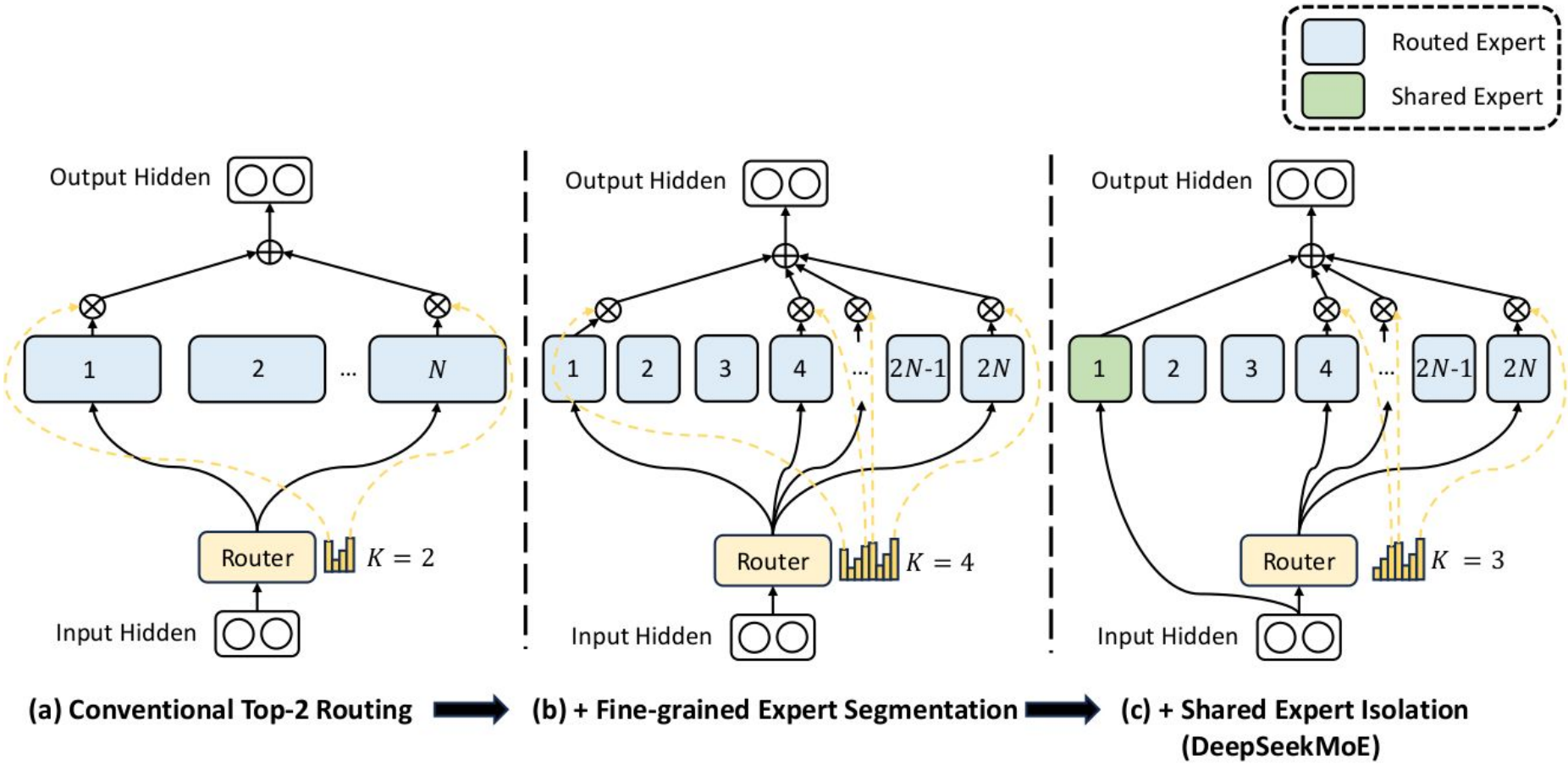
Issue: each token gets sent to a very small number of experts.

Idea: each expert is subdivided into m smaller experts (keeping the number of parameters constant). This way, each token gets sent a m times more experts.

DeepSeek's MoE: shared expert isolation

Issue: tokens assigned to different experts may require common knowledge.

Idea: introduce shared experts that are used for each token.



(a) Conventional Top-2 Routing → **(b) + Fine-grained Expert Segmentation** → **(c) + Shared Expert Isolation (DeepSeekMoE)**

Hidden under the carpet

We need to make sure that:

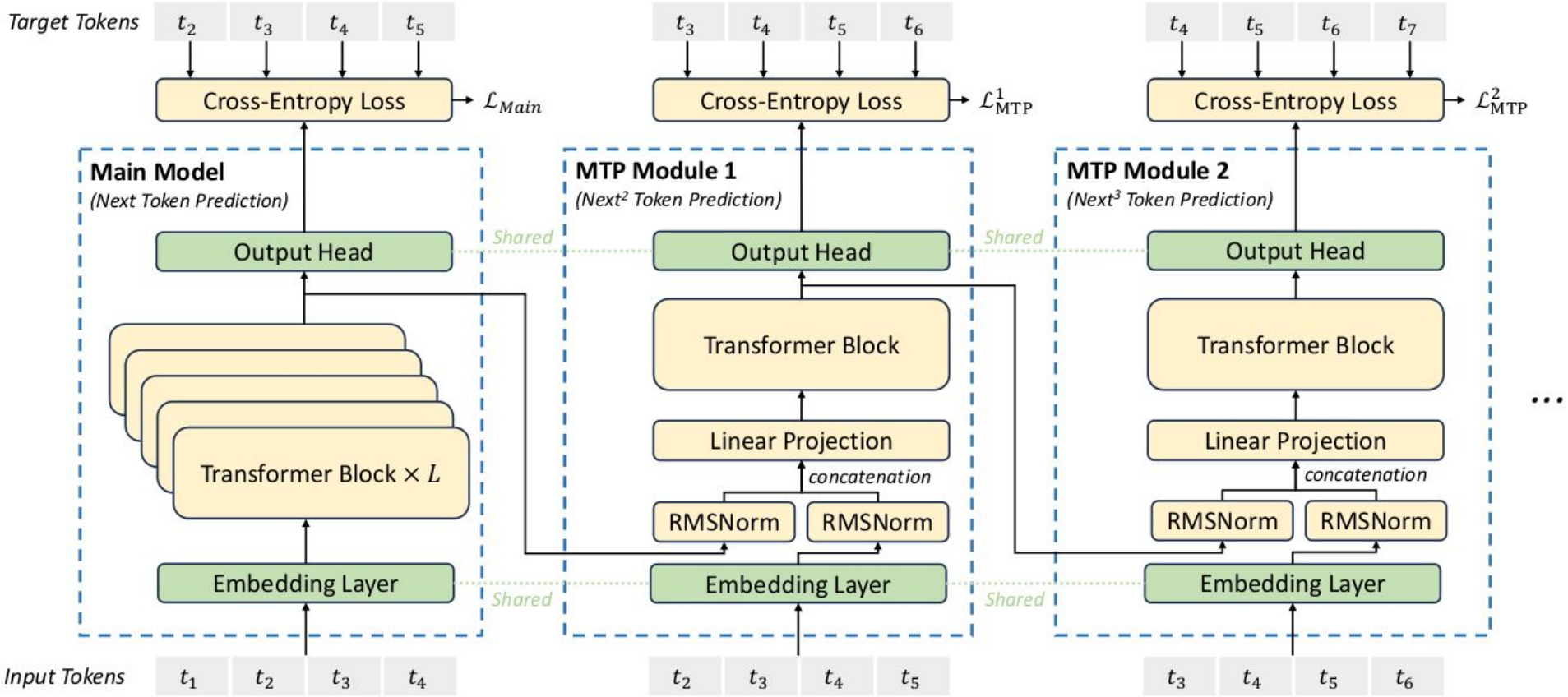
- Each expert gets enough training to avoid *routing collapse*
- Tokens inside a sequence are spread to different experts to avoid *computation bottlenecks*

Classical solution: auxiliary losses, which degrade performances.

DeepSeek's approach: adding dynamic bias for each expert

Multi-token prediction

Can we predict multiple tokens at once?



Multi-head Latent attention

It's a long (and still developing) story

- It starts with KV cache, which caches keys and values when generating long sequences
- But KV cache uses a lot of memory, so different methods were proposed to reduce memory, such as Multi-Query Attention (MQA) vs Grouped-Query Attention (GQA)
- Multi-head Latent Attention (MLA) is another attempt to lower memory, by projecting up and down in a latent space

Some pointers

- https://huggingface.co/blog/kv-cache-quantization
- https://towardsdatascience.com/deepseek-v3-explained-1-multi-head-latent-attention-ed6bee2a67c4/

Group relative Policy optimization (GRPO)

The three stages of understanding

(1) What you tell your grandparents about Deepseek
(2) The high-level ideas
(3) The fineprints

What is the goal?

Short version: post-training **reasoning** models

(not all models need to reason!)

Long version: we start from either a foundation model or an instruct model, and we want to teach the model to **reason** to solve maths, logic, or programming tasks.

Important: We will ask the LLM to think before giving an answer (chain of thoughts).

FOR YOUR GRANDPARENTS

The version for your grandparents (1/4)

I’m teaching my 5 year old daughter additions. Here are three approaches:

- v0: I give her examples and explain for each of them how I perform the addition (“12 + 19 = 31: I first add the units, remember the carry…”)
- v1: I give her questions (“how much is 12 + 13?”) and ask her to give me the result and her reasoning. I reward her when both are correct.
- v2: I give her questions and ask her to think about it and give me the result. I reward her when the answer is correct, ignoring the reasoning.

The version for your grandparents (2/4)

Believe it or not:

- v0 is absolutely useless, she gets bored very quickly with my fathersplaining
- v1 does not work so much either because she doesn’t like me correcting her reasoning, it is a bit too abstract
- v2 works a lot better: the answers become more and more correct over time, although her explanations are not very convincing (even when the result is correct)

Somehow in v2 I rely on her to improve her reasoning, I do not impose my way of reasoning on her

The version for your grandparents (3/4)

- v0 is called “supervised fine-tuning”
- v1 is called “reinforcement learning with human feedback”, because it uses an advanced reward model (me!)
- v2 is a “reinforcement learning without reward model”

The observation about my daughter’s explanations mirrors the DeepSeek’s observations: the model needs reasoning to improve its performance, but it becomes ununderstandable (by humans)

The version for your grandparents (4/4)

To move towards the “group relative policy optimization” developed by DeepSeek, the analogy breaks: I can ask the same question to an LLM and collect different answers (my daughter refuses to do that!)

The algorithm goes as follows: I collect all responses for a fixed question, compute the average score (only based on answer’s correctness), and then reward based on the “advantage” of each response, which is its difference to the average score.

High-level ideas

Different approaches for post-training

- Supervised fine-tuning (SFT)
- Reinforcement Learning from Human Feedback (RLHF)
- Direct Preference Optimization (DPO)
- Group Relative Policy Optimization (GRPO)

Option 1: Supervised fine-tuning (SFT)

The baseline approach:

- **Collect data**: construct a dataset of pairs (prompt, response)
- **Train**: classical fine-tuning, teach the model how to respond to each prompt

Issues with SFT

- Need to have a clean, large dataset
- Not well suited for reasoning: there are many ways of getting to the right answer
- Does not take into account aligning with human preferences
- May induce catastrophic forgetting: we can include a penalty term in the loss for not deviating too much from the original model

Option 1 bis: distillation

Distillation is a technique for training a smaller, faster, and more efficient model (the “student”) by transferring knowledge from a larger, more complex model (the “teacher”)

There are (at least) two different understandings of what this means:

- The traditional one
- The data augmentation one

Option 1 bis: data-augmentation distillation

In “data-augmentation” distillation, the teacher is used to generate the dataset (either both questions and responses, or only responses).

This is particularly interesting for reasoning models, because good reasoning is hard to come by!

Option 1 bis: traditional distillation

In “traditional” distillation, we have two targets:

- The **hard target** is the ground truth
- The **soft target** is the logits of the teacher

The student learns by minimising a loss consisting of two terms:

- Cross-entropy loss to match the hard target
- Distillation loss to match the soft target

Option 2: Reinforcement Learning from Human Feedback (RLHF)

Reference: https://arxiv.org/abs/1909.08593

- **Collect data**: construct a dataset of pairs (prompt, sets of responses)
- **Collect human data**: ask humans for each prompt to rank responses
- **Train a reward model**: the reward model takes as input a prompt and a response, and returns a reward (numerical score)
- **Train the model**: fine-tune the model with an RL algorithm to optimize rewards

Flash introduction to Reinforcement Learning

An agent evolves in an (unknown) environment by taking actions through a policy. In a single step, from state s playing action a we get reward r and go to state s'

The goal of the agent is to maximise the total reward:

π: policy

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Objective:

$$\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right]$$

Which RL algorithm for RLHF?

RLHF is parameterized by the RL algorithm used. Classical choices include:

- Deep Q-Networks (DQN), the classic
- Proximal Policy Optimization (PPO), the default option

Issues with RLHF

- Humans are expensive!
- Training a reward model is hard, unreliable, and costly
- Need to be careful about rewards
- RL itself is hard

A small note

Here we see RLHF just as a fine-tuning algorithm. A slightly different point of view on RLHF:

- **Pre-training**: teaching the LLM language (through language modelling, meaning next token prediction)
- **Fine-tuning**: instructing the LLM on downstream tasks
- **Alignement**: ensures that the model aligns with human values

Option 3: Direct preference optimization (DPO)

Reference: https://arxiv.org/abs/2305.18290

Key idea: get rid of the reward model

- **Collect data**: construct a dataset of pairs (prompt, pairs of responses), with one response preferred the other
- **Train**: maximize the probability of generating preferred responses

Issues with DPO

Requires a human to determine which of the two responses are better

Option 4: Group Relative Policy Optimization (GRPO)

Reference: https://arxiv.org/abs/2402.03300

Key idea: Instead of trying to assign an absolute “goodness” score to each response (like a reward model does), GRPO focuses on relative comparisons within a group of responses.

Option 4: Group Relative Policy Optimization (GRPO)

Key assumption: we can evaluate the final result (**but not the reasoning!**). Examples:

- maths problems: the answer is a number
- code problems: the answer is a piece of code, which can be executed and tested against example inputs
- logical / reasoning problems: the answer is a value

In other words: **rule-based reward model** instead of **model-based reward model**

Option 4: Group Relative Policy Optimization (GRPO)

- **Collect data**: construct a dataset of pairs (prompt, answer)

!!!IMPORTANT!!! It is a lot easier if you do not need to provide the reasoning, just the answer!

- **Group evaluation**: given a prompt, ask the model to generate a group of responses. Evaluate each response only based on the answer (not the reasoning!). Averaging yields a reference point
- **Relative advantage**: Compute the advantage of each response relative to the group
- **Train**: update the probability of generating responses based on their advantages

Observation about GRPO

The first experiment is to fine-tune the V3 model using GRPO, leading to R1-Zero. The results are:

- The performances on answers are impressive
- The thinking time grows larger over the course of the training
- The model learns reflection and develops reasoning approaches
- ***BUT*** explanations become poor and it mixes language

THE FINE PRINTS

Option 1: Supervised fine-tuning (SFT)

$\mathcal{D}$: distribution of pairs (prompt, response)

π_θ: model with parameters θ

Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(x,y)\sim\mathcal{D}}\left[\frac{1}{|y|}\sum_{t=1}^{|y|}\log \pi_\theta(y_t \mid x, y_{<t})\right]$$

Option 2: Reinforcement Learning from Human Feedback (RLHF)

Three steps:

- Learning a reward model
- Adding KL-constraint to the reward
- Applying an RL algorithm

Step 1: Learning a reward model

$\mathcal{D}$: distribution of triples (prompt, better response, worse response)

r_ϕ: reward model with parameters ϕ.

r^*: latent reward model

$r^*(x, y)$ is the reward of response y to prompt x

Bradley-Terry model:

$$p^*(y_1 > y_2 \mid x) = \frac{\exp(r^*(x, y_1))}{\exp(r^*(x, y_1)) + \exp(r^*(x, y_2))}$$

σ: logistic function $\sigma(x) = \frac{1}{1+\exp(-x)}$

Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{(x,y_b,y_w)\sim\mathcal{D}} \left[\log \sigma(r_\phi(x, y_b) - r_\phi(x, y_w))\right]$$

Step 2: Adding a KL-constraint to the reward

We have learned a reward model r(x,y)

Important: r (typically) gives very sparse reward, only when y is entirely generated!

To keep close to the original model, we add a “per-token KL penalty”

Step 2: Adding a KL-constraint to the reward

KL = Kullback-Leibler divergence

D_KL(P || Q) measures how far is the model probabilistic distribution Q far from the true distribution P

See: https://en.wikipedia.org/wiki/Kullback%E2%80%93Leibler_divergence

Step 2: Adding a KL-constraint to the reward

$\mathcal{D}$: distribution of prompts
r: learned reward model
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
RL loss

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(x)}\left[r(x,y)\right]$$

KL-constrained loss

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(x)}\left[r(x,y) - \beta D_{\text{KL}}(\pi_\theta(y \mid x) \mid\mid \pi_{\text{ref}}(y \mid x))\right]$$

Step 2: Adding a KL-constraint to the reward

A small detail:

We use a (differentiable) estimate for the KL term:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(x)}\left[r(x,y) - \beta \sum_{t=1}^{|y|} \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\right]$$

Step 3: Proximal Policy Optimization (PPO)

First: We'll discuss PPO for RL

Second: We'll see what this means for LLMs

Policy gradient methods

PPO is a “policy gradient method”, meaning it iterates over policies and applies gradient descent to improve policies:

π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} r_t \right]$$

The advantage function

Key quantity: the advantage function quantifies the *relative* benefit of an action

π: current policy
$V_\pi(s)$: expected return from s using π
$Q_\pi(s, a)$: expected return from s playing a and then using π
Advantage function:

$$A_\pi(s, a) = Q_\pi(s, a) - V_\pi(s)$$

Computing the gradient

Lemma: π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Gradient:

$$\nabla_\theta \mathcal{L} = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

We use here the advantage because it minimises variance, but we could use other so-called “baselines”. See here for a proof: https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

Advantage estimates

Key question: how do we estimate the advantage?

Probabilistic estimates are about finding a tradeoff between:

- **Bias**: how close if your estimate to the true value? (*unbiased* = equal in expectation)
- **Variance**: how much your estimates depend on chance?

Generalized Advantage Estimation (GAE)

Idea: GAE adds a discount lambda over future steps to balance *bias* (how accurate is the estimate) and *variance* (how the estimates vary).

- For lambda = 0: only considers the next step (*high bias, zero variance*)
- For lambda = 1: considers all future steps (*no bias, high variance*)

Reference: https://arxiv.org/abs/1506.02438

Generalized Advantage Estimation (GAE)

The case lambda = 0:

$V_{\pi,\gamma}$: latent value function for policy π with discount γ

V: approximate value function

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: TD residual of V with discounted γ

If $V = V_{\pi,\gamma}$, then δ_t is an unbiased estimate of $A_{\pi,\gamma}$:

$$\begin{aligned}
\mathbb{E}_{s_{t+1}\sim\pi}\left[\delta_t\right] &= \mathbb{E}_{s_{t+1}\sim\pi}\left[r_t + \gamma V_{\pi,\gamma}(s_{t+1}) - V_{\pi,\gamma}(s_t)\right] \\
&= \mathbb{E}_{s_{t+1}\sim\pi}\left[Q_{\pi,\gamma}(s_t, a_t) - V_{\pi,\gamma}(s_t)\right] \\
&= A_{\pi,\gamma}(s_t, a_t)
\end{aligned}$$

Generalized Advantage Estimation (GAE)

General case

$\lambda \in [0, 1]$ hyperparameter

$V_{\pi,\gamma}$: latent value function for policy π with discount γ

V: approximate value function

$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$: TD residual of V with discounted γ

GAE:

$$\widehat{A}(t, \gamma, \lambda) = \sum_{\ell=0}^{\infty} (\gamma\lambda)^{\ell} \delta_{t+\ell}$$

Generalized Advantage Estimation (GAE)

Long story short for GAE:

- If we have an estimate for the (discounted) value function,
- Then we can estimate the advantage

All neat and tidy, but this means that we need a model for estimating the value function…

Vanilla policy gradient

At this point we have a “vanilla policy gradient” algorithm.

Key issue: after each update we need to recompute the gradient. If we perform multiple updates, we face performance collapse (by diverging too far from the existing policy).

Question: can we learn more from data, making multiple updates on the same datapoint?

Surrogate objective

The surrogate objective focuses on the update

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a)\sim\pi_\theta}\left[\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)} \cdot A_{\pi_\theta}(s,a)\right]$$

Trust region policy optimization (TRPO)

The surrogate objective was introduced for the TRPO algorithm, which added a KL-divergence constraint.

The algorithm is complicated to implement…

Reference: https://arxiv.org/abs/1502.05477

Proximal Policy Optimization (PPO)

The first contribution of PPO: introducing the **clipped surrogate objective**

Key idea: limit the amount the policy can change directly in the objective

Reference: https://arxiv.org/abs/1707.06347

Proximal Policy Optimization (PPO)

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{(s,a)\sim\pi_\theta}\left[\min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)} \cdot A_{\pi_\theta}(s,a),\ g(\epsilon, A_{\pi_\theta}(s,a))\right)\right]$$

$$g(\epsilon, A) = \begin{cases} (1+\epsilon)A & \text{if } A \geq 0, \\ (1-\epsilon)A & \text{if } A < 0. \end{cases}$$

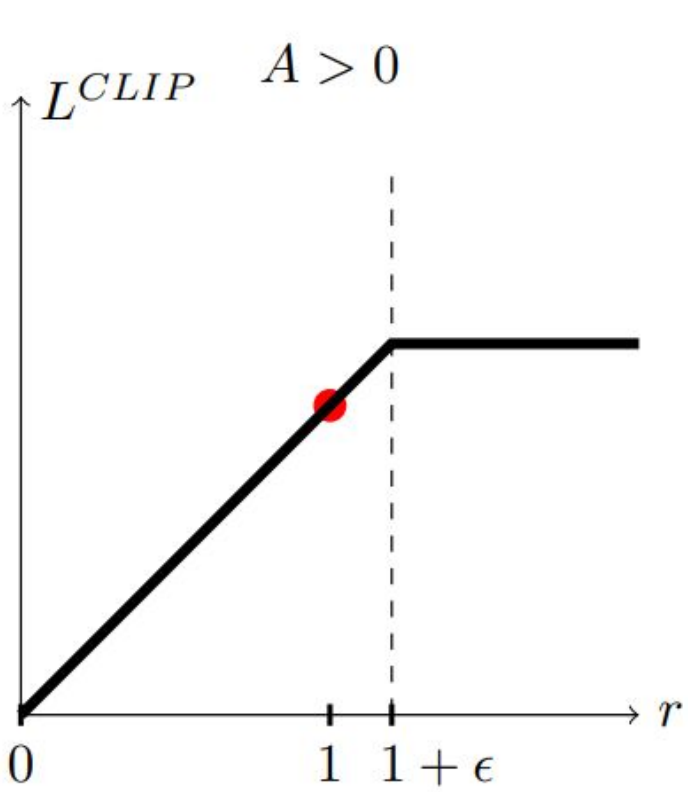
Proximal Policy Optimization (PPO)

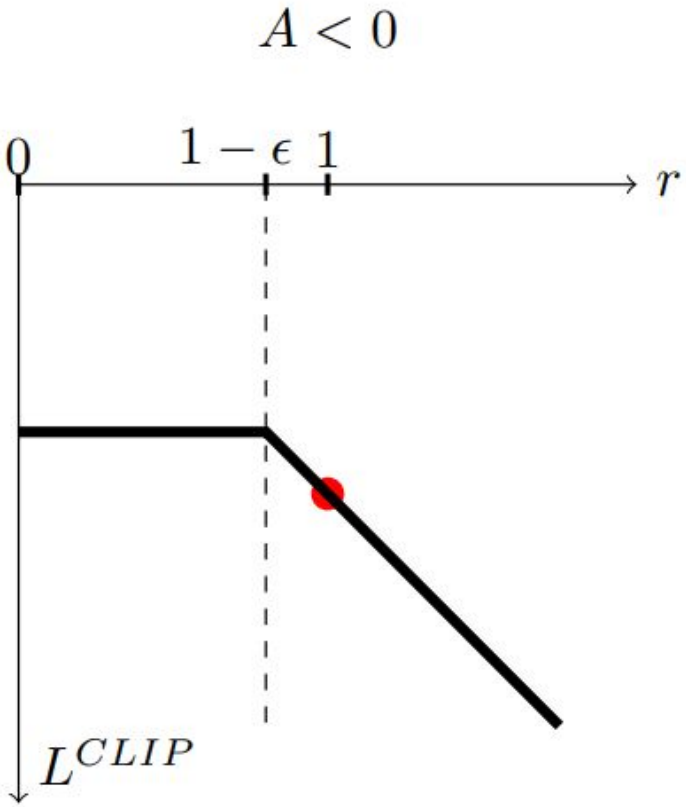
A special case to understand:

π_{ref}: reference policy (frozen)
π_θ: policy with parameters θ
If $A_{\pi_\theta}(s, a)$ is positive:

$$\min\left(\frac{\pi_\theta(a \mid s)}{\pi_{\text{ref}}(a \mid s)},\ 1+\epsilon\right) \cdot A_{\pi_\theta}(s, a)$$

The loss does not grow beyond (1 + eps) A


L^{CLIP}
$A > 0$
r
0
1
$1+\epsilon$


$A < 0$
0
$1-\epsilon$
1
r
L^{CLIP}

Proximal Policy Optimization (PPO)

The consequence of the clipped surrogate objective is that we can perform multiple updates using the same data points!

Even better, they can be performed in parallel.

PPO is the standard out-of-the-box SOTA algorithm for RL.

Step 3: Proximal Policy Optimization (PPO)

Now, let's see what this means for LLMs.

The "state" is the context, the "action" is the next token.

$\mathcal{D}$: distribution of prompts
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
Loss:

$$\mathcal{L}(\theta) = -\mathbb{E}_{x\sim\mathcal{D},y\sim\pi_{\text{ref}}(x)}\left[\frac{1}{|y|}\sum_{t=1}^{|y|}\min\left(\frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\cdot A_{\pi_\theta}((x, y_{<t}), y_t),\ g(\epsilon, A_{\pi_\theta}((x, y_{<t}), y_t))\right)\right]$$

Option 4: Group Relative Policy Optimization (GRPO)

Remember, somewhere in the middle of step 3 for PPO:

- *"We need to estimate advantages (to compute the gradient)"*
- *"GAE can do that if we have estimates for the value function"*

But this is computationally very expensive: it requires training another model for value estimates!

Starting point of GRPO: can we get rid of the value function?

Option 4: Group Relative Policy Optimization (GRPO)

Key idea: the value function is used as a “baseline”.

Here is another (unbiased!) estimate for the value function:

- Given a prompt, generate several responses
- Compute their individual rewards (using a rule-based reward model)
- An estimate of the value function is the average score

Option 4: Group Relative Policy Optimization (GRPO)

$\mathcal{D}$: distribution of prompts
G: number of responses generated for a single prompt
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{x\sim\mathcal{D},(y_i)_{i\in[1,G]}\sim\pi_{\text{ref}}(x)}\left[\frac{1}{G}\sum_{i=1}^{G}F(x,y_i)\right]$$

where

$$F(x,y) = \frac{1}{|y|}\sum_{t=1}^{|y|}\min\left(\frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\text{ref}}(y_t \mid x, y_{<t})}\cdot A_{\pi_\theta}((x,y_{<t}),y_t),\ g(\epsilon, A_{\pi_\theta}((x,y_{<t}),y_t))\right)$$

Option 4: Group Relative Policy Optimization (GRPO)

The advantage is not anymore computed using GAE, it is based on this simple computation:

x: prompt

$y_1, \ldots, y_G$: responses

$r_1, \ldots, r_G$: rewards according to the rule-based reward model

Advantage:

$$A((x, y_{i,<t}), y_{i,t}) = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r}) + \epsilon}$$

Option 4: Group Relative Policy Optimization (GRPO)

Another (minor) difference with PPO:

In PPO, the KL-penalty term for not deviating was added to the reward

In GRPO, the reward is unchanged and the KL-penalty term is added to the loss using a different estimate:

$$D_{\mathrm{KL}}(\pi_\theta(y \mid x) \,||\, \pi_{\mathrm{ref}}(y \mid x)) = \sum_{t=1}^{|y|} \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\mathrm{ref}}(y_t \mid x, y_{<t})} - \log \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\mathrm{ref}}(y_t \mid x, y_{<t})} - 1$$

Option 4: Group Relative Policy Optimization (GRPO)

GRPO is “just” a simpler way of estimating the advantage, it is not specific to the “clipped surrogate objective” (TROP / PPO style), it can also be adapted to the original objective (more like “vanilla policy gradient”). Recall the formula:

π_θ: policy with parameters θ

$\rho = (s_0, a_0, r_0, s_1, a_1, r_1, \dots)$: trajectory

Gradient:

$$\nabla_\theta \mathcal{L} = -\mathbb{E}_{\rho \sim \pi_\theta} \left[\sum_{t=0}^{\infty} A_{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right]$$

Option 3: Direct preference optimization (DPO)

Interesting exercise: understand how the loss is derived, see https://arxiv.org/abs/2305.18290

$\mathcal{D}$: distribution of triples (prompt, better response, worse response)
π_{ref}: reference model (frozen)
π_θ: model with parameters θ
σ: logistic function $\sigma(x) = \frac{1}{1+\exp(-x)}$
Loss:

$$\mathcal{L}(\phi) = -\mathbb{E}_{(x,y^+,y^-)\sim\mathcal{D}}\left[\log\sigma\left(\beta\frac{1}{|y^+|}\sum_{t=1}^{|y^+|}\log\frac{\pi_\theta(y_t^+ \mid x, y_{<t}^+)}{\pi_{\text{ref}}(y_t^+ \mid x, y_{<t}^+)} - \beta\frac{1}{|y^-|}\sum_{t=1}^{|y^-|}\log\frac{\pi_\theta(y_t^- \mid x, y_{<t}^-)}{\pi_{\text{ref}}(y_t^- \mid x, y_{<t}^-)}\right)\right]$$